Министерство образования и науки Российской Федерации

Федеральное государственное автономное
образовательное учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

М. Э. Абрамян

# Введение в стандартную библиотеку шаблонов C++

## Описание, примеры использования, учебные задачи

*Учебник по курсу*
*«Стандартная библиотека C++»*
*для студентов направления 02.03.02*
*«Фундаментальная информатика*
*и информационные технологии» (бакалавриат)*

Ростов-на-Дону – Таганрог
Издательство Южного федерального университета
2017

УДК 004
ББК 32.973
А 13

Печатается по решению учебно-методической комиссии
Института математики, механики и компьютерных наук им. И.И. Воровича
Южного федерального университета (протокол № 4 от 14 апреля 2017 г.)

Рецензенты:
доктор технических наук, профессор кафедры «Информатика»
Ростовского государственного университета путей сообщения (РГУПС)
**М. А. Бутакова**
кандидат физико-математических наук, доцент кафедры алгебры
и дискретной математики Института математики, механики
и компьютерных наук им. И. И. Воровича Южного федерального университета
**С. С. Михалкович**

**Абрамян, М. Э.**

Введение в стандартную библиотеку шаблонов C++. Описание, примеры использования, учебные задачи : учебник / М. Э. Абрамян ; Южный федеральный университет. – Ростов-на-Дону ; Таганрог : Изд-во ЮФУ, 2017. – 177 с.

ISBN 978-5-9275-2374-0

Учебник состоит из трех основных разделов. Первый раздел содержит описание стандартной библиотеки шаблонов C++, во втором приводятся примеры ее применения, а третий представляет собой задачник из 300 учебных заданий, охватывающих все разделы стандартной библиотеки. При описании библиотеки учитываются нововведения стандарта C++11. В четвертом, дополнительном разделе дается обзор средств электронного задачника Programming Taskbook for STL, позволяющих выполнять учебные задания более быстро и эффективно.

Для студентов бакалавриата, обучающихся по направлению подготовки 02.03.02 «Фундаментальная информатика и информационные технологии».

УДК 004
ББК 32.973

ISBN 978-5-9275-2374-0

© М. Э. Абрамян, 2017

# Оглавление

Предисловие ... 5
Раздел 1. Описание библиотеки STL ... 8
1.1. Итераторы ... 8
1.1.1. Общее описание ... 8
1.1.2. Итераторы потоков ввода-вывода ... 9
1.2. Контейнеры ... 10
1.2.1. Общее описание ... 10
1.2.2. Типы, определенные в контейнерах. Параметры конструкторов ... 14
1.2.3. Функции-члены всех контейнеров ... 16
1.2.4. Функции-члены последовательных контейнеров ... 17
1.2.5. Дополнительные функции-члены класса list ... 20
1.2.6. Функции-члены ассоциативных контейнеров ... 21
1.2.7. Вставка и удаление в последовательных контейнерах ... 24
1.2.8. Контейнеры-адаптеры и контейнеры, добавленные в стандарт C++11 ... 26
1.2.9. Дополнение: обратные итераторы ... 32
1.3. Алгоритмы ... 34
1.3.1. Общее описание ... 34
1.3.2. Соглашения об именовании параметров ... 35
1.3.3. Алгоритмы общего назначения ... 36
1.3.4. Численные алгоритмы ... 53
1.3.5. Дополнение: итераторы вставки ... 55
1.4. Стандартные функциональные объекты ... 57
1.4.1. Общее описание ... 57
1.4.2. Функциональные адаптеры ... 58
1.4.3. Функциональные объекты для арифметических и логических операций ... 60
Раздел 2. Использование библиотеки STL ... 62
2.1. Знакомство с итераторами и алгоритмами: STL1Iter17 ... 62
2.1.1. Установка задачника, создание проекта-заготовки и знакомство с заданием ... 62
2.1.2. Выполнение задания ... 68
2.1.3. Другой вариант правильного решения ... 74
2.2. Работа с последовательными контейнерами: STL2Seq22 ... 76
2.2.1. Создание проекта-заготовки и знакомство с заданием ... 76
2.2.2. Выполнение задания ... 78
2.3. Использование алгоритмов ... 81
2.3.1. Преобразование списка с использованием различных видов итераторов: STL3Alg35 ... 81
2.3.2. Преобразование символьных строк: STL4Str27 ... 85

2.4. Использование ассоциативных контейнеров и дополнительных видов функциональных объектов ........ 88
2.4.1. Группировка данных с применением отображений: STL5Assoc20 ........ 88
2.4.2. Дополнительные виды функциональных объектов: STL6Func9 ........ 95
2.5. Применение различных средств стандартной библиотеки C++: STL7Mix4 ........ 103
2.5.1. Создание проекта-заготовки и знакомство с заданием. Дополнительные средства окна задачника, связанные с просмотром файловых данных ........ 103
2.5.2. Выполнение задания ........ 109
2.5.3. Другие варианты решения ........ 113
Раздел 3. Учебные задачи ........ 117
3.1. Знакомство с итераторами и алгоритмами ........ 120
3.2. Последовательные контейнеры ........ 125
3.2.1. Заполнение контейнеров и доступ к элементам. Обратные итераторы ........ 126
3.2.2. Вставка элементов ........ 128
3.2.3. Удаление элементов ........ 130
3.3. Обобщенные алгоритмы ........ 132
3.3.1. Алгоритмы поиска ........ 133
3.3.2. Базовые модифицирующие алгоритмы. Итераторы вставки ........ 136
3.3.3. Сортировка и слияние ........ 139
3.3.4. Перестановки и работа с кучей ........ 140
3.3.5. Численные алгоритмы ........ 142
3.4. Строки как последовательные контейнеры ........ 143
3.5. Ассоциативные контейнеры ........ 146
3.5.1. Множества. Теоретико-множественные алгоритмы ........ 147
3.5.2. Отображения. Группировка и объединение данных ........ 149
3.6. Функциональные объекты: дополнительные возможности ........ 156
3.7. Применение различных средств стандартной библиотеки C++ ........ 162
3.7.1. Избранные задачи из группы STL7Mix ........ 163
3.7.2. Указания к задачам группы STL7Mix ........ 165
Раздел 4. Приложение. Средства ввода-вывода, реализованные в задачнике Programming Taskbook ........ 173
4.1. Поток ввода-вывода pt и связанные с ним итераторы ........ 173
4.2. Вывод отладочной информации: функции Show и ShowLine ........ 174
4.3. Вывод отладочной информации: вспомогательные функции ........ 176
Литература ........ 177

# Предисловие

Книга, предлагаемая вашему вниманию, представляет собой практико-ориентированный учебник по стандартной библиотеке шаблонов языка C++. Для библиотеки шаблонов часто используется название STL (Standard Template Library), которое является неформальным, однако позволяет отличить ее от остальных частей стандартной библиотеки C++. Начиная с 1998 г. библиотека STL входит в стандарт C++ ISO/IEC 14882 (C++98); она содержит средства для создания и преобразования различных структур данных и использует технологию обобщенного программирования. Архитектура библиотеки STL базируется на трех основных компонентах: контейнерах, итераторах и алгоритмах. *Контейнеры* предназначены для хранения наборов объектов в памяти; STL включает две группы контейнеров: последовательные и ассоциативные, в каждую группу входят контейнеры с различными свойствами, что позволяет выбирать контейнер, наиболее подходящий для решения поставленной задачи. *Итераторы* обеспечивают унифицированные средства доступа к содержимому контейнера. Благодаря концепции итераторов, базирующейся на средствах обобщенного программирования, оказалось возможным реализовать универсальные *алгоритмы* – вычислительные процедуры, предназначенные для анализа и преобразования контейнеров. Один и тот же алгоритм может быть применен к любым контейнерам, обладающим требуемыми для этого алгоритма свойствами (точнее, имеющим итераторы того типа, который необходим для корректной работы алгоритма). Еще одной составной частью библиотеки STL являются *функциональные объекты*, представляющие собой обобщения функций и фигурирующие во многих алгоритмах. В пересмотренном стандарте C++ ISO/IEC 14882:2011 (C++11) библиотека STL была дополнена рядом новых возможностей.

Библиотека STL является одной из наиболее трудных для изучения частей стандартной библиотеки C++. Во-первых, это достаточно большая часть стандартной библиотеки: она включает 5 основных видов итераторов, а также их модификации, 7 основных и ряд дополнительных контейнеров, около 70 (в стандарте C++11 – около 90) алгоритмов, большинство из которых реализовано в нескольких вариантах, и большое число стандартных функциональных объектов. Во-вторых, архитектура библиотеки

STL основана на шаблонах – весьма сложном разделе языка C++ [3]. Следует заметить, что особенности механизма шаблонов языка C++ затрудняют поиск и исправление ошибок, допущенных при использовании средств библиотеки STL (в частности, сообщения компилятора об ошибке нередко связываются с фрагментами стандартного программного кода, а не с теми операторами разрабатываемой программы, в которых фактически была допущена ошибка). В то же время библиотека STL относится к тем основным частям стандартной библиотеки, владение которыми является обязательным условием для квалифицированной разработки программ на языке C++.

По библиотеке STL имеется обширная учебная литература, в том числе и на русском языке. Можно отметить книги [2, 4, 6, 7], целиком посвященные STL, а также соответствующие разделы в известных учебниках [5, 8]. Однако очень немногие издания содержат наборы упражнений, позволяющие закрепить полученные знания (в частности, из перечисленных книг упражнения содержат лишь учебники универсального содержания [5, 8]). При этом предлагаемые упражнения не охватывают все возможности библиотеки и являются достаточно сложными, что затрудняет их использование при проведении лабораторных занятий. Настоящее издание призвано восполнить этот пробел. Помимо компактного, но в то же время достаточно подробного описания всех элементов стандартной библиотеки шаблонов, приведенного в разделе 1, а также примеров их применения (которым посвящен раздел 2), оно содержит набор из 300 задач по всем основным разделам стандартной библиотеки и, таким образом, позволяет не только ознакомиться с ее возможностями, но и освоить эту библиотеку на практике. Задачи разбиты на 7 групп; содержание групп, их особенности и формулировки всех задач приведены в разделе 3.

В описании основных компонентов библиотеки STL учитываются нововведения стандарта C++11. Задания ориентированы в основном на базовый вариант библиотеки STL, соответствующий стандарту C++98, однако при их выполнении вполне допустимо (и более удобно) использовать новые возможности, появившиеся в стандарте C++11.

Все задачи, приведенные в книге, входят в состав *электронного задачника Programming Taskbook for STL (PT for STL)*, являющегося одним из дополнений универсального задачника по программированию Programming Taskbook. Задачник PT for STL может использоваться совместно со средами программирования Microsoft Visual Studio 2008, 2010, 2012, 2013, 2015, 2017 и Code::Blocks, начиная с версии 13. Он позволяет генерировать программы-заготовки для выбранных заданий, предоставляет программам наборы тестовых исходных данных, проверяет правильность полученных результатов, диагностирует различные виды ошибок и отображает на экране все данные, связанные с заданием. Все эти возможности

существенно ускоряют выполнение заданий. Особенности применения задачника при выполнении заданий подробно описываются в разделе 2, а дополнительные средства задачника, упрощающие ввод, вывод и отладочную печать данных, – в разделе 4.

Получить дополнительную информацию об электронном задачнике Programming Taskbook и его дополнении Programming Taskbook for STL (а также других его дополнениях) и скачать их дистрибутивы можно на сайте электронного задачника – `http://ptaskbook.com/`.

Автор считает своим приятным долгом выразить искреннюю благодарность Денису Владимировичу Дуброву и Артему Михайловичу Пеленицыну, которые прочитали первый вариант рукописи и высказали много ценных замечаний.

# Раздел 1. Описание библиотеки STL

## *1.1. Итераторы*

### 1.1.1. Общее описание

В библиотеке STL используются пять основных видов итераторов:

- итераторы чтения;
- итераторы записи;
- однонаправленные итераторы;
- двунаправленные итераторы;
- итераторы произвольного доступа.

Для каждого вида итераторов определен набор операций, причем двумя операциями, доступными для всех видов итераторов, являются *операция инкремента* ++, которая передвигает итератор p на следующий элемент последовательности (++p и p++), и *операция разыменования* *, возвращающая значение текущего элемента (*p и вариант p->m для доступа к члену m разыменованного объекта).

Операция разыменования имеет следующие особенности:

- в случае итераторов чтения операция * не может использоваться для изменения элемента;
- в случае итераторов записи операция * не может использоваться для получения значения элемента (выражение *p можно использовать только в левой части присваивания);
- для прочих итераторов операция * может использоваться как для получения значения элемента, так и для изменения этого значения.

*Операции сравнения итераторов на равенство* == и != реализованы для всех итераторов, кроме итераторов записи.

Для однонаправленных итераторов не определяются новые операции (по сравнению с итераторами чтения или записи).

Для двунаправленных итераторов в дополнение к операции инкремента ++ вводится *операция декремента* -- (также в двух видах: --p и p--).

Наконец, для итераторов произвольного доступа добавляются *операция индексирования* [ ], позволяющая сразу обратиться к элементу последовательности с требуемым индексом (p[i]), и *операция смещения на указанное количество элементов*, причем в оба направления (p + i и p - i). Имеется также *операция разности двух итераторов*, позволяющая определить расстояние между элементами, с которыми они связаны (p2 - p1).

Таким образом, набор операций для итераторов произвольного доступа аналогичен набору операций для обычных указателей.

Для итераторов, не являющихся итераторами произвольного доступа, также можно выполнять действия, связанные со смещением и определением расстояния, используя функции из заголовочного файла <iterator>:

- advance(p, n) – передвигает итератор p на n позиций вперед (n >= 0); для двунаправленного итератора можно использовать n < 0 для перемещения назад;
- distance(p1, p2) – возвращает расстояние между итераторами p1 и p2 (в предположении, что расстояние неотрицательно, т. е. что итератор p1 предшествует итератору p2 или совпадает с ним; для двунаправленных итераторов p2 может предшествовать итератору p1, в этом случае расстояние будет отрицательным).

Два итератора обычно используются для задания *диапазона элементов*, при этом предполагается, что первый итератор (*first*) указывает на начальный элемент диапазона, а второй итератор (*last*) указывает на позицию *за* конечным элементом диапазона (причем эта позиция может не быть связана с существующим элементом). Чтобы подчеркнуть отмеченные особенности для диапазонов, определяемых итераторами, они часто записываются в виде *полуинтервала* [*first*, *last*) (левая граница диапазона включается, правая – нет). Полуинтервал [*first*, *first*) не содержит ни одного элемента.

В качестве итераторов чтения и итераторов записи можно использовать итераторы всех остальных видов (однонаправленные, двунаправленные, произвольного доступа); следует лишь учитывать, что итераторы записи можно инкрементировать неограниченно, тогда как итераторы других видов всегда связываются с некоторым диапазоном допустимых элементов. В качестве однонаправленных итераторов можно использовать двунаправленные итераторы и итераторы произвольного доступа, а в качестве двунаправленных итераторов – итераторы произвольного доступа.

Для всех видов итераторов определены их модификации – *константные итераторы*, отличающиеся от обычных тем, что их разыменование дает константное значение.

Особыми итераторами являются *итераторы потоков ввода–вывода* (см. п. 1.1.2), *обратные итераторы* (см. п. 1.2.9) и *итераторы вставки* (см. п. 1.3.4).

### 1.1.2. Итераторы потоков ввода-вывода

Стандартные потоковые итераторы istream_iterator<T> и ostream_iterator<T> (шаблонные классы) определены в заголовочном файле <iterator>.

Имеются два варианта конструктора для *итератора потокового чтения* istream_iterator: вариант с параметром-потоком stream создает итератор для чтения из данного потока, вариант без параметров создает итератор, обозначающий конец потока (все итераторы, обозначающие конец потока,

считаются равными друг другу и не равными никаким другим итераторам потокового чтения).

Ниже перечислены свойства потоковых итераторов чтения:

- тип T определяет тип элементов данных, которые считываются из потока;
- чтение элемента из потока выполняется в начальный момент работы с итератором, а затем при каждой операции инкремента ++;
- имеются два варианта операции ++: префиксный инкремент (++p) и постфиксный инкремент (p++);
- операция * (и ее вариант ->) возвращает последнее прочитанное значение, причем эту операцию можно использовать неоднократно для получения того же самого значения;
- при достижении конца потока итератор становится равным итератору конца потока; последующие вызовы операции инкремента игнорируются, а в результате вызова операции * всегда возвращается значение последнего прочитанного из потока элемента (если же с итератором был связан пустой поток, то результат операции * не определен, хотя и не приводит к аварийному завершению программы).

Для *итератора потоковой записи* ostream_iterator<T> также определены два конструктора: первый конструктор содержит единственный параметр stream, задающий поток вывода, а второй конструктор дополнительно к параметру stream содержит второй параметр delim, задающий *разделитель*, который добавляется в поток вывода после каждого выведенного элемента (если параметр delim не указан, то между выводимыми элементами никакой разделитель не добавляется).

Ниже перечислены свойства потоковых итераторов записи:

- специальный конструктор для создания итератора конца потока вывода не предусмотрен;
- операции * и ++ не выполняют никаких действий и просто возвращают сам итератор;
- операция присваивания p = *выражение* (где p – имя итератора записи) записывает значение выражения в поток вывода.

## *1.2. Контейнеры*

### 1.2.1. Общее описание

Данный раздел посвящен контейнерам, входящим в стандартную библиотеку шаблонов C++. Подробно описываются те основные виды последовательных и ассоциативных контейнеров, с которыми связаны задания, приводимые в книге: это векторы (vector), деки (deque), списки (list), множества (set), мультимножества (multiset), отображения (map) и мультиотобра-

жения (multimap), а также текстовые строки (string), которые относят к *псевдоконтейнерам*. Другие виды контейнеров кратко описываются в п. 1.2.8: это *контейнеры-адаптеры* стек (stack), очередь (queue) и очередь с приоритетом (priority_queue), а также контейнеры, добавленные в библиотеку STL в стандарте C++11 (array, forward_list и ассоциативные контейнеры на базе хеш-функций). Все контейнеры определены в пространстве имен std.

В таблицах 1 и 2 перечислены характеристики основных видов последовательных и ассоциативных контейнеров.

*Таблица 1*

**Последовательные контейнеры**

| Имя | Описание | Итераторы | Заголовочный файл |
|---|---|---|---|
| vector<T> | *Вектор* с элементами типа T | Произвольного доступа | <vector> |
| deque<T> | *Дек* с элементами типа T | Произвольного доступа | <deque> |
| list<T> | *Список* с элементами типа T | Двунаправленные | <list> |
| string | *Строка* с элементами типа char | Произвольного доступа | <string> |

*Таблица 2*

**Ассоциативные контейнеры**

| Имя | Описание | Итераторы | Заголовочный файл |
|---|---|---|---|
| set<Key**[**, Compare**]**> | *Множество* с элементами типа Key и операцией сравнения Compare | Двунаправленные | <set> |
| multiset <Key**[**, Compare**]**> | *Мультимножество* с элементами типа Key и операцией сравнения Compare | Двунаправленные | <set> |
| map <Key, T**[**, Compare**]**> | *Отображение* с ключами типа Key, значениями типа T и операцией сравнения для ключей Compare | Двунаправленные | <map> |
| multimap <Key, T**[**, Compare**]**> | *Мультиотображение* с ключами типа Key, значениями типа T и операцией сравнения для ключей Compare | Двунаправленные | <map> |

В описаниях шаблонов контейнеров, приводимых в таблицах 1 и 2, и далее при описании конструкторов и функций-членов этих контейнеров (см. п. 1.2.2–1.2.6) не указывается дополнительный тип Alloc, который обычно устанавливается по умолчанию. Необязательные параметры заключаются в квадратные скобки, набранные полужирным шрифтом: **[ ]**. В частности, если в шаблоне ассоциативного контейнера не указывается операция сравнения Compare, то она считается равной less<Key>.

Контейнеры могут содержать данные только тех типов T, которые удовлятворяют некоторым естественным условиям (например, в стандарте C++98 требуется, чтобы для типа T был определен конструктор копирования и операция присваивания).

Все рассматриваемые **последовательные контейнеры** допускают вставку новых элементов в любую позицию и удаление элементов из любой позиции. *Векторы* оптимизированы для быстрого (за константное время) выполнения операций вставки и удаления, связанных с концом последовательности элементов (функции-члены push_back и pop_back), а *деки* – для операций, связанных как с началом, так и с концом последовательности (функции-члены push_back и pop_back, push_front и pop_front). В то же время, векторы обладают рядом особенностей, отсутствующих у деков; в частности, они имеют такую характеристику, как *емкость*, которая доступна и для чтения (функция-член capacity) и для изменения (функция-член reserve). *Текстовые строки* string обладают возможностями, аналогичными возможностям векторов с символьными элементами. *Списки* позволяют выполнять быструю вставку и удаление элементов для любой позиции, однако доступ к элементу списка по его номеру требует линейного времени (т. е. зависит от текущего размера списка). По этой причине для списков не реализована операция индексирования, а связанные со списками итераторы являются двунаправленными (а не итераторами произвольного доступа, как для всех остальных последовательных контейнеров). Еще одной особенностью списка является то, что операции вставки и удаления не влияют на корректность итераторов и ссылок, связанных с другими его элементами, в то время как для векторов и деков вставка или удаление элементов может приводить к тому, что некоторые (или все) итераторы и/или ссылки окажутся недействительными (подробности приведены в п. 1.2.7). Кроме того, для списков предусмотрен набор дополнительных функций-членов, отсутствующих у других последовательных контейнеров и представляющих собой оптимизированные реализации соответствующих алгоритмов (см. п. 1.2.5).

Все рассматриваемые **ассоциативные контейнеры** хранят последовательности своих элементов в отсортированном виде. Сортировка выполняется по ключу, причем для множеств и мультимножеств ключами выступают сами элементы (типа T), а в отображениях и мультиотображениях

хранятся пары типа pair<Key, T>, первый компонент которых считается *ключом* (key), а второй – *значением* (value). По умолчанию порядок определяется операцией < для типа ключа Key, однако его можно явно указать в шаблоне контейнера в виде функционального объекта, реализующего бинарный предикат с параметрами типа Key и со свойствами операции сравнения «меньше». Мультимножества и мультиотображения, в отличие от множеств и отображений, позволяют хранить *набор* элементов с эквивалентными ключами (ключи считаются эквивалентными, если ни один из них не является меньшим, чем другой). Для отображения определена операция индексирования с дополнительными возможностями (см. п. 1.2.6). Вставка новых элементов в любой ассоциативный контейнер сохраняет его упорядоченность. И операция вставки, и операция удаления для ассоциативных контейнеров требует логарифмического времени, если для этих операций указывается параметр-ключ. За это же время выполняется и поиск элементов по ключу, для реализации которого в ассоциативных контейнерах предусмотрен целый набор функций-членов. Указанные свойства ассоциативных контейнеров делают их удобным механизмом для группировки и объединения наборов данных по ключу.

Поскольку контейнеры, перечисленные в таблицах 1 и 2, имеют много одинаковых функций-членов, все они далее рассматриваются совместно: в п. 1.2.2 перечисляются типы, связанные с контейнерами, и описываются варианты конструкторов, в п. 1.2.3 приводятся функции-члены, имеющиеся у всех контейнеров, в п. 1.2.4 – функции-члены последовательных контейнеров, в п. 1.2.5 – дополнительные функции-члены списков, в п. 1.2.6 – функции-члены ассоциативных контейнеров. В каждом пункте все функции-члены приводятся в алфавитном порядке их имен. Если некоторые функции-члены имеются не у всех рассматриваемых типов контейнеров, то это явно указывается; кроме того, специальным образом помечаются функции-члены, добавленные в стандарте C++11 (например, текст *vector(C++11), string* означает, что соответствующая функция-член доступна только для классов vector и string, причем для класса vector – только начиная со стандарта C++11). Если один из прежних вариантов функции-члена отсутствует в стандарте C++11, то он помечается текстом *C++98*.

Класс string имеет гораздо больше функций-членов, чем остальные контейнеры, однако в данном разделе приводятся только те из них, которые имеются также и у других последовательных контейнеров.

Если требуется одновременно упомянуть и множество, и мультимножество, то используется слово «(мульти)множество»; если требуется одновременно упомянуть и отображение, и мультиотображение, то используется слово «(мульти)отображение».

В последующих описаниях функций-членов некоторые переменные всегда связываются с данными фиксированного типа (этот тип определяется в самом контейнере – см. п. 1.2.2):

- n имеет тип size_type;
- k имеет тип key_type;
- x (а также x1, x2, …) имеет тип value_type;
- init_list имеет тип *списка инициализации* initializer_list<value_type> (элементы списка инициализации разделяются запятыми, сам список заключается в фигурные скобки);
- pos, hintpos, first и last (а также pos_lst, first_lst, last_lst) имеют тип итератора соответствующего контейнера (iterator).

Переменная other обозначает параметр, являющийся контейнером того же типа, что и контейнер, для которого вызывается функция-член. Переменные InIterFirst и InIterLast обозначают итераторы чтения, которые могут быть связаны с контейнером другого типа (при этом тип элементов контейнера должен совпадать с типом элементов контейнера, для которого вызывается функция-член).

Функции-члены begin, end, rbegin, rend, front, back, at, equal_range, find, lower_bound, upper_bound (а также data для векторов и operator[ ] для *последовательных* контейнеров) реализованы в двух вариантах: неконстантном и константном (например, iterator begin(...) и const_iterator begin(...) const); в дальнейшем это особо не оговаривается и константный вариант не приводится. В стандарте C++11 константные варианты функций-членов begin, end, rbegin, rend можно использовать с именами cbegin, cend, crbegin, crend соответственно.

### 1.2.2. Типы, определенные в контейнерах. Параметры конструкторов

Для доступа к указанным типам используется нотация ::, например vector<int>::iterator.

```
iterator, const_iterator, reverse_iterator,
    const_reverse_iterator
```

Типы *итераторов*, связанные с данным контейнером.

```
reference, const_reference
```

Типы *ссылок* на элементы данного контейнера.

```
pointer, const_pointer
```

Типы *указателей* на элементы данного контейнера.

```
size_type
```

Тип, используемый при указании *размера* контейнера.

`value_type`

Тип *элементов* контейнера (T для последовательных контейнеров, Key для (мульти)множеств, pair<const Key, T> для (мульти)отображений).

*ассоциативные контейнеры*

`key_type`

Тип Key (*элементы-ключи* для (мульти)множеств, *ключи* для (мульти)отображений).

*(мульти)отображения*

`mapped_type`

Тип *значений* T для (мульти)отображений.

*ассоциативные контейнеры*

`key_compare`

Тип функционального объекта, используемого при *сравнении ключей* типа key_type.

*ассоциативные контейнеры*

`value_compare`

Тип функционального объекта, используемого при *сравнении элементов* типа value_type по ключу типа key_type.

Ниже приводятся варианты параметров для конструкторов контейнеров. Большинство вариантов может использоваться для всех видов рассматриваемых контейнеров; единственный особый вариант для последовательных контейнеров помечен соответствующим образом. Следует обратить внимание на вариант конструктора, появившийся в стандарте C++11 и использующий список инициализации.

`конструктор без параметров`

Создает пустой контейнер. Для ассоциативных контейнеров можно дополнительно указать операцию сравнения comp (по умолчанию используется операция сравнения Compare(), взятая из шаблона).

`(InIterFirst, InIterLast)`

Создает контейнер, содержащий элементы (типа value_type) из диапазона [InIterFirst, InIterLast). Для ассоциативных контейнеров можно дополнительно указать операцию сравнения comp (по умолчанию используется операция сравнения Compare(), взятая из шаблона). Для ассоциативных контейнеров вставляемые элементы не обязаны быть упорядоченными, однако если они упорядочены, то время их вставки ускоряется.

`(other)`

Создает копию контейнера other (тип контейнера other должен совпадать с типом создаваемого контейнера). В стандарте C++11 добавлен вариант конструктора с параметром other, обеспечивающий *перемещение* эле-

ментов контейнера other, если контейнер other является *ссылкой на r-значение* (r-value reference; для описания подобной ссылки используется двойной символ &&)

*последовательные контейнеры*

`(n, x = T())`

Создает последовательный контейнер, содержащий n копий значения x. Для строк string обязательными являются оба параметра. В стандарте C++11 вариант с одним параметром оптимизирован таким образом, чтобы избежать создания ненужных копий объектов T.

*C++11*

`init_list`

Конструктор, использующий *список инициализации* (initializer list). Перед списком может указываться символ =. Например, создать вектор с элементами 1, 2, 4 можно с помощью любого из следующих вариантов описания:

```
vector<int> a{1, 2, 4};
vector<int> a = {1, 2, 4};
```

### 1.2.3. Функции-члены всех контейнеров

`operator=(other)`

Удаляет все элементы контейнера и копирует в него все элементы контейнера other (тип контейнера other должен совпадать с типом преобразуемого контейнера). Возвращает полученный контейнер. В стандарте C++11 добавлен вариант операции =, обеспечивающий *перемещение* элементов контейнера other, если контейнер other является ссылкой на r-значение (r-value reference), а также вариант со *списком инициализации* init_list (см. описание последнего варианта конструктора в п. 1.2.2).

`iterator begin()`

Возвращает итератор, указывающий на первый элемент контейнера.

`void clear()`

Удаляет все элементы контейнера.

`bool empty() const`

Возвращает true, если контейнер пуст, и false в противном случае.

`iterator end()`

Возвращает итератор, указывающий на позицию за последним элементом контейнера.

`size_type max_size() const`

Возвращает максимально возможный размер контейнера.

```
reverse_iterator rbegin()
```

Возвращает обратный итератор, связанный с последним элементом контейнера.

```
reverse_iterator rend()
```

Возвращает обратный итератор, связанный с позицией перед первым элементом контейнера.

```
size_type size() const
```

Возвращает текущий размер контейнера.

```
void swap(other)
```

Меняет местами содержимое данного контейнера и контейнера other того же типа.

### 1.2.4. Функции-члены последовательных контейнеров

```
void assign(n, x)
void assign(InIterFirst, InIterLast)
                                                    C++11
void assign(init_list)
```

Удаляет все элементы контейнера и копирует в него новые данные (n копий значения x или элементы из диапазона [InIterFirst, InIterLast)). В стандарте C++11 добавлен вариант с параметром init_list – списком инициализации. Данная функция расширяет возможности, предоставляемые операцией копирования =.

```
                                    vector, deque, string
reference operator[](n)
```

Возвращает ссылку на элемент с индексом n (0 <= n < size()). Выход за границы не контролируется. Для типа string в случае n == size() возвращается символ с кодом 0.

```
                                    vector, deque, string
reference at(n)
```

Возвращает ссылку на элемент с индексом n (0 <= n < size()). Выход за границы приводит к возбуждению исключения out_of_range.

```
                      vector, deque, list, string(C++11)
reference back()
```

Возвращает ссылку на последний элемент контейнера. Для пустого контейнера поведение не определено.

```
                                           vector, string
size_type capacity() const
```

Возвращает текущую емкость контейнера.

*vector(C++11), string*

`T* data()`

Возвращает указатель на внутренний массив, содержащий элементы вектора или символы строки. Для строк реализован только в константном варианте и возвращает константный указатель.

*vector(C++11), deque(C++11), list(C++11)*

`iterator emplace(pos, arg1, arg2, …)`

Вставляет в позицию pos контейнера новый элемент, создавая этот элемент «на месте» и используя при его конструировании параметры arg1, arg2, … . Позволяет избежать дополнительных операций копирования или перемещения, выполняемых при использовании функции-члена insert. Возвращает итератор, указывающий на вставленный элемент.

*vector(C++11), deque(C++11), list(C++11)*

`void emplace_back(arg1, arg2, …)`

Добавляет в конец контейнера новый элемент, создавая этот элемент «на месте» и используя при его конструировании параметры arg1, arg2, … . Позволяет избежать дополнительных операций копирования или перемещения, выполняемых при использовании функции-члена push_back.

*deque(C++11), list(C++11)*

`void emplace_front(arg1, arg2, …)`

Добавляет в начало контейнера новый элемент, создавая этот элемент «на месте» и используя при его конструировании параметры arg1, arg2, … . Позволяет избежать дополнительных операций копирования или перемещения, выполняемых при использовании функции-члена push_front.

```
iterator erase(pos)
iterator erase(first, last)
```

Удаляет элемент на позиции pos или все элементы в диапазоне [first, last) и возвращает итератор, указывающий на элемент, следующий за последним удаленным элементом (или итератор end(), если были удалены конечные элементы контейнера).

*vector, deque, list, string(C++11)*

`reference front()`

Возвращает ссылку на первый элемент контейнера. Для пустого контейнера поведение не определено.

```
iterator insert(pos, x)
void insert(pos, n, x)
```

*C++98*

`void insert(pos, InIterFirst, InIterLast)`

*C++11*

```
iterator insert(pos, InIterFirst, InIterLast)
iterator insert(pos, init_list)
```

Вставляет в контейнер новые данные, начиная с позиции pos (соответственно одно или n значений x, элементы из диапазона [InIterFirst, InIterLast) или элементы из списка инициализации init_list). Первый вариант функции-члена возвращает итератор, указывающий на вставленный элемент. Два последних варианта, добавленных в стандарт C++11 вместо третьего варианта, возвращают итератор, указывающий на первый вставленный элемент, или исходное значение pos, если диапазон или список инциализации являются пустыми.

*vector, deque, list, string(C++11)*

```
void pop_back()
```

Удаляет последний элемент. Для пустого контейнера поведение не определено.

*deque, list*

```
void pop_front()
```

Удаляет первый элемент. Для пустого контейнера поведение не определено.

```
void push_back(x)
```

Добавляет x в конец контейнера.

*deque, list*

```
void push_front(x)
```

Добавляет x в начало контейнера.

*vector, string*

```
void reserve(n)
```

Резервирует емкость размером не менее n.

```
void resize(n, x = T())
```

Изменяет размер контейнера, делая его равным n. Если n > size(), то в конец контейнера добавляется требуемое число копий x. Если n < size(), то удаляется требуемое количество конечных элементов контейнера. В стандарте C++11 вариант с одним параметром оптимизирован таким образом, чтобы избежать создания ненужных копий объектов T.

*vector(C++11), deque(C++11), string(C++11)*

```
void shrink_to_fit()
```

Позволяет уменьшить размер памяти, выделенной для хранения элементов контейнера, однако не гарантирует, что память действительно будет уменьшена.

### 1.2.5. Дополнительные функции-члены класса list

Все дополнительные функции-члены класса list, кроме splice, представляют собой специальные реализации соответствующих алгоритмов, которые необходимо использовать вместо стандартных алгоритмов при обработке списков.

```
void merge(lst[, comp])
```

Выполняет операцию слияния текущего списка и списка lst того же типа (оба списка должны быть предварительно отсортированы). При слиянии элементы сравниваются с помощью операции < или предиката comp, если он явно указан (и эта же операция или предикат должны быть ранее использованы для сортировки списков). Слияние является *устойчивым*, т. е. относительный порядок следования элементов исходных списков не нарушается. Если «одинаковые» элементы присутствуют как в текущем списке, так и в списке lst, то элемент из lst помещается *после* элемента, уже присутствующего в текущем списке. В результате слияния список lst становится пустым. В стандарте C++11 добавлены варианты с параметром lst, являющимся ссылкой на r-значение (r-value reference).

```
void remove(x)
void remove_if(pred)
```

Удаляет из списка, соответственно, все вхождения элемента x или все элементы, для которых предикат pred возвращает значение true.

```
void reverse()
```

Изменяет порядок элементов списка на обратный.

```
void sort([comp])
```

Выполняет сортировку списка, используя операцию < или предикат comp, если он явно указан. Сортировка является *устойчивой*, т. е. относительный порядок элементов с одинаковыми ключами сортировки не изменяется.

```
void splice(pos, lst)
void splice(pos, lst, pos_lst)
void splice(pos, lst, first_lst, last_lst)
```

Перемещает элементы из списка lst в текущий список (элементы размещаются, начиная с позиции pos). Перемещаются, соответственно, все элементы списка lst, элемент списка lst, расположенный на позиции pos_lst, и элементы списка lst из диапазона [first_lst, last_lst) (если текущий список совпадает со списком lst, то итератор pos не должен входить в диапазон [first_lst, last_lst)). В стандарте C++11 добавлены варианты с параметром lst, являющимся ссылкой на r-значение (r-value reference).

`void unique([pred])`

Удаляет соседние «одинаковые» элементы списка, оставляя первый из набора «одинаковых» элементов. Для сравнения элементов используется операция == или предикат pred, если он явно указан.

### 1.2.6. Функции-члены ассоциативных контейнеров

*map*

`T& operator[](k)`

Возвращает ссылку на значение, связанное с ключом k. Если ключ k отсутствует в контейнере, то в контейнер добавляется пара с ключом k и значением по умолчанию T(), и операция [] возвращает ссылку на это значение. Фактически данная операция возвращает следующее выражение: insert(make_pair(k, T())).first->second. В стандарте C++11 добавлен вариант с параметром k, являющимся ссылкой на r-значение (r-value reference).

*map(C++11)*

`T& at(k)`

Возвращает ссылку на значение, связанное с ключом k. Если ключ k отсутствует в контейнере, то возбуждается исключение out_of_range.

`size_type count(k) const`

Возвращает число ключей со значением k. Для множества и отображения это либо 0, либо 1; для мультимножества и мультиотображения возвращаемое значение может быть больше 1.

*set(C++11), map(C++11)*

`pair<iterator, bool> emplace(arg1, arg2, …)`

*multiset(C++11), multimap(C++11)*

`iterator emplace(arg1, arg2, …)`

Вставляет в контейнер новый элемент, создавая этот элемент «на месте» и используя при его конструировании параметры arg1, arg2, … . Позволяет избежать дополнительных операций копирования или перемещения, выполняемых при использовании функции-члена insert. Если элемент с указанным ключом уже имеется в контейнере, то в случае множества и отображения попытка вставки игнорируется. Возвращает итератор, указывающий на вставленный элемент, а также (в варианте для множества и отображения) логическое значение, определяющее, была ли произведена вставка. Если вставка не была произведена из-за того, что в контейнере (множестве или отображении) уже существует элемент с таким же ключом, то возвращается позиция уже имеющегося элемента с этим ключом.

*C++11*

```
iterator emplace_hint(hintpos, arg1, arg2, …)
```

Вставляет в контейнер новый элемент, создавая этот элемент «на месте» и используя при его конструировании параметры arg1, arg2, … . Позволяет избежать дополнительных операций копирования или перемещения, выполняемых при использовании функции-члена insert. Параметр hintpos является «подсказкой» для позиции вставки: элемент x вставляется максимально близко *перед* позицией hintpos. Возвращает итератор, указывающий на вставленный элемент (если элемент с указанным ключом уже имеется в контейнере, то в случае множества и отображения попытка вставки игнорируется и возвращается позиция уже имеющегося элемента с этим ключом).

```
pair<iterator, iterator> equal_range(k)
```

Возвращает результат вызова функций lower_bound и upper_bound в виде пары итераторов: make_pair(lower_bound(k), upper_bound(k)).

```
size_type erase(k)
```

*C++98*

```
void erase(pos)
void erase(first, last)
```

*C++11*

```
iterator erase(pos)
iterator erase(first, last)
```

Удаляет элемент (элементы) с ключом k, элемент в позиции pos или все элементы в диапазоне [first, last). В первом случае возвращает количество удаленных элементов (для множества и отображения это либо 0, либо 1). Два последних варианта, добавленных в стандарт C++11 вместо двух предыдущих вариантов, возвращают итератор, указывающий на элемент, следующий за последним удаленным элементом (или итератор end(), если были удалены конечные элементы контейнера).

```
iterator find(k)
```

Ищет ключ k и возвращает итератор, указывающий на соответствующий элемент контейнера, или end(), если ключ не найден. В случае мультимножества и мультиотображения итератор может указывать на любой из элементов с ключом k.

*set, map*

```
pair<iterator, bool> insert(x)
```

*multiset, multimap*

```
iterator insert(x)
```

*set, map, multiset, multimap*

```
iterator insert(hintpos, x)
void insert(InIterFirst, InIterLast)
```

*C++11*

```
void insert(init_list)
```

Вставляет в контейнер новые данные. Если данные с указанным ключом уже имеются в контейнере, то в случае множества и отображения попытка вставки игнорируется. Во всех вариантах, кроме двух последних, функция возвращает позицию вставленного элемента, а также (в первом варианте, имеющемся только у множества и отображения) логическое значение, определяющее, была ли произведена вставка. Если вставка не была произведена из-за того, что в контейнере (множестве или отображении) уже существует элемент с таким же ключом, то возвращается позиция уже имеющегося элемента с этим ключом. Параметр hintpos является «подсказкой» для позиции вставки: элемент x вставляется максимально близко к позиции hintpos (в стандарте C++11 уточняется, что вставка выполняется *перед* позицией hintpos). Вариант с параметрами InIterFirst, InIterLast обеспечивает вставку всех элементов из диапазона [InIterFirst, InIterLast); эти элементы не обязаны быть упорядоченными по ключу, однако если они упорядочены, то время их вставки уменьшается. Вариант с параметром init_list (списком инициализации) вставляет в контейнер все элементы из указанного списка; этот вариант добавлен в стандарте C++11.

```
key_compare key_comp() const
```

Возвращает функциональный объект, обеспечивающий сравнение ключей.

```
iterator lower_bound(k)
```

Если в множестве или отображении присутствует элемент с ключом k, то возвращается его позиция (в случае мультимножества или мультиотображения возвращается позиция *первого* элемента с ключом k); если такого элемента нет, то возвращается позиция, куда будет вставлен такой элемент.

```
iterator upper_bound(k)
```

Если в множестве или отображении присутствует элемент с ключом k, то возвращается позиция элемента, следующего за ним (в случае мультимножества и мультиотображения возвращается позиция элемента, следующего за *последним* элементом с ключом k); если элемента с ключом k нет, то возвращается позиция, куда будет вставлен такой элемент.

```
value_compare value_comp() const
```

Возвращает функциональный объект, обеспечивающий сравнение элементов контейнера по их ключам. В случае (мульти)множества совпа-

дает с объектом key_compare, в случае (мульти)отображения выполняет сравнение пар pair<const Key, T> по их первому компоненту Key.

### 1.2.7. Вставка и удаление в последовательных контейнерах

**Функция-член insert** реализована во всех последовательных контейнерах в трех вариантах (в стандарте C++11 добавлен еще один вариант). Первый параметр во всех вариантах – итератор pos, определяющий позицию вставки. Новые данные вставляются, начиная с указанной позиции pos; все прежние элементы, начиная с позиции pos и далее, смещаются вправо (по направлению к концу контейнера). Варианты различаются параметрами, определяющими, что именно вставляется: это либо (1) один параметр x типа T (вставляется единственное значение x), либо (2) параметры n и x (вставляются n значений x), либо (3) два итератора чтения InIterFirst и InIterLast (вставляются все элементы из диапазона [InIterFirst, InIterLast)), либо (4, в стандарте C++11) список инициализации init_list. До появления стандарта C++11 только вариант (1) функции insert возвращал значение, этим значением являлся итератор, указывающий на вставленный элемент. В стандарте C++11 варианты (3) и (4) также возвращают значение – итератор, указывающий на первый вставленный элемент, или исходное значение pos, если диапазон или список инциализации являются пустыми. Параметр-итератор pos и возвращаемый итератор всегда прямые (обычные) итераторы; обратные итераторы в качестве pos использовать нельзя.

Имеются дополнительные функции-члены, связанные со вставкой: это push_back(x) – вставка одного элемента в конец контейнера (реализована для всех последовательных контейнеров) и push_front(x) – вставка одного элемента в начало контейнера (реализована для дека и списка). Эти функции не возвращают значения.

**Функция-член erase** реализована во всех последовательных контейнерах в двух вариантах: (1) с параметром-итератором pos, определяющим позицию удаляемого элемента, и с двумя параметрами-итераторами first и last, определяющими диапазон [first, last) удаляемых элементов. В обоих вариантах возвращается итератор, который указывает на элемент, расположенный за удаленным элементом (или удаленным диапазоном).

Имеются дополнительные функции-члены, связанные с удалением: это pop_back() – удаление последнего элемента из контейнера (реализована для всех последовательных контейнеров; поддержка для строк string добавлена в стандарте C++11), pop_front() – удаление первого элемента из контейнера (реализована для дека и списка), clear() – удаление всех элементов из контейнера (реализована для всех контейнеров). Эти функции не возвращают значения.

Альтернативой функциям insert и erase для *списков* list являются три варианта **функции-члена splice**, позволяющие перемещать отдельные эле-

менты или их диапазоны между различными списками или между различными позициями одного списка. Все варианты функции splice начинаются с параметров pos (итератора, определяющего место вставки) и lst (спискаисточника вставляемых данных). Если других параметров нет, то списокисточник lst целиком вставляется в позицию pos списка-приемника; если имеется один дополнительный параметр-итератор pos_lst, то из спискаисточника в список-приемник перемещается единственный элемент, связанный с итератором pos_lst; если имеются два дополнительных параметра first_lst и last_lst, то перемещается диапазон элементов [first_lst, last_lst). Все перемещаемые элементы удаляются из списка-источника.

При выполнении вставки и удаления важно знать, когда в результате выполнения этих действий итераторы и ссылки становятся недействительными (одновременно со ссылками становятся недействительными и указатели).

**Вектор**

*Вставка:*

- если в результате вставки выполняется перераспределение памяти (увеличивается емкость), то становятся недействительными все итераторы и ссылки;
- если перераспределения памяти не производится, то итераторы и ссылки до позиции вставки остаются корректными, а прочие – недействительными.

*Удаление:*

- все итераторы и ссылки до позиции удаления остаются корректными, а прочие – недействительными.

**Дек**

*Вставка:*

- все итераторы делаются недействительными;
- ссылки делаются недействительными при вставке в середину дека и остаются корректными при вставке в начало или конец дека.

*Удаление:*

- все итераторы делаются недействительными;
- ссылки на оставшиеся элементы делаются недействительными при удалении элементов из середины дека и остаются корректными при удалении начальных или конечных элементов дека.

**Список**

*Вставка*:

- все итераторы и ссылки остаются корректными.

*Удаление:*

- все итераторы и ссылки на оставшиеся элементы списка остаются корректными.

### 1.2.8. Контейнеры-адаптеры и контейнеры, добавленные в стандарт C++11

В данном пункте, как и в пунктах 1.2.2–1.2.7, посвященных основным видам контейнеров, при описании конструкторов и функций-членов контейнеров не указывается дополнительный тип Alloc, который обычно устанавливается по умолчанию. Все рассматриваемые в данном пункте контейнеры определены в пространстве имен std.

Начнем с описания *контейнеров-адаптеров*: стека (stack), очереди (queue) и очереди с приоритетом (priority_queue).

Особенностью контейнеров-адаптеров является то, что они основаны на одном из «настоящих» контейнеров, который хранится в качестве *внутреннего контейнера* (underlying container) с именем c – защищенного члена контейнера-адаптера. Для очереди с приоритетом используется еще один защищенный член – функциональный объект comp, используемый для сравнения элементов.

Реализация функций-членов контейнеров-адаптеров сводится к вызову соответствующих функций-членов «внутреннего» контейнера c. При этом набор средств, доступных для контейнеров-адаптеров, существенно сокращен по сравнению со средствами внутреннего контейнера. В частности, с контейнерами-адаптерами не связываются итераторы, что не позволяет их использовать совместно со стандартными алгоритмами.

В таблице 3 приводится описание шаблонов контейнеров-адаптеров.

*Таблица 3*

**Контейнеры-адаптеры**

| **Имя** | **Описание** | **Заголовочный файл** |
|---|---|---|
| stack<T, Containter = deque<T>> | *Стек* с элементами типа T на базе контейнера Container | <stack> |
| queue<T, Containter = deque<T>> | *Очередь* с элементами типа T на базе контейнера Container | <queue> |
| priority_queue<T, Containter = vector<T>, Compare = less<Container::value_type>> | *Очередь с приоритетом* с элементами типа T на базе контейнера Container и с операцией сравнения Compare | <queue> |

Для всех контейнеров-адаптеров можно использовать конструктор без параметров и *конструктор копии* с параметром other того же типа, что и создаваемый адаптер; в стандарте C++11 добавлен *конструктор перемещения*, в котором параметр other является ссылкой на r-значение.

Кроме того, для стека и очереди можно использовать конструктор с явно указанным контейнером cont типа const Container& (в стандарте C++11 добавлен вариант с типом Container&&), содержимое которого копируется (или, соответственно, перемещается) во внутренний контейнер c адаптера.

Для очереди с приоритетом предусмотрены также конструкторы со следующими параметрами (в квадратных скобках, как обычно, указаны необязательные параметры):

```
(comp, [cont])
(first, last, [comp, [cont]])
```

Здесь параметр comp имеет тип const Compare& и определяет операцию сравнения для очереди с приоритетом, параметр cont имеет тип const Container& или Container&& (C++11) и определяет начальное содержимое, которое копируется или, соответственно, перемещается во внутренний контейнер c адаптера. При наличии параметров-итераторов чтения first и last после первоначального заполнения внутреннего контейнера с помощью параметра cont (или, по умолчанию, создания пустого внутреннего контейнера) для него вызывается функция-член c.insert(c.end(), first, last). На завершающем этапе формирования очереди с приоритетом с использованием данных вариантов конструктора для ее внутреннего контейнера вызывается алгоритм make_heap(c.begin(), c.end(), comp).

Для всех контейнеров-адаптеров определена операция =, обеспечивающая копирование (а в стандарте C++11 и перемещение) элементов из одного адаптера в другой того же типа.

Кроме того, для всех контейнеров-адаптеров (как и для всех других контейнеров) определены функции-члены bool empty() const и size_type size() const (выполнение которых сводится к вызову одноименных функций-членов внутренних контейнеров: c.empty() и c.size()), а также функция-член void swap(other), обеспечивающая обмен содержимым между двумя адаптерами одинакового типа (для этого выполняется вызов алгоритма swap(c, other.c), а в случае очереди с приоритетом дополнительно вызывается алгоритм swap(comp, other.comp)).

Для доступа к элементам контейнеров-адаптеров предусмотрены следующие функции-члены:

- *стек*: reference top() – доступ к верхнему элементу стека (возвращает c.back());
- *очередь*: reference front() – доступ к начальному элементу очереди (возвращает c.front()) и reference back() – доступ к конечному элементу очереди (возвращает c.back());
- *очередь с приоритетом*: reference front() – доступ к наибольшему элементу очереди (возвращает c.front()).

Контейнеры-адаптеры реализуют ограниченный набор действий. *Стек* позволяет добавлять новый элемент в вершину, получать значение верхнего элемента и удалять верхний элемент. *Очередь* позволяет добавлять новый элемент в конец, получать значение начального и конечного элемента и удалять начальный элемент. *Очередь с приоритетом* позволяет добавлять новый элемент, получать значение наибольшего элемента и удалять наибольший элемент.

Во всех контейнерах-адаптерах используются одинаковые имена для функций-членов, связанных с добавлением и удалением элементов. Эти функции-члены описаны в таблице 4.

*Таблица 4*

**Вставка и удаление для контейнеров-адаптеров**

| | void push(value) | *C++11*<br>void emplace(arg1, arg2, …) | void pop() |
|---|---|---|---|
| Стек | Элемент со значением value добавляется в вершину стека:<br>c.push_back(value); | Элемент конструируется «на месте» и добавляется в вершину стека:<br>c.emplace_back(arg1, arg2, …); | Из стека удаляется верхний элемент:<br>c.pop_back(); |
| Очередь | Элемент со значением value добавляется в конец очереди:<br>c.push_back(value); | Элемент конструируется «на месте» и добавляется в конец очереди:<br>c.emplace_back(arg1, arg2, …); | Из очереди удаляется начальный элемент:<br>c.pop_front(); |
| Очередь с приоритетом | Элемент со значением value добавляется в очередь с приоритетом:<br>c.push_back(value);<br>push_heap(c.begin(), c.end(), comp); | Элемент конструируется «на месте» и добавляется в очередь с приоритетом:<br>c.emplace_back(arg1, arg2, …);<br>push_heap(c.begin(), c.end(), comp); | Из очереди с приоритетом удаляется наибольший элемент:<br>pop_heap(c.begin(), c.end(), comp);<br>c.pop_back(); |

Теперь кратко опишем те новые виды контейнеров, которые появились в стандарте C++11.

**Контейнер** array является контейнерным аналогом *массива фиксированного размера*. Его шаблон имеет вид: array<T, N>, где T – это тип элементов, а N – число, задающее размер контейнера. Данный контейнер определен в заголовочном файле <array>. Помимо конструктора без параметров

контейнер array можно инициализировать с помощью списка инициализации. Все, кроме одной, функции-члены контейнера array аналогичны по своему назначению одноименным функциям-членам контейнера vector (можно сказать, что array – это контейнер vector без возможности вставки и удаления элементов). Перечислим эти функции-члены:

- функции, описанные в п. 1.2.3: begin и его константный вариант cbegin, empty, end и его константный вариант cend, max_size, rbegin и его константный вариант crbegin, rend и его константный вариант crend, size, swap;
- функции, описанные в п. 1.2.4: operator[], at, back, data, front.

Единственная функция-член, имеющаяся в контейнере array и при этом отсутствующая в контейнере vector, – это функция void fill(value), позволяющая заполнить существующий контейнер array одинаковыми значениями value.

**Контейнер** forward_list является контейнерной реализацией *односвязного списка* (в отличие от контейнера list, реализующего двусвязный список). Данный контейнер требует меньше памяти для хранения своих элементов, но при этом обладает и более ограниченным по сравнению со списком list набором возможностей. Его шаблон имеет вид: forward_list<T>, где T – тип элементов списка. Контейнер определен в заголовочном файле <forward_list>. Контейнер forward_list имеет те же варианты конструктора и операции =, что и контейнер list.

Перечислим функции-члены, которые имеются у обеих реализаций списков – как list, так и forward_list – и выполняются аналогичным образом:

- функции, описанные в п. 1.2.3: begin и его константный вариант cbegin, clear, empty, end и его константный вариант cend, max_size, swap;
- функции, описанные в п. 1.2.4: assign, emplace_front, front, pop_front, push_front, resize;
- функции, описанные в п. 1.2.5: merge, remove, remove_if, reverse, sort, unique.

Обратите внимание на то, что у списка forward_list отсутствуют средства быстрого доступа к его конечному элементу, а также обратные итераторы и функция-член size.

Функции-члены, связанные со вставкой и удалением элементов списка forward_list, отличаются от аналогичных функций списка list тем, что в качестве параметра pos указывается не позиция вставляемого или удаляемого элемента, а позиция, *предшествующая* позиции вставляемого или удаляемого элемента (что обусловлено односвязностью списка forward_list). По этой причине все функции-члены, связанные со вставкой и удалением, снабжены в классе forward_list суффиксом «after»: insert_after, emplace_after, erase_after, splice_after. Назначение этих функций и смысл их параметров те же, что и для аналогичных функций-членов контейнера list без суффикса

_after: insert, emplace, erase (см. п. 1.2.4) и splice (см. п. 1.3.5). Исключение составляет параметр pos, указывающий, как было отмечено выше, позицию, *предшествующую* позиции вставляемого или удаляемого элемента, а также параметр first для функции erase_after(first, last) и параметры pos_lst и first_lst для функций splice_after(pos, lst, pos_lst) и splice_after(pos, lst, first_lst, last_lst):

- функция erase_after(first, last) удаляет элементы в диапазоне (first, last) (элемент first *не удаляется*, в чем состоит отличие от реализации функции-члена erase(first, last) в других последовательных контейнерах – см. п. 1.2.4);
- функция splice_after(pos, lst, pos_lst) перемещает из списка lst (типа forward_list) в текущий список элемент, *следующий* за элементом в позиции pos_lst (и помещает его в позицию, *следующую* за позицией pos), а функция splice_after(pos, lst, first_lst, last_lst) перемещает в позицию, следующую за позицией pos, элементы списка lst, расположенные в диапазоне (first_lst, last_lst) (элемент first_lst в диапазон *не включается*). Это отличается от поведения функций-членов splice контейнера list с аналогичным набором параметров (см. п. 1.2.5).

Контейнер forward_list содержит также функции-члены before_begin и cbefore_begin, которые возвращают обычный и константный итератор, указывающий на позицию, *предшествующую* первому элементу контейнера. Эти итераторы позволяют использовать функции insert_after, emplace_after, splice_after и erase_after для вставки новых данных *в начало* контейнера forward_list и удаления *начальной части* его элементов.

**Неупорядоченные ассоциативные контейнеры** unordered_set, unordered_multiset, unordered_map и unordered_multimap обеспечивают ту же функциональность, что и стандартные упорядоченные ассоциативные контейнеры set, multiset, map и multimap (см. п. 1.2.1, 1.2.2, 1.2.6), однако для поиска по ключу в них используются *хеш-функции* (hash functions), генерирующие *хеш-коды* ключей, а также функции, сравнивающие ключи *на равенство*. Все элементы, ключи которых возвращают одинаковый хеш-код, помещаются в одну *ячейку* (bucket) неупорядоченного ассоциативного контейнера (число ячеек для контейнера может либо устанавливаться по умолчанию, либо указываться в его шаблоне). При поиске элемента по ключу вначале вычисляется хеш-код ключа, определяющий ячейку, которая может содержать элемент с данным ключом. Если эта ячейка содержит несколько элементов, то элемент с нужным ключом ищется в ней обычным перебором. Высокая скорость в подобном механизме поиска обеспечивается за счет того, что для каждого ключа можно быстро определить его хеш-код, позволяющий сразу обратиться к нужной ячейке, которая, как правило, содержит небольшое число элементов.

Таким образом, поиск по ключу в неупорядоченных ассоциативных контейнерах выполняется с помощью двух видов операций сравнения на

равенство, определяемых в шаблоне контейнера: это операция сравнения хеш-кодов, вычисленных хеш-функцией, и операция сравнения на равенство самих ключей (выполняемая при переборе элементов в пределах одной ячейки). Это отличает неупорядоченные контейнеры от упорядоченных, в которых ключи сравниваются тем или иным вариантом операции <.

Неупорядоченные ассоциативные контейнеры содержат практически все средства, имеющиеся у упорядоченных контейнеров (см. п. 1.2.2 и 1.2.6); отсутствуют лишь функции для работы с обратными итераторами, а также функции lower_bound и upper_bound (хотя функция-член equal_range имеется). Кроме того, вместо функций key_comp и value_comp у неупорядоченных контейнеров предусмотрены функции-члены hash_function (возвращает хеш-функцию), и key_eq (возвращает функцию для сравнения ключей на равенство). Разумеется, при переборе элементов неупорядоченных ассоциативных контейнеров не гарантируется, что они будут располагаться по возрастанию ключей.

Неупорядоченные контейнеры включают также следующие функции-члены для работы с ячейками:

- max_bucket_count() возвращает максимальное количество ячеек, которое можно выделить для данного контейнера;
- bucket_count() возвращает количество ячеек, выделенных для данного контейнера;
- bucket(key) возвращает *индекс* ячейки, содержащей элементы с ключом key;
- bucket_size(n) возвращает размер ячейки с указанным индексом n;
- begin(n), end(n) и cbegin(n), cend(n) возвращают итераторы для перебора всех элементов, входящих в ячейку с индексом n.

Наконец, еще одна группа функций-членов предназначена для оптимизации размещения данных в неупорядоченных контейнерах:

- load_factor() возвращает среднее число элементов к ячейке (число типа float, равное size()/bucket_count());
- max_load_factor позволяет определить (функция без параметров, возвращающая результат типа float) и изменить (void-функция с параметром типа float) максимальное среднее число элементов в ячейке; контейнер автоматически увеличивает количество ячеек, если значение load_factor() превысит указанное максимальное значение;
- rehash(count) позволяет явно изменить количество ячеек; в результате новое значение bucket_count() будет больше или равно count, а также больше size()/max_load_factor();
- reserve(n) настраивает контейнер таким образом, чтобы его размер можно было увеличивать до n элементов без автоматического увеличения количества ячеек.

### 1.2.9. Дополнение: обратные итераторы

Получить обратный итератор r можно из обычного (прямого) итератора p явным приведением типа, например:

```
r = (vector<int>::reverse_iterator)p;
```

Имеется функция-член rbegin(), которая возвращает приведенный к типу обратного итератора итератор end(), и функция-член rend(), возвращающая приведенный к типу обратного итератора итератор begin().

Операции инкремента и декремента прямого и обратного оператора взаимно обратны: r++ перемещает итератор в том же направлении, что и p--, а r-- – в том же направлении, что и p++.

Для операции разыменования * выполняется следующее базовое соотношение: если r может быть получен из p, то *r равно *(p – 1).

Функция-член base() обратного итератора возвращает прямой итератор, который можно было бы использовать для получения данного обратного итератора явным приведением типа: если r может быть получен из p, то r.base() == p. Или, иначе говоря, ((reverse_iterator)p).base == p.

***Примеры***

В следующем примере рассматривается последовательный контейнер cont с исходными элементами 1, 2, 3, 4, 5. Итераторы p2, p3, p4, p5 связаны с элементами 2, 3, 4, 5. Обратные итераторы r2, r3, r4, r5 определены следующим образом (rev – псевдоним типа обратного итератора для cont):

```
r2=(rev)p2;
r3=(rev)p3;
r4=(rev)p4;
r5=(rev)p5;
```

Значения разыменованных итераторов для исходного контейнера:

| | *p2 | *p3 | *p4 | *p5 | *r2 | *r3 | *r4 | *r5 |
|---|---|---|---|---|---|---|---|---|
| (вектор) | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 |
| (дек) | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 |
| (список) | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 |

После выполнения оператора

```
x = cont.erase(p3); // или x = cont.erase(r3.base());
```

значения разыменованных итераторов будут следующими («*» означает, что попытка разыменования приводит к непредсказуемым результатам):

| | *x | *p2 | *p3 | *p4 | *p5 | *r2 | *r3 | *r4 | *r5 |
|---|---|---|---|---|---|---|---|---|---|
| (вектор) | 4 | 2 | * | * | * | 1 | * | * | * |
| (дек) | 4 | * | * | * | * | * | * | * | * |
| (список) | 4 | 2 | * | 4 | 5 | 1 | * | 2 | 4 |

Теперь повторно инициализируем итераторы p4 и r4

```
p4 = x;
r4 = (rev)p4;
```

и выполним операторы

```
p3 = cont.insert(p4, 3);
    // или p3 = cont.insert(r4.base(), 3);
r3 = (rev)p3;
```

В результате значения разыменованных итераторов изменятся следующим образом:

```
          *p2 *p3 *p4 *p5 *r2 *r3 *r4 *r5
(вектор) 2   3   *   *   1   2   *   *
(дек)    *   3   *   *   *   2   *   *
(список) 2   3   4   5   1   2   3   4
```

Анализ полученных результатов полностью соответствует ранее описанным правилам использования функций insert и erase, а также правилам, связанным с корректностью итераторов. Имеется лишь одно не вполне очевидное обстоятельство, касающееся того, что происходит с обратными итераторами списка, значения которых были связаны с удаляемым элементом (r4) и с элементом, предшествующим удаляемому (r3).

Итератор r3 становится недействительным, что является вполне естественным, так как уничтожается тот элемент, на который указывал итератор r3.base().

В случае итератора r4 ситуация интереснее. Несмотря на то, что значение, которое он возвращал, пропало, сам этот итератор сохранился, поскольку сохранился связанный с ним прямой итератор r4.base() (и, хотя это не отражено в приведенных данных, после выполнения операции удаления значение r4.base() не изменилось). Однако, поскольку после удаления элемента 3 элементом, предшествующим «базовому» элементу, связанному с итератором r4.base(), оказался элемент 2, именно его значение возвращается при разыменовании обратного итератора r4. Таким образом, перед удалением элемента 3 значение итератора r4 было равно 3, а после его удаления значение становится равным *предшествующему* значению (т. е. 2). При вставке элемента 3 перед элементом 4 базовый элемент для обратного итератора r4 не изменился (он по-прежнему равен p4), но, поскольку теперь перед ним находится элемент 3, именно это значение (3) возвращается разыменованным итератором r4.

## 1.3. Алгоритмы

### 1.3.1. Общее описание

Данный раздел содержит описание всех алгоритмов стандартной библиотеки шаблонов, включенных в стандарт C++11. Новые алгоритмы, появившиеся в этом стандарте, помечены текстом *C++11*. Алгоритм random_shuffle, который объявлен в стандарте C++11 устаревшим, помечен текстом *deprecated*. Алгоритмы, определенные в заголовочном файле <algorithm>, описаны в п. 1.3.3, алгоритмы, определенные в заголовочном файле <numeric>, – в п. 1.3.4. В каждом пункте алгоритмы располагаются в алфавитном порядке своих имен.

Все алгоритмы определены в пространстве имен std. В таблице 5 алгоритмы сгруппированы в соответствии со способом их применения.

*Таблица 5*

**Алгоритмы STL по категориям**

| Категория | Алгоритмы |
|---|---|
| Немодифицирующие операции для последовательностей | all_of (C++11), any_of (C++11), none_of (C++11), for_each, count, count_if, mismatch, equal, find, find_if, find_if_not (C++11), find_end, find_first_of, adjacent_find, search, search_n |
| Базовые модифицирующие операции для последовательностей | copy, copy_backward, copy_if (C++11), copy_n (C++11), move (C++11), move_backward (C++11), fill, fill_n, transform, generate, generate_n, remove, remove_if, remove_copy, remove_copy_if, replace, replace_if, replace_copy, replace_copy_if, swap, swap_ranges, iter_swap, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, shuffle (C++11), unique, unique_copy |
| Разбиение последовательностей | is_partitioned (C++11), partition, partition_copy (C++11), stable_partition, partition_point (C++11) |
| Сортировка последовательностей | is_sorted (C++11), is_sorted_until (C++11), sort, partial_sort, stable_sort, nth_element |
| Слияние (для отсортированных последовательностей) | merge, inplace_merge |
| Бинарный поиск (для отсортированных последовательностей) | lower_bound, upper_bound, binary_search, equal_range |

*Таблица 5 (продолжение)*

| Категория | Алгоритмы |
|---|---|
| Теоретико-множественные операции (для отсортированных последовательностей) | includes, set_difference, set_intersection, set_symmetric_difference, set_union |
| Работа с *кучей* (heap) | is_heap (C++11), is_heap_until (C++11), make_heap, push_heap, pop_heap, sort_heap |
| Нахождение минимумов и максимумов, сравнение последовательностей | max, max_element, min, min_element, minmax (C++11), minmax_element (C++11), lexicographical_compare |
| Перестановки | is_permutation (C++11), next_permutation, prev_permutation |
| Численные алгоритмы (заголовочный файл <numeric>) | iota (C++11), accumulate, inner_product, adjacent_difference, partial_sum |

### 1.3.2. Соглашения об именовании параметров

В качестве типов для параметров-итераторов first, last, result, result_last (возможно, дополненных номерами 1 или 2) указываются:

- InIter – итератор чтения (input);
- OutIter – итератор записи (output);
- FwdIter – однонаправленный итератор (forward);
- BidiIter – двунаправленный итератор (bidirectional);
- RandIter – итератор произвольного доступа (random).

В качестве типа значения для входных последовательностей указывается T; если выходная последовательность может иметь тип элементов, отличный от T, то для него используется имя TRes. Итераторы из диапазонов [first, last), [first1, last1), [first2, last2) обозначаются с помощью переменных p, p1, p2 соответственно.

Для типов функциональных объектов в описаниях алгоритмов используются следующие имена:

- UnaryOp – унарная операция (функциональный объект с операцией (), имеющей один параметр типа T; при этом тип возвращаемого значения может отличаться от типа T);
- BinaryOp – бинарная операция (функциональный объект с операцией (), имеющей два параметра, как правило, одинакового типа T; тип возвращаемого значения может отличаться от типа T); если параметры бинарной операции могут иметь различные типы, то об этом явно говорится в описании соответствующего алгоритма;
- Predicate – унарный предикат (унарная операция, возвращающая логическое значение);
- BinaryPredicate – бинарный предикат (бинарная операция с параметрами типа T, возвращающая логическое значение);

- Compare – бинарный предикат, предназначенный для сравнения элементов (аналог операции <);
- Generator – генератор последовательности (функциональный объект с операцией (), не имеющей параметров и возвращающей значение типа TRes);
- RandomGenerator – генератор случайных целых чисел, равномерно распределенных в диапазоне [0, n).

Во всех алгоритмах, связанных с копированием или перемещением данных, предполагается, что в результирующей последовательности зарезервировано достаточно места для размещения всех получаемых элементов.

Всюду при указании сложности алгоритма под $N$ понимается разность итераторов distance(first, last) (если $N$ имеет индекс, то подразумевается, что итераторы имеют такой же номер, например $N_1$ = distance(first1, last1)). Если сложность алгоритма является постоянной, т. е. не зависит от размера обрабатываемой последовательности, то она не указывается.

### 1.3.3. Алгоритмы общего назначения

Алгоритмы, описываемые в данном пункте, определены в заголовочном файле <algorithm>.

```
FwdIter adjacent_find(FwdIter first, FwdIter last[,
    BinaryPredicate pred])
```

Находит первую пару соседних элементов из диапазона [first, last), которые равны (или, при наличии предиката pred(*p, *(p + 1)), для которых данный предикат возвращает true). Возвращает итератор, связанный с первым элементом найденной пары, или last, если пара не найдена.

Сложность линейная (не более $N + 1$ вызовов pred).

*C++11*

```
bool all_of(InIter first, InIter last, Predicate pred)
```

Возвращает true, если все элементы диапазона [first, last) удовлетворяют предикату pred. В случае пустого диапазона также возвращается true.

Сложность линейная (не более $N$ вызовов pred).

*C++11*

```
bool any_of(InIter first, InIter last, Predicate pred)
```

Возвращает true, если хотя бы один элемент диапазона [first, last) удовлетворяет предикату pred. В случае пустого диапазона возвращается false.

Сложность линейная (не более $N$ вызовов pred).

```
bool binary_search(FwdIter first, FwdIter last, const T& value[,
    Compare comp])
```

Использует двоичный поиск для проверки того, содержится ли в диапазоне [first, last) значение value (если значение найдено, то возвращает true,

иначе false). Содержимое диапазона должно быть предварительно отсортировано в соответствии с порядком, задаваемым предикатом comp(*p1, *p2) или (по умолчанию) операцией <.

Сложность логарифмическая (не более log $N + 2$ сравнений).

```
OutIter copy(InIter first, InIter last, OutIter result)
```

Копирует элементы из [first, last) в диапазон, начинающийся с result, и возвращает позицию за последним скопированным элементом в полученном диапазоне. Итератор result не может находиться в исходном диапазоне [first, last), но другие части выходного диапазона могут накладываться на исходный диапазон. Таким образом, данный алгоритм можно применять для «копирования влево», т. е. копирования в ситуации, когда левая граница выходного диапазона находится слева от исходного диапазона.

Сложность линейная ($N$ присваиваний).

```
BidiIter2 copy_backward(BidiIter1 first, BidiIter1 last,
    BidiIter2 result_last)
```

Выполняет те же действия, что и copy, но перебирает исходные данные в обратном порядке: от элемента, предшествующего last, до first. Итератор result_last должен указывать на элемент, следующий за концом выходной последовательности; возвращаемое значение – это итератор, указывающий на первый элемент выходной последовательности. Итератор result_last не может находиться в диапазоне (first, last] (обратите внимание на границы этого диапазона), но другие части выходного диапазона могут накладываться на исходный диапазон. Таким образом, данный алгоритм можно применять для «копирования вправо», т. е. копирования в ситуации, когда правая граница выходного диапазона находится справа от исходного диапазона.

Сложность линейная ($N$ присваиваний).

*C++11*

```
OutIter copy_if(InIter first, InIter last, OutIter result,
    Predicate pred)
```

Копирует в диапазон, начинающийся с result, все элементы диапазона [first, last), для которых pred возвращает true. Возвращает позицию за последним скопированным элементом в полученном диапазоне. Относительный порядок элементов в полученном диапазоне сохраняется. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная ($N$ сравнений).

```
OutIter copy_n(InIter first, Size n, OutIter result );
```

Копирует в диапазон, начинающийся с result, n элементов диапазона, начинающегося с first.

Сложность линейная ($n$ присваиваний).

```
difference_type count(InIter first, InIter last, const T& value)
```

Возвращает количество элементов в диапазоне [first, last), которые равны значению value.

Сложность линейная (*N* сравнений).

```
difference_type count_if(InIter first, InIter last, Predicate
    pred)
```

Возвращает количество элементов в диапазоне [first, last), для которых выражение pred(*p) равно true.

Сложность линейная (*N* вызовов pred).

```
bool equal(InIter1 first1, InIter1 last1, InIter2 first2[,
    BinaryPredicate pred])
```

Возвращает true, если два диапазона содержат одни и те же элементы в одинаковом порядке. Первый диапазон – [first1, last1), второй начинается с first2 и имеет такую же длину; диапазоны могут перекрываться. Для сравнения используется предикат pred(*p1, *p2) или (по умолчанию) операция ==.

Сложность линейная (не более $N_1$ сравнений).

```
pair<FwdIter, FwdIter> equal_range(FwdIter first, FwdIter last,
    const T& value[, Compare comp])
```

Проверяет, имеется ли в диапазоне [first, last) значение value, и возвращает пару итераторов, которые указывают на начало диапазона, содержащего value, и на элемент за концом этого диапазона (если значение не найдено, то оба итератора указывают на позицию в диапазоне, в которую можно вставить value, не нарушая порядка сортировки). Содержимое диапазона должно быть предварительно отсортировано в соответствии с порядком, задаваемым предикатом comp(*p1, *p2) или (по умолчанию) операцией <.

Сложность логарифмическая (не более $2*\log N + 1$ сравнений).

```
void fill(FwdIter first, FwdIter last, const T& value)
```

Заполняет выходной диапазон [first, last) значениями value.

Сложность линейная (*N* присваиваний).

```
void fill_n(OutIter first, Size n, const T& value)
```

Заполняет выходной диапазон из n элементов, начиная с first, значениями value.

Сложность линейная (*n* присваиваний).

```
InIter find(InIter first, InIter last, const T& value)
```

Возвращает итератор, указывающий на первое вхождение элемента value в диапазоне [first, last), или last, если элемент value отсутствует. Для сравнения элементов используется операция ==.

Сложность линейная (не более *N* сравнений).

```
FwdIter1 find_end(FwdIter1 first1, FwdIter1 last1, FwdIter2
    first2, FwdIter2 last2[, BinaryPredicate pred])
```

Находит последнюю (самую правую) подпоследовательность [first2, last2) в диапазоне [first1, last1). Возвращает итератор, который указывает на начало найденной подпоследовательности, или last1, если подпоследовательность не найдена. Для сравнения элементов используется предикат pred(*p1, *p2) или (по умолчанию) операция ==.

Сложность линейная (не более $N_1$*$N_2$ сравнений).

```
FwdIter1 find_first_of(FwdIter1 first1, FwdIter1 last1, FwdIter2
    first2, FwdIter2 last2[, BinaryPredicate pred])
```

Находит первое вхождение любого элемента подпоследовательности [first2, last2) в диапазон [first1, last1); возвращает итератор, который указывает на найденный элемент, или last1, если элемент не найден. Для сравнения элементов используется предикат pred(*p1, *p2) или (по умолчанию) операция ==.

Сложность линейная (не более $N_1$*$N_2$ сравнений).

```
InIter find_if(InIter first, InIter last, Predicate pred)
```

Возвращает итератор, указывающий для диапазона [first, last) на первое вхождение элемента, для которого выражение pred(*p) возвращает true; если требуемые элементы отсутствуют, то возвращает last.

Сложность линейная (не более $N$ вызовов pred).

*C++11*

```
InIter find_if_not(InIter first, InIter last, Predicate pred)
```

Возвращает итератор, указывающий для диапазона [first, last) на первое вхождение элемента, для которого выражение pred(*p) возвращает false; если требуемые элементы отсутствуют, то возвращает last.

Сложность линейная (не более $N$ вызовов pred).

```
Func for_each(InIter first, InIter last, Func f)
```

Вызывает функциональный объект f (как f(*p)) для всех элементов из диапазона [first, last) и возвращает этот же функциональный объект.

Сложность линейная ($N$ вызовов f).

```
void generate(FwdIter first, FwdIter last, Generator gen)
```

Заполняет диапазон [first, last), последовательно присваивая элементам диапазона результат вызова функционального объекта gen (как gen()).

Сложность линейная ($N$ вызовов gen).

```
void generate_n(OutIter first, Size n, Generator gen)
```

Заполняет последовательность, начинающуюся с позиции first, n элементами, полученными в результате вызова функционального объекта gen (как gen()).

Сложность линейная ($n$ вызовов gen).

```
bool includes(InIter1 first1, InIter1 last1, InIter2 first2,
    InIter2, last2[, Compare comp])
```

Возвращает true, если все элементы предварительно отсортированной последовательности [first2, last2) содержатся в предварительно отсортированной последовательности [first1, last1), и false в противном случае (фактически ищется вхождение подпоследовательности [first2, last2) в диапазон [first1, last1)). Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более $2*(N_1 + N_2) - 1$ сравнений).

```
void inplace_merge(BidiIter first, BidiIter middle, BidiIter
    last[, Compare comp])
```

Выполняет слияние двух предварительно отсортированных частей [first, middle) и [middle, last) последовательности на месте, в результате чего создается единый отсортированный диапазон [first, last). Слияние является устойчивым; кроме того, в полученном диапазоне равные элементы из первого диапазона будут располагаться перед равными им элементами из второго диапазона. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная ($N + 1$ сравнений) или (при нехватке памяти) $N*\log N$ сравнений.

*C++11*

```
bool is_heap(RandIter first, RandIter last[, Compare comp])
```

Возвращает true, если диапазон [first, last) представляет собой *кучу* (см. алгоритм make_heap). Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более $N$ сравнений).

*C++11*

```
RandIter is_heap_until(RandIter first, RandIter last[, Compare
    comp])
```

Определяет наибольший диапазон в пределах исходного диапазона [first, last), который начинается с first и представляет собой *кучу* (см. алгоритм make_heap). Возвращает позицию за концом найденного диапазона. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более $N$ сравнений).

*C++11*

```
bool is_partitioned(InIter first, InIter last, Predicate pred)
```

Возвращает true, если в диапазоне [first, last) все элементы, удовлетворяющие предикату pred, расположены перед всеми элементами, которые

предикату не удовлетворяют. В случае пустого диапазона также возвращается true.

Сложность линейная (не более *N* вызовов pred).

*C++11*

```
bool is_permutation(FwdIter1 first1, FwdIter1 last1, FwdIter2
    first2[, BinaryPredicate pred])
```

Возвращает true, если диапазон [first1, last1) представляет собой *перестановку* элементов диапазона, который начинается с first2 и имеет такую же длину. Для сравнения используется предикат pred(*p1, *p2) или (по умолчанию) операция ==.

Сложность: не более *N***N* вызовов pred (ровно *N* вызовов в случае, если элементы первого диапазона совпадают с соответствующими элементами второго диапазона).

*C++11*

```
bool is_sorted(FwdIter first, FwdIter last[, Compare comp])
```

Возвращает true, если диапазон [first, last) представляет собой отсортированную последовательность. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более *N* сравнений).

*C++11*

```
FwdIter is_sorted_until(FwdIter first, FwdIter last[, Compare
    comp])
```

Определяет наибольший диапазон в пределах исходного диапазона [first, last), который начинается с first и представляет собой отсортированную последовательность. Возвращает позицию за концом найденного диапазона. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более *N* сравнений).

```
void iter_swap(FwdIter1 a, FwdIter2 b)
```

Меняет местами значения, на которые указывают итераторы a и b.

```
bool lexicographical_compare(InIter1 first1, InIter1 last1,
    InIter2 first2, InIter2 last2[, Compare comp])
```

Возвращает true, если последовательность [first1, last1) «меньше» (в лексикографическом смысле), чем последовательность [first2, last2), и false в противном случае (в частности, если последовательности равны, то возвращается false, а если первая последовательность является собственным префиксом второй, то возвращается true). Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более $\min\{N_1, N_2\}$ сравнений).

```
FwdIter lower_bound(FwdIter first, FwdIter last, const T& value[,
    Compare comp])
```

Проверяет, содержится ли в диапазоне [first, last) значение value, и возвращает итератор, который указывает на первое вхождение value (если значение не найдено, то итератор указывает на позицию в диапазоне, в которую можно вставить value, не нарушая порядка сортировки). Содержимое диапазона должно быть предварительно отсортировано в соответствии с порядком, задаваемым предикатом comp(*p1, *p2) или (по умолчанию) операцией <.

Сложность логарифмическая (не более $\log N + 1$ сравнений).

```
void make_heap(RandIter first, RandIter last[, Compare comp])
```

Переупорядочивает элементы диапазона [first, last), получая из него *кучу* (т. е. очередь с приоритетом, для которой первый элемент всегда больше остальных, а добавление нового элемента или удаление первого элемента может быть произведено за логарифмическое время, и результат тоже будет кучей). Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более $3*N$ сравнений).

```
const T& max(const T& a, const T& b[, Compare comp])
```

Возвращает большее из значений a и b (при их равенстве возвращается a). Для сравнения значений используется предикат comp(a, b) или (по умолчанию) операция <.

```
FwdIter max_element(FwdIter first, FwdIter last[, Compare comp])
```

Возвращает позицию первого наибольшего элемента в диапазоне [first, last). В случае пустого диапазона возвращает last. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная ($\max\{N - 1, 0\}$ сравнений).

```
OutIter merge(InIter1 first1, InIter1 last1, InIter2 first2,
    InIter2 last2, OutIter result[, Compare comp])
```

Выполняет слияние двух предварительно отсортированных диапазонов [first1, last1) и [first2, last2) и копирование результата в последовательность, начиная с result (в выходной последовательности должно быть достаточно места для полученного набора данных). Возвращает выходной итератор, указывающий на позицию за концом добавленного набора данных. Выходной диапазон не должен накладываться ни на один из исходных диапазонов. Слияние является устойчивым; кроме того, в полученном диапазоне равные элементы из первого диапазона будут располагаться перед равными им элементами из второго диапазона. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более $N_1 + N_2 - 1$ сравнений).

```
const T& min(const T& a, const T& b[, Compare comp])
```

Возвращает меньшее из значений a и b (при их равенстве возвращается a). Для сравнения значений используется предикат comp(a, b) или (по умолчанию) операция <.

```
FwdIter min_element(FwdIter first, FwdIter last[, Compare comp])
```

Возвращает позицию первого наименьшего элемента в диапазоне [first, last). В случае пустого диапазона возвращает last. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная ($\max\{N-1, 0\}$ сравнений).

*C++11*

```
pair<const T&, const T&> minmax(const T& a, const T& b[,
    Compare comp])
```

Возвращает пару, содержащую меньшее и большее из значений a и b (при их равенстве возвращается пара (a, b)). Для сравнения значений используется предикат comp(a, b) или (по умолчанию) операция <.

*C++11*

```
pair<FwdIter, FwdIter> minmax_element(FwdIter first,
    FwdIter last[, Compare comp])
```

Возвращает пару, содержащую позиции первого наименьшего и последнего наибольшего элемента в диапазоне [first, last). В случае пустого диапазона возвращает пару (last, last). Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более $\max\{\mathrm{floor}(3/2*(N-1)), 0\}$ сравнений).

```
pair<InIter1, InIter2> mismatch(InIter1 first1, InIter1 last1,
    InIter2 first2[, BinaryPredicate pred])
```

Выполняет попарное сравнение элементов из диапазона [first1, last1) и диапазона, начинающегося с first2 и имеющего длину, не меньшую, чем длина первого диапазона. Возвращает первую пару различных элементов (или, если все элементы первого диапазона совпадают с соответствующими элементами второго диапазона, пару из итератора last1 и соответствующего итератора для второго диапазона). Для сравнения элементов используется предикат pred(*p1, *p2) или (по умолчанию) операция ==.

Сложность линейная (не более $N_1$ сравнений).

*C++11*

```
OutIter move(InIter first, InIter last, OutIter result)
```

Перемещает элементы из [first, last) в диапазон, начинающийся с result, и возвращает позицию за последним перемещенным элементом в полученном диапазоне. Итератор result не может находиться в исходном диапазоне [first, last). После выполнения этого алгоритма диапазон [first, last) будет по-

прежнему содержать элементы того же типа, но их значения могут отличаться от исходных.

Сложность линейная (*N* присваиваний).

*C++11*

```
BidiIter2 move_backward(BidiIter1 first, BidiIter1 last,
    BidiIter2 result_last)
```

Выполняет те же действия, что и move, но перебирает исходные данные в обратном порядке: от элемента, предшествующего last, до first. Итератор result_last должен указывать на элемент, следующий за концом формируемой выходной последовательности; возвращаемое значение – это итератор, указывающий на первый элемент выходной последовательности. Итератор result_last не может находиться в диапазоне (first, last] (обратите внимание на границы этого диапазона).

Сложность линейная (*N* присваиваний).

```
bool next_permutation(BidiIter first, BidiIter last[,
    Compare comp])
```

Переупорядочивает содержимое диапазона [first, last), создавая следующую перестановку из набора лексикографически упорядоченных перестановок элементов данного диапазона. Возвращает true, если перестановка была создана успешно, или false, если исходный диапазон представлял собой последнюю (в лексикографическом порядке) перестановку; в этом последнем случае генерируется первая в лексикографическом порядке перестановка (в которой все элементы расположены в порядке возрастания). Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более *N*/2 перемещений).

*C++11*

```
bool none_of(InIter first, InIter last, Predicate pred)
```

Возвращает true, если ни один из элементов диапазона [first, last) не удовлетворяет предикату pred. В случае пустого диапазона также возвращается true.

Сложность линейная (не более *N* вызовов pred).

```
void nth_element(RandIter first, RandIter nth, RandIter last[,
    Compare comp])
```

Переупорядочивает диапазон [first, last) таким образом, чтобы в позиции nth размещался элемент, который стоял бы на этом месте в случае, если бы весь диапазон был отсортирован. Кроме того, в результате выполнения данного алгоритма все элементы в диапазоне [first, nth) не будут превосходить элементы из диапазона [nth, last). Алгоритм не является устойчивым: если имеется несколько элементов, которые при сортировке могли бы ока-

заться на позиции nth, то нельзя сказать, какой из них будет перемещен на эту позицию. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность в среднем линейная (около *N* сравнений).

```
void partial_sort(RandIter first, RandIter middle, RandIter
    last[, Compare comp])
```

Частично сортирует элементы диапазона [first, last), размещая отсортированные элементы в диапазоне [first, middle). Оставшиеся элементы никак не упорядочиваются. Алгоритм не является устойчивым. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность: примерно $N*\log(middle - first)$ сравнений.

```
RandIter partial_sort_copy(InIter first1, InIter last1, RandIter
    first2, RandIter last2[, Compare comp])
```

Частично сортирует элементы из диапазона [first1, last1) и копирует отсортированную часть в диапазон [first2, last2). Размер сортируемой части определяется размером второго диапазона: если он меньше первого, то сортируется часть первого диапазона, если он больше или равен размеру первого диапазона, то сортируется весь первый диапазон.

Возвращается итератор второго диапазона, указывающий на позицию за концом отсортированного набора данных, добавленного из первого диапазона. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность: примерно $N_1*\log(\min\{N_1, N_2\})$ сравнений.

```
BidiIter partition(BidiIter first, BidiIter last, Predicate pred)
```

Меняет местами элементы диапазона [first, last) так, чтобы все элементы, удовлетворяющие предикату pred, были расположены перед теми, которые ему не удовлетворяют. Относительный порядок следования элементов не сохраняется. Алгоритм возвращает итератор, указывающий на первый элемент, для которого pred возвращает false (или итератор last, если таких элементов нет).

Сложность линейная (*N* вызовов pred и не более *N*/2 перемещений элементов).

*C++11*

```
pair<OutIter1, OutIter2> partition_copy(InIter first, InIter
    last, OutIter1 result1, OutIter2 result2, Predicate pred)
```

Копирует элементы из диапазона [first, last) в два выходных диапазона: элементы, удовлетворяющие предикату pred, копируются в первый диапазон (начинающийся с result1), а остальные элементы – во второй диапазон (начинающийся с result2). Исходный диапазон не должен перекрываться

с выходными диапазонами. Возвращает пару итераторов, определяющих позиции за концами первого и второго полученного диапазона (в указанном порядке).

Сложность линейная (*N* вызовов pred).

*C++11*

```
FwdIter partition_point(FwdIter first, FwdIter last, Predicate
    pred)
```

В предположении, что все элементы, удовлетворяющие предикату pred, расположены в начале диапазона [first, last), находит и возвращает позицию первого элемента, не удовлетворяющего предикату pred. Если все элементы диапазона удовлетворяют предикату, то возвращается last.

Сложность логарифмическая (не более log *N* вызовов pred).

```
void pop_heap(RandIter first, RandIter last[, Compare comp])
```

При условии, что диапазон [first, last) является *кучей* (см. алгоритм make_heap), перемещает первый (наибольший) элемент этой кучи в конец этого диапазона (т. е. в элемент *(last – 1)) и гарантирует, что элементы, оставшиеся в диапазоне [first, last – 1), образуют кучу. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность логарифмическая (не более 2*log *N* сравнений).

```
bool prev_permutation(BidiIter first, BidiIter last[,
    Compare comp])
```

Переупорядочивает содержимое диапазона [first, last), создавая предыдущую перестановку из набора лексикографически упорядоченных перестановок элементов данного диапазона. Возвращает true, если перестановка была создана успешно, или false, если исходный диапазон представлял собой первую (в лексикографическом порядке) перестановку; в этом последнем случае генерируется последняя в лексикографическом порядке перестановка (где все элементы расположены в порядке убывания). Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более *N*/2 перемещений).

```
void push_heap(RandIter first, RandIter last[, Compare comp])
```

При условии, что диапазон [first, last – 1) является *кучей* (см. алгоритм make_heap), добавляет в эту кучу элемент, расположенный в позиции last – 1, формируя тем самым кучу в диапазоне [first, last). Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность логарифмическая (не более log *N* сравнений).

*deprecated*

```
void random_shuffle(RandIter first, RandIter last[,
    RandomGenerator rand])
```

Случайным образом изменяет порядок элементов из диапазона [first, last). Для генерации случайных чисел по умолчанию используется встроенный генератор с равномерным распределением; может также использоваться явно заданный генератор rand(n), возвращающий целое случайное число в диапазоне [0, n). В стандарте C++11 алгоритм random_shuffle объявлен устаревшим; вместо него рекомендуется использовать алгоритм shuffle.

Сложность линейная ($N + 1$ перемещений элементов).

```
FwdIter remove(FwdIter first, FwdIter last, const T& value)
```

«Удаляет» из диапазона [first, last) элементы, равные value (для сравнения элементов используется операция ==). Возвращает итератор middle, определяющий конец диапазона [first, middle), содержащего преобразованный набор без удаленных элементов. Относительный порядок оставшихся в диапазоне [first, middle) элементов сохраняется. В результате выполнения алгоритма значения элементов в диапазоне [middle, last) становятся неопределенными.

Сложность линейная ($N$ сравнений).

```
OutIter remove_copy(InIter first, InIter last, OutIter result,
    const T& value)
```

Копирует в диапазон, начинающийся с result, все элементы диапазона [first, last), кроме элементов, равных value (для сравнения элементов используется операция ==). Возвращает позицию за последним скопированным элементом в полученном диапазоне. Относительный порядок элементов в полученном диапазоне сохраняется. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная ($N$ сравнений).

```
OutIter remove_copy_if(InIter first, InIter last, OutIter result,
    const T& value, Predicate pred)
```

Копирует в диапазон, начинающийся с result, все элементы диапазона [first, last), кроме элементов, для которых pred возвращает true. Возвращает позицию за последним скопированным элементом в полученном диапазоне. Относительный порядок элементов в полученном диапазоне сохраняется. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная ($N$ сравнений).

```
FwdIter remove_if(FwdIter first, FwdIter last, Predicate pred)
```

«Удаляет» из диапазона [first, last) элементы, для которых pred возвращает true. Возвращает итератор middle, определяющий конец диапазона [first, middle), содержащего преобразованный набор без удаленных элементов.

Относительный порядок оставшихся в диапазоне [first, middle) элементов сохраняется. В результате выполнения алгоритма значения элементов в диапазоне [middle, last) становятся неопределенными.

Сложность линейная (*N* сравнений).

```
void replace(FwdIter first, FwdIter last, const T& old_value,
    const T& new_value)
```

Заменяет в диапазоне [first, last) все вхождения значения old_value на new_value.

Сложность линейная (*N* сравнений).

```
OutIter replace_copy(InIter first, InIter last, OutIter result,
    const T& old_value, const T& new_value)
```

Копирует элементы диапазона [first, last) в диапазон, начинающийся с result, заменяя при этом все элементы, равные old_value, на new_value. Возвращает позицию за последним скопированным элементом в полученном диапазоне. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная (*N* сравнений).

```
OutIter replace_copy_if(InIter first, InIter last, OutIter
    result, Predicate pred, const T& new_value)
```

Копирует элементы диапазона [first, last) в диапазон, начинающийся с result, заменяя при этом все элементы, для которых pred возвращает true, на new_value. Возвращает позицию за последним скопированным элементом в полученном диапазоне. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная (*N* сравнений).

```
void replace_if(FwdIter first, FwdIter last, Predicate pred,
    const T& new_value)
```

Заменяет в диапазоне [first, last) все элементы, для которых pred возвращает true, на new_value.

Сложность линейная (*N* сравнений).

```
void reverse(BidiIter first, BidiIter last)
```

Переставляет элементы диапазона [first, last) в обратном порядке.

Сложность линейная (*N*/2 обменов).

```
OutIter reverse_copy(BidiIter first, BidiIter last,
    OutIter result)
```

Копирует в обратном порядке элементы диапазона [first, last) в диапазон, начинающийся с result. Возвращает позицию за последним скопированным элементом в полученном диапазоне. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная (*N* присваиваний).

```
void rotate(FwdIter first, FwdIter middle, FwdIter last)
```

Циклически сдвигает элементы диапазона [first, last) так, что элементы из диапазона [middle, last) оказываются в начале новой последовательности (элементы из диапазона [first, middle) передвигаются в конец последовательности).

Сложность линейная (не более *N* перестановок).

```
OutIter rotate_copy(FwdIter first, FwdIter middle, FwdIter last,
    OutIter result)
```

Копирует в диапазон, начинающийся с result, вначале элементы из диапазона [middle, last), а затем – элементы из диапазона [first, middle). Возвращает позицию за последним скопированным элементом в полученном диапазоне. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная (*N* присваиваний).

```
FwdIter1 search(FwdIter1 first1, FwdIter1, last1, FwdIter2
    first2, FwdIter2 last2[, BinaryPredicate pred])
```

Находит первую (самую левую) подпоследовательность [first2, last2) в диапазоне [first1, last1). Возвращает итератор, который указывает на начало найденной подпоследовательности, или last1, если подпоследовательность не найдена. Для сравнения элементов используется предикат pred(*p1, *p2) или (по умолчанию) операция ==.

Сложность линейная (не более $N_1*N_2$ сравнений).

```
FwdIter search_n(FwdIter first, FwdIter last, Size count, const
    T& value[, BinaryPredicate pred])
```

Находит первую (самую левую) подпоследовательность из count соседних одинаковых значений value в диапазоне [first, last). Возвращает итератор, который указывает на начало найденной подпоследовательности, или last, если подпоследовательность не найдена. Для сравнения элементов используется предикат pred(*p1, *p2) или (по умолчанию) операция == (фактически второй параметр в предикате всегда полагается равным value).

Сложность линейная (не более $N*count$ сравнений).

```
OutIter set_difference(InIter1 first1, InIter1 last1, InIter2
    first2, InIter2 last2, OutIter result[, Compare comp])
```

Копирует элементы предварительно отсортированного диапазона [first1, last1) в диапазон, начинающийся с result; при этом копируются только те элементы, которых нет в диапазоне [first2, last2) (этот диапазон также должен быть предварительно отсортирован). Возвращает позицию за последним скопированным элементом в полученном диапазоне. Таким образом, алгоритм реализует аналог теоретико-множественной операции *разности*, при котором в исходных наборах допускаются одинаковые, с точки зрения операции сравнения, элементы. Например, при обработке наборов

(10, 10, 10, 20, 20, 40) и (10, 10, 30, 40) будет получен набор (10, 20, 20). Результирующий диапазон не должен перекрываться с любым из исходных. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более $2*(N_1 + N_2) - 1$ сравнений).

```
OutIter set_intersection(InIter1 first1, InIter1 last1, InIter2
    first2, InIter2 last2, OutIter result[, Compare comp])
```

Копирует элементы предварительно отсортированного диапазона [first1, last1) в диапазон, начинающийся с result; при этом копируются только те элементы, которые присутствуют в диапазоне [first2, last2) (этот диапазон также должен быть предварительно отсортирован). Возвращает позицию за последним скопированным элементом в полученном диапазоне. Таким образом, алгоритм реализует аналог теоретико-множественной операции *пересечения*, при котором в исходных наборах допускаются одинаковые, с точки зрения операции сравнения, элементы. Например, при обработке наборов (10, 10, 10, 20, 20, 40) и (10, 10, 30, 40) будет получен набор (10, 10, 40). Результирующий диапазон не должен перекрываться с любым из исходных. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более $2*(N_1 + N_2) - 1$ сравнений).

```
OutIter set_symmetric_difference(InIter1 first1, InIter1 last1,
    InIter2 first2, InIter2 last2, OutIter result[,
    Compare comp])
```

Выполняет слияние предварительно отсортированных диапазонов [first1, last1) и [first2, last2) и копирует полученные отсортированные элементы в диапазон, начинающийся с result; при этом копируются только те элементы, которые присутствуют только в одном из исходных диапазонов. Возвращает позицию за последним скопированным элементом в полученном диапазоне. Таким образом, алгоритм реализует аналог теоретико-множественной операции *симметричной разности*, при котором в исходных наборах допускаются одинаковые, с точки зрения операции сравнения, элементы. Например, при обработке наборов (10, 10, 10, 20, 20, 40) и (10, 10, 30, 40) будет получен набор (10, 20, 20, 30). Используемый алгоритм слияния обладает теми же свойствами, что и алгоритм merge; в частности, он является устойчивым. Результирующий диапазон не должен перекрываться с любым из исходных. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более $2*(N_1 + N_2) - 1$ сравнений).

```
OutIter set_union(InIter1 first1, InIter1 last1, InIter2 first2,
    InIter2 last2, OutIter result[, Compare comp])
```

Выполняет слияние предварительно отсортированных диапазонов [first1, last1) и [first2, last2) и копирует полученные отсортированные элементы в диапазон, начинающийся с result; при этом равные элементы, присутствующие в каждом из исходных диапазонов, копируются только один раз (копируется элемент из первого диапазона). Возвращает позицию за последним скопированным элементом в полученном диапазоне. Таким образом, алгоритм реализует аналог теоретико-множественной операции *объединения*, при котором в исходных наборах допускаются одинаковые, с точки зрения операции сравнения, элементы. Например, при обработке наборов (10, 10, 10, 20, 20, 40) и (10, 10, 30, 40) будет получен набор (10, 10, 10, 20, 20, 30, 40). Используемый алгоритм слияния обладает теми же свойствами, что и алгоритм merge; в частности, он является устойчивым. Результирующий диапазон не должен перекрываться с любым из исходных. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность линейная (не более $2*(N_1 + N_2) - 1$ сравнений).

*C++11*

```
void shuffle(RandIter first, RandIter last, RandomGenerator rand)
```

Случайным образом изменяет порядок элементов из диапазона [first, last). Требует явного указания генератора случайных чисел с равномерным распределением (набор стандартных генераторов содержится в заголовочном файле <random>). Алгоритм добавлен в стандарт C++11 с целью замены алгоритма random_shuffle, объявленного в этом стандарте устаревшим.

Сложность линейная ($N + 1$ перемещений элементов).

```
void sort(RandIter first, RandIter last[, Compare comp])
```

Сортирует элементы в диапазоне [first, last). Сортировка не является устойчивой, т. е. равные элементы могут не сохранять свой исходный порядок. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность: в среднем $N*\log N$ сравнений.

```
void sort_heap(RandIter first, RandIter last[, Compare comp])
```

Сортирует элементы в диапазоне [first, last) в предположении, что исходный диапазон образует *кучу* (см. алгоритм make_heap). Сортировка не является устойчивой, т. е. равные элементы могут не сохранять свой исходный порядок. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность: не более $N*\log N$ сравнений.

```
BidiIter stable_partition(BidiIter first, BidiIter last,
    Predicate pred)
```

Меняет местами элементы диапазона [first, last) так, чтобы все элементы, удовлетворяющие предикату pred, были расположены перед теми, которые ему не удовлетворяют. Относительный порядок следования элементов сохраняется. Алгоритм возвращает итератор, указывающий на первый элемент, для которого pred возвращает false (или итератор last, если таких элементов нет).

Сложность линейная ($N$ вызовов pred и не более $N$*log $N$ перестановок даже в случае ограниченного объема памяти).

```
void stable_sort(RandIter first, RandIter last[, Compare comp])
```

Сортирует элементы в диапазоне [first, last). Сортировка является устойчивой, т. е. равные элементы сохраняют свой исходный порядок. Для сравнения элементов используется предикат comp(*p1, *p2) или (по умолчанию) операция <.

Сложность: в среднем $N$*log $N$ сравнений, если достаточно памяти (при ограниченном объеме памяти – не более $N$*log $N$*log $N$ сравнений).

```
void swap(T&a, T&b)
```

Меняет местами значения a и b.

```
FwdIter2 swap_ranges(FwdIter1 first1, FwdIter1 last1,
    FwdIter2 first2)
```

Меняет местами элементы диапазона [first1, last1) с элементами диапазона, начинающегося с first2 и имеющего ту же длину, что и первый диапазон. Возвращает позицию за последним элементом второго диапазона. Диапазоны не должны перекрываться.

Сложность линейная ($N_1$ перестановок).

```
OutIter transform(InIter first, InIter last, OutIter result,
    UnaryOp unop)
OutIter transform(InIter1 first1, InIter1 last1, InIter2 first2,
    OutIter result, BinaryOp binop)
```

Присваивает новые значения всем элементам диапазона, начинающегося с result. В первой версии алгоритма используются значения unop(*p), где p – итератор из диапазона [first, last); во второй версии алгоритма используются значения binop(*p1, *p2), где p1 – итератор из диапазона [first1, last1), а p2 – итератор из диапазона, начинающегося с first2 и имеющего ту же длину, что и первый диапазон. Возвращает позицию за последним элементом полученного диапазона. Результирующий диапазон может совпадать с любым из входных диапазонов. Тип TRes возвращаемого значения операций unop и binop может отличаться от типа T их параметров.

Сложность линейная ($N$ вызовов unop или $N_1$ вызовов binop).

```
FwdIter unique(FwdIter first, FwdIter last[,
    BinaryPredicate pred])
```

«Удаляет» из диапазона [first, last) повторные вхождения соседних равных элементов (удаляется вся последовательность соседних равных элементов, кроме ее начального элемента). Возвращает итератор middle, определяющий конец диапазона [first, middle), содержащего преобразованный набор без соседних равных элементов. Относительный порядок оставшихся в диапазоне [first, middle) элементов сохраняется. В результате выполнения алгоритма значения элементов в диапазоне [middle, last) становятся неопределенными. Если исходный диапазон [first, last) предварительно отсортирован, то в полученном диапазоне [first, middle) содержатся уникальные значения (также отсортированные). Для сравнения элементов используется предикат pred(*p1, *p2) или (по умолчанию) операция ==.

Сложность линейная ($\max\{0, N-1\}$ сравнений).

```
OutIter unique_copy(InIter first, InIter last, OutIter result[,
    BinaryPredicate pred])
```

Копирует элементы диапазона [first, last) в диапазон, начинающийся с result, удаляя повторные вхождения соседних равных элементов (удаляется вся последовательность соседних равных элементов, кроме ее начального элемента). Возвращает позицию за последним элементом полученного диапазона. Для сравнения элементов используется предикат pred(*p1, *p2) или (по умолчанию) операция ==.

Сложность линейная ($\max\{0, N-1\}$ сравнений).

```
FwdIter upper_bound(FwdIter first, FwdIter last, const T& value[,
    Compare comp])
```

Проверяет, содержится ли в диапазоне [first, last) значение value, и возвращает итератор, который указывает на элемент, расположенный после последнего элемента со значением value (если значение не найдено, то итератор указывает на позицию в диапазоне, в которую можно вставить value, не нарушая порядка сортировки). Содержимое диапазона должно быть предварительно отсортировано в соответствии с порядком, задаваемым предикатом comp(*p1, *p2) или (по умолчанию) операцией <.

Сложность логарифмическая (не более $\log N + 1$ сравнений).

### 1.3.4. Численные алгоритмы

Алгоритмы, описываемые в данном пункте, определены в заголовочном файле <numeric>.

```
TRes accumulate(InIter first, InIter last, TRes init[,
    BinaryOp binop])
```

Обрабатывает элементы из диапазона [first, last), добавляя их к переменной tmp типа TRes, инициализированной значением init, и возвращает

полученный результат. При добавлении используется бинарная операция binop (в виде tmp = binop(tmp, *p)) или (по умолчанию) операция +. Первый параметр операции binop и ее возвращаемое значение должны иметь тип TRes, а второй параметр – тип T элементов исходной последовательности. В случае использования операции + возвращаемое значение имеет тип T. Функциональный объект binop не должен иметь побочных эффектов (в стандарте C++11 это ограничение снято).

Сложность линейная (*N* вызовов binop).

```
OutIter adjacent_difference(InIter first, InIter last, OutIter
    result[, BinaryOp binop])
```

Обрабатывает пары соседних элементов из диапазона [first, last) с помощью операции binop (в виде binop(*(p+1), *p)) или (по умолчанию) бинарной операции «–» (*(p+1) – *p) и записывает полученные значения в диапазон вывода, начиная с result (первым записывается значение *first, затем результат binop(*(first+1),*first) и т. д.). Возвращает позицию за последним элементом полученного диапазона. Тип TRes возвращаемого значения для операции binop может отличаться от типа T элементов исходной последовательности. Если типы T и TRes совпадают, то итератор result может совпадать с first. Функциональный объект binop не должен иметь побочных эффектов (в стандарте C++11 это ограничение снято).

Сложность линейная ($N - 1$ вызовов binop).

```
TRes inner_product(InIter1 first1, InIter1 last1, InIter2 first2,
    TRes init[, BinaryOp1 binop1, BinaryOp2 binop2])
```

Вычисляет сумму попарных произведений («внутреннее произведение») элементов двух диапазонов: [first1, last1) и диапазона такой же длины, начинающегося с first2 (переменная суммирования tmp инициализируется значением init). Вместо бинарных операций «+» и «*» могут использоваться операции binop1 и binop2 соответственно: tmp = binop1(tmp, binop2(*p1, *p2)). Параметры операции binop2 должны иметь тип T элементов исходной последовательности, а тип возвращаемого значения должен совпадать с типом второго параметра операции binop1. Первый параметр операции binop1 и ее возвращаемое значение должны иметь тип TRes. В случае использования операций «+» и «*» возвращаемое значение алгоритма имеет тип T. Функциональные объекты binop1 и binop2 не должны иметь побочных эффектов (в стандарте C++11 это ограничение снято).

Сложность линейная ($N_1$ вызовов binop1 и binop2).

*C++11*

```
void iota(FwdIter first, FwdIter last, T value)
```

Заполняет диапазон [first, last) последовательно увеличивающимися значениями типа T, причем первое значение равно value, а все последующие полагаются равными результату очередного вызова операции инкремента

++value (название алгоритма заимствовано из языка программирования APL, в котором имеется одноименная операция).

Сложность линейная (*N* – 1 операций инкремента и *N* присваиваний).

```
OutIter partial_sum(InIter first, InIter last, OutIter result[,
    BinaryOp binop])
```

Записывает частичные суммы (*first, *first + *(first+1), (*first + *(first+1)) + *(first+2), …) в диапазон, начинающийся с result. Вместо операции по умолчанию «+» может использоваться бинарная операция binop. Возвращает позицию за последним элементом полученного диапазона. Итератор result может совпадать с first. Функциональный объект binop не должен иметь побочных эффектов (в стандарте C++11 это ограничение снято).

Сложность линейная (*N* – 1 вызовов binop).

### 1.3.5. Дополнение: итераторы вставки

С помощью обычных итераторов модифицирующие алгоритмы могут лишь изменять содержимое существующих элементов. *Итераторы вставки* позволяют вставлять новые элементы в контейнеры при выполнении модифицирующих алгоритмов. Итераторы вставки могут использоваться в качестве любых параметров-итераторов, предназначенных для записи данных. Имеются три вида итераторов вставки:

- back_insert_iterator (для вставки в конец контейнера)

Доступен для всех последовательных контейнеров. Может создаваться с помощью функции back_inserter(c), где c – контейнер. Каждое обращение к подобному итератору для записи в контейнер c элемента e приводит к вызову функции-члена c.push_back(e).

- front_insert_iterator (для вставки в начало контейнера)

Доступен для тех контейнеров, для которых реализована функция-член push_front(), т. е. для дека и списка. Может создаваться с помощью функции front_inserter(c), где c – контейнер. Каждое обращение к подобному итератору для записи в контейнер c элемента e приводит к вызову функции-члена c.push_front(e). В результате последовательность элементов, добавляемых с помощью данного итератора, будет размещена в начале контейнера *в обратном порядке*.

- insert_iterator (для вставки в любую позицию контейнера)

Доступен для всех контейнеров. Может создаваться с помощью функции inserter(c, pos), где c – контейнер, а pos – итератор данного контейнера, определяющий место вставки. Каждое обращение к подобному итератору для записи в контейнер c элемента e приводит к выполнению операторов pos = c.insert(pos, e); ++pos; таким образом, следующая вставка будет выполняться за вставленным ранее элементом. В частности, вставка с помощью итератора inserter(c, c.begin()) отличается от вставки с помощью итератора front_inserter(c) тем, что в первом случае вставляемые элементы будут рас-

полагаться в контейнере в исходном порядке. Применение итераторов inserter(c, c.end()) и back_inserter(c) приводит к одинаковому результату.

С осторожностью следует использовать итераторы вставки при добавлении данных в тот же контейнер, из которого они извлекаются, поскольку операция вставки может приводить к тому, что некоторые (или даже все) итераторы, используемые для чтения, окажутся недействительными. Напомним, в частности, что любое действие по вставке элементов в дек делает недействительными все его итераторы. Безопасным контейнером в этом отношении является список, так как вставка в него новых данных оставляет корректными все связанные с ним итераторы. Промежуточное положение занимает вектор, который сохраняет корректными все итераторы до позиции вставки. Однако в случае перераспределения памяти (увеличения емкости) делаются недействительными все итераторы вектора.

Впрочем, проблемы могут возникнуть даже при работе со списками. Рассмотрим задачу дублирования всех элементов списка путем их записи в его конец. Казалось бы, достаточно использовать следующий оператор:

```
copy(v.begin(), v.end(), back_inserter(v));
```

При использовании компилятора gcc этот оператор решает поставленную задачу, однако в Visual Studio он приводит к зависанию программы.

Если требуется продублировать все элементы списка, записав их в конец в обратном порядке, то вариант

```
copy(v.rbegin(), v.rend(), back_inserter(v));
```

решает задачу в gcc, а в Visual Studio создает список, в котором последний элемент исходного списка появляется не 2, а 3 раза. Например, список 1, 2, 3 преобразуется в 1, 2, 3, 3, 3, 2, 1.

В то же время при дублировании элементов списка в его начало проблем не возникает; в частности, первая из рассмотренных задач успешно решается в Visual Studio оператором

```
copy(v.rbegin(), v.rend(), front_inserter(v));
```

а для решения второй можно использовать два действия:

```
copy(v.begin(), v.end(), front_inserter(v));
v.reverse();
```

Таким образом, можно сформулировать следующие рекомендации по безопасному использованию итераторов вставки:

- если алгоритм используется для преобразования и вставки данных из одного контейнера в другой, то такое действие всегда является безопасным;
- если требуется преобразовать и вставить данные из одной части контейнера в другую его часть, то следует принимать во внимание вид контейнера и расположение позиции вставки и исходных дан-

ных. Если позиция вставки и диапазон исходных данных «отграничены» друг от друга, то в случае списка действие всегда является безопасным, а в случае вектора оно будет безопасным, если, в дополнение к предыдущему условию, диапазон исходных данных расположен до позиции вставки и, кроме того, емкость вектора позволит завершить процедуру вставки без перераспределения памяти. В случае дека указанное действие никогда не является безопасным;

- если действие не является безопасным, то желательно выполнять его с использованием вспомогательного контейнера, инициализировав его требуемым диапазоном данных из исходного контейнера и затем использовав вспомогательный контейнер в алгоритме в качестве источника данных.

## *1.4. Стандартные функциональные объекты*

### 1.4.1. Общее описание

Стандартные функциональные объекты определены в пространстве имен std (за исключением объектов-*заменителей* _1, _2, _3, …, определенных в пространстве имен std::placeholders). Для возможности работы с ними надо подключить заголовочный файл <functional>.

В настоящем разделе описывается большинство стандартных функциональных объектов, включая функциональные адаптеры (см. п. 1.4.2) и объекты для арифметических и логических операций (см. п. 1.4.3).

Раздел стандартной библиотеки шаблонов, связанный со стандартными функциональными объектами, в стандарте C++11 был подвергнут наиболее существенным изменениям. В то время как для контейнеров и алгоритмов нововведения стандарта C++11 связаны в основном с добавлением новых возможностей (и небольшими модификациями прежних), практически все прежние функциональные адаптеры в стандарте C++11 были объявлены устаревшими и заменены новыми вариантами. В п. 1.4.2, посвященном функциональным адаптерам, описаны как прежние их варианты (они помечены словом *deprecated*), так и новые (помечены текстом *C++11*).

Интересно отметить, что адаптеры-*инверторы* not1 и not2, будучи объявлены устаревшими, не получили в стандарте C++11 должной замены; их новый вариант not_fn предполагается включить лишь в стандарт C++17. По указанной причине, хотя в данной книге не рассматриваются нововведения языка C++, добавленные после стандарта C++11, для адаптера not_fn было сделано исключение.

### 1.4.2. Функциональные адаптеры

*C++11*

`_1, _2, _3, …`

Объекты-*заменители* (placeholders), определенные в пространстве имен std::placeholders и используемые в функциональном адаптере-связывателе bind. При указании в программе директивы using namespace std::placeholders; она должна располагаться после подключения заголовочного файла <functional>.

*deprecated*

**binary_function**`<TArg1, TArg2, TRes>`

Шаблон класса, который должен быть базовым для классов, реализующих функциональный объект в виде бинарной операции () с параметрами типа TArg1 и TArg2 и возвращаемым значением типа TRes.

*C++11*

**bind**`(op, arg1, arg2, arg3, …, argN)`

Функциональный адаптер-*связыватель* (binder), принимающий функциональный объект op(x1, x2, x3, ..., xN) с $N$ параметрами и возвращающий функциональный объект с $M$ параметрами (y1, y2, ..., yM), где $M \le N$. Среди параметров arg1, …, argN должно быть $M$ *заменителей* вида _1, _2, _3, …, определяющих положение и порядок параметров нового функционального объекта (_1 соответствует параметру y1, _2 – параметру y2 и т. д.). Параметры из списка arg1, …, argN, не являющиеся заменителями, задают фиксированные значения для соответствующих параметров исходного функционального объекта.

Например, в случае вызова bind(op, a1, _3, _2, a4, _1) на основе объекта op(x1, x2, x3, x4, x5) с пятью параметрами создается объект с тремя параметрами (y1, y2, y3), который определяется через исходный объект следующим образом: op(a1, y3, y2, a4, y1) (первый и четвертый параметры исходного функционального объекта получают фиксированные значения a1 и a4 соответственно).

В отличие от старых связывателей bind1st и bind2nd связыватель bind может принимать в качестве своего первого параметра непосредственно функции-члены классов (для которых не требуется вызывать дополнительный функциональный адаптер mem_fn).

*deprecated*

**bind1st**`(op, arg1)`

Функциональный адаптер-*связыватель* (binder), принимающий функциональный объект op(x1, x2) с двумя параметрами и возвращающий функциональный объект op(arg1, x2) с одним параметром x2 (первый параметр исходного функционального объекта получает фиксированное значение arg1).

*deprecated*

**bind2nd**(op, arg2)

Функциональный адаптер-*связыватель* (binder), принимающий функциональный объект op(x1, x2) с двумя параметрами и возвращающий функциональный объект op(x1, arg2) с одним параметром x1 (второй параметр исходного функционального объекта получает фиксированное значение arg2).

*C++11*

**function**<TRes(TArg1, TArg2, …)>

Шаблон, являющийся полиморфной оболочкой, обобщающей понятие указателя на функцию. Может использоваться в качестве базового для классов, реализующих функциональный объект в виде операции () с параметрами типа (TArg1, TArg2, ...) и возвращаемым значением типа TRes. Число параметров может быть произвольным.

*C++11*

**mem_fn**(fmember)

Функциональный адаптер, принимающий указатель на функцию-член fmember некоторого класса T и возвращающий его обертку в виде функционального объекта. Полученный функциональный объект может применяться к *объекту* класса T (или производного от него класса) или к *указателю* на такой объект.

*deprecated*

**mem_fun**(fmember)

Функциональный адаптер, принимающий указатель на функцию-член fmember некоторого класса T и возвращающий его обертку в виде функционального объекта. Полученный функциональный объект должен применяться к *указателю* на объект класса T (или производного от него класса).

*deprecated*

**mem_fun_ref**(fmember)

Функциональный адаптер, принимающий указатель на функцию-член fmember некоторого класса T и возвращающий его обертку в виде функционального объекта. Полученный функциональный объект должен применяться к *объекту* класса T (или производного от него класса).

*C++17*

**not_fn**(pred)

Функциональный адаптер-*инвертор* (negator), принимающий функциональный объект-предикат pred(x1, x2, ...) с произвольным числом параметров и возвращающий функциональный объект, являющийся логическим отрицанием объекта pred: !pred(x1, x2, ...).

*deprecated*

`not1(pred)`

Функциональный адаптер-*инвертор* (negator), принимающий функциональный объект-предикат pred(x) с одним параметром и возвращающий функциональный объект, являющийся логическим отрицанием объекта pred: !pred(x).

Инвертор not1 не может применяться к функциональному объекту, возвращаемому связывателем bind: комбинация not1(bind(pred, ...)) приводит к ошибке компиляции. В то же время допустимыми являются комбинации not1(bind1st(pred, arg1)) и not1(bind2nd(pred, arg2)).

*deprecated*

`not2(pred)`

Функциональный адаптер-*инвертор* (negator), принимающий функциональный объект-предикат pred(x1, x2) с двумя параметрами и возвращающий функциональный объект, являющийся логическим отрицанием объекта pred: !pred(x1, x2).

Инвертор not2 не может применяться к функциональному объекту, возвращаемому связывателем bind: комбинация not2(bind(pred, ...)) приводит к ошибке компиляции. В то же время допустимыми являются комбинации bind(not2(pred), ...), bind1st(not2(pred), arg1) и bind2nd(not2(pred), arg2).

*deprecated*

`unary_function<TArg, TRes>`

Шаблон класса, который должен быть базовым для классов, реализующих функциональный объект в виде унарной операции () с параметром типа TArg и возвращаемым значением типа TRes.

### 1.4.3. Функциональные объекты для арифметических и логических операций

`divides<T>()`

Функциональный объект-обертка для бинарной операции / с параметрами типа const T& и возвращаемым значением типа T.

`equal_to<T>()`

Функциональный объект-обертка для бинарной операции == с параметрами типа const T& и возвращаемым значением типа bool.

`greater<T>()`

Функциональный объект-обертка для бинарной операции > с параметрами типа const T& и возвращаемым значением типа bool.

`greater_equal<T>()`

Функциональный объект-обертка для бинарной операции >= с параметрами типа const T& и возвращаемым значением типа bool.

`less<T>()`

Функциональный объект-обертка для бинарной операции < с параметрами типа const T& и возвращаемым значением типа bool.

`less_equal<T>()`

Функциональный объект-обертка для бинарной операции <= с параметрами типа const T& и возвращаемым значением типа bool.

`logical_and<T>()`

Функциональный объект-обертка для бинарной операции && с параметрами типа const T& и возвращаемым значением типа bool.

`logical_not<T>()`

Функциональный объект-обертка для унарной операции ! с параметром типа const T& и возвращаемым значением типа bool.

`logical_or<T>()`

Функциональный объект-обертка для бинарной операции || с параметрами типа const T& и возвращаемым значением типа bool.

`minus<T>()`

Функциональный объект-обертка для бинарной операции – с параметрами типа const T& и возвращаемым значением типа T.

`modulus<T>()`

Функциональный объект-обертка для бинарной операции % с параметрами типа const T& и возвращаемым значением типа T.

`multiplies<T>()`

Функциональный объект-обертка для бинарной операции * с параметрами типа const T& и возвращаемым значением типа T.

`negate<T>()`

Функциональный объект-обертка для унарной операции – с параметром типа const T& и возвращаемым значением типа T.

`not_equal_to<T>()`

Функциональный объект-обертка для бинарной операции != с параметрами типа const T& и возвращаемым значением типа bool.

`plus<T>()`

Функциональный объект-обертка для бинарной операции + с параметрами типа const T& и возвращаемым значением типа T.

# Раздел 2. Использование библиотеки STL

## *2.1. Знакомство с итераторами и алгоритмами: STL1Iter17*

### 2.1.1. Установка задачника, создание проекта-заготовки и знакомство с заданием

Задания группы STL1Iter знакомят с такими базовыми средствами стандартной библиотеки, как итераторы и алгоритмы.

*Итераторы* представляют собой объекты, обеспечивающие перемещение по последовательности и доступ к ее элементам (можно сказать, что итератор – это абстракция понятия указателя). Хотя обычно итераторы используются при работе с контейнерами; наиболее простые виды итераторов – итераторы для чтения и итераторы для записи – могут применяться в программе для доступа к потокам чтения/записи, поэтому начать знакомство с итераторами можно без предварительного изучения контейнеров STL (которым посвящены группы STL2Seq, STL4Str и STL5Assoc).

Впрочем, сами по себе итераторы не позволяют воспользоваться всеми возможностями стандартной библиотеки: для этого их необходимо применять совместно с *алгоритмами* – функциями из библиотеки STL, которые обеспечивают выполнение различных стандартных операций над последовательностями данных и используют итераторы для доступа к обрабатываемым последовательностям. Благодаря применению итераторов алгоритмы не «привязываются» к конкретному виду последовательности (например, контейнеру определенного типа), а могут обрабатывать любые последовательности, поддерживающие работу с итераторами (даже если они не хранятся в памяти, а извлекаются из некоторого потока ввода или, наоборот, после генерации элементов сразу передаются в поток вывода).

В заданиях группы STL1Iter для обработки исходных наборов данных требуется использовать простейшие алгоритмы STL, которые применяются непосредственно к потокам ввода-вывода через связанные с ними итераторы. В качестве потоков ввода-вывода используются стандартные файловые потоки, а также специальный поток pt, определенный в задачнике Programming Taskbook.

Большинство важных особенностей заданий группы STL1Iter можно проиллюстрировать на примере выполнения задания STL1Iter17.

STL1Iter17. Дана строка *name* и набор символов. Записать в текстовый файл с именем *name* удвоенные кодовые значения всех символов из исходного набора в том же порядке, добавляя после каждого числа один пробел. Использовать итераторы ptin_iterator, ostream_iterator и алгоритм transform.

Перед тем как приступить к выполнению заданий из задачника Programming Taskbook for STL, необходимо установить на компьютер универсальный задачник Programming Taskbook и задачник Programming Taskbook for STL, являющийся дополнением универсального задачника. Дистрибутивные пакеты для установки задачников можно скачать с сайта электронного задачника http://ptaskbook.com/ (страница «Главная | Услуги | Скачивание дистрибутивов»). Задачник PT for STL может использоваться как с полным вариантом универсального задачника, так и с его мини-вариантом, не требующим последующей регистрации. После установки универсального задачника будет автоматически запущена программа его настройки PT4Setup, в которой будут указаны все среды программирования (из числа поддерживаемых задачником), обнаруженные на компьютере. Необходимо убедиться, что в число этих сред входит хотя бы одна среда для языка C++. В этой же программе можно указать рабочий каталог, в котором будут выполняться задания (по умолчанию рабочий каталог размещается на диске C и имеет имя PT4Work), а также задать имя студента, которое будет храниться в файле с результатами выполнения заданий. Заметим, что программа PT4Setup, как и все другие компоненты задачника, доступна из его меню «Пуск | Программы | Programming Taskbook 4».

После установки универсального задачника Programming Taskbook следует установить его дополнение Programming Taskbook for STL.

Выполнение задания с применением задачника Programming Taskbook начинается с создания *проекта-заготовки* для выбранного задания. Для создания заготовки предназначен программный модуль PT4Load, входящий в состав задачника. Вызвать этот модуль можно с помощью ярлыка Load.lnk, который автоматически создается в рабочем каталоге студента.

После запуска модуля PT4Load на экране появится его окно, в котором будут перечислены все доступные группы заданий (рис. 1).

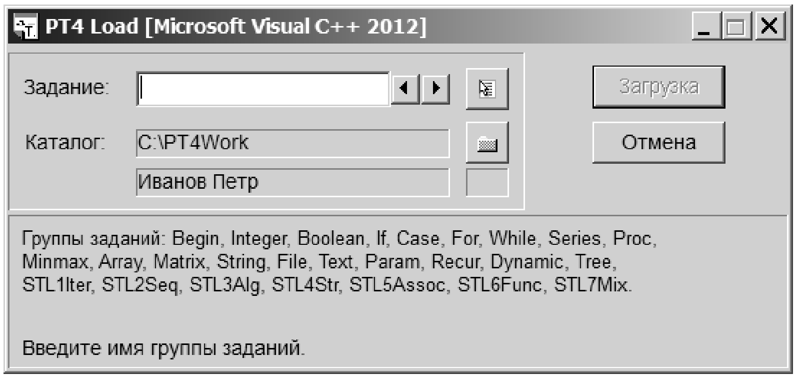


**Рис. 1**. Окно модуля PT4Load

Заметим, что с помощью модуля PT4Load можно не только создать заготовку для задания, но и определить любое количество новых рабочих каталогов; для этого достаточно нажать на кнопку и выбрать существу-

ющий или создать новый каталог, после чего указать имя студента, связанное с этим рабочим каталогом).

В заголовке окна указывается имя текущей среды программирования и номер ее версии. Приведенный рисунок соответствует настройке, при которой текущей средой является среда Microsoft Visual Studio 2012 для языка C++.

При выполнении заданий, посвященных стандартной библиотеке шаблонов C++, можно использовать любые среды программирования для языка C++, которые поддерживаются электронным задачником. В версии 4.17 задачника это среды Microsoft Visual Studio 2008, 2010, 2012, 2013, 2015, 2017 и Code::Blocks 13. Если указана среда, отличная от требуемой, следует вызвать контекстное меню модуля PT4Load (щелкнув в его окне правой кнопкой мыши или нажав на кнопку ) и выбрать нужную среду из появившегося списка. В дальнейшем будем предполагать, что текущей средой является одна из версий среды Visual Studio.

В списке групп заданий должна содержаться группа STL1Iter. Ее отсутствие может объясняться двумя причинами: либо в качестве текущего языка выбран язык, отличный от C++, либо на компьютере не установлено дополнение Programming Taskbook for STL.

После ввода имени задания (в нашем случае STL1Iter17) кнопка «Загрузка» в окне PT4Load станет доступной, и, нажав ее (или клавишу [Enter]), мы создадим заготовку для указанного задания, которая немедленно загрузится в среду Visual Studio. Созданный проект включает несколько файлов, однако для решения задачи нам потребуется только файл с именем STL1Iter17.cpp. Именно этот файл будет загружен в редактор среды Visual Studio.

Приведем текст файла STL1Iter17.cpp (этот текст будет одинаковым для всех сред программирования, которые можно использовать для решения задачи):

```
#include "pt4.h"
using namespace std;

void Solve()
{
    Task("STL1Iter17");
}
```

Файл начинается с директивы подключения заголовочного файла pt4.h, в котором содержатся описания дополнительных функций, обеспечивающих взаимодействие учебной программы с задачником (в число этих функций входит, в частности, функция Task, предназначенная дли инициализации требуемого задания).

Затем указывается директива using namespace std, благодаря которой в последующем тексте программы можно не уточнять стандартное пространство имен std для связанных с ним типов (заметим, что все типы библиотеки STL, за очень небольшим исключением, определены именно в пространстве имен std).

Наконец, описывается функция Solve, в которой требуется запрограммировать решение задачи; в этой функции уже содержится вызов функции Task, инициализирующей задание.

Следует обратить внимание на то, что в тексте файла STL1Iter17.cpp отсутствует описание стартовой функции (которая обычно имеет имя main или, для среды Visual Studio, WinMain). Данная функция, разумеется, входит в созданный проект, но поскольку корректировать ее содержание не требуется, ее описание перенесено в другой файл (а именно в файл pt4.h).

Для запуска созданной программы достаточно нажать клавишу [F5] (при использовании среды Code::Blocks используется клавиша [F9]).

**Примечание 1.** Если при попытке запуска программы система Visual Studio пытается выполнить другое действие, то следует щелкнуть правой кнопкой мыши на имени проекта ptprj в окне «Solution Explorer» («Обозреватель решений») и в появившемся контекстном меню выполнить команду «Set as StartUp Project» («Назначить запускаемым проектом»).

После запуска программы на экране появится окно задачника (рис. 2). Приведенный на рисунке вид соответствует *режиму с динамической компоновкой*, появившемуся в версии 4.11 задачника.

В дальнейшем мы всегда будем использовать режим с динамической компоновкой. Если окно отображается в «старом» режиме с фиксированной компоновкой (рис. 3), то для переключения на новый режим достаточно нажать клавишу [F4].

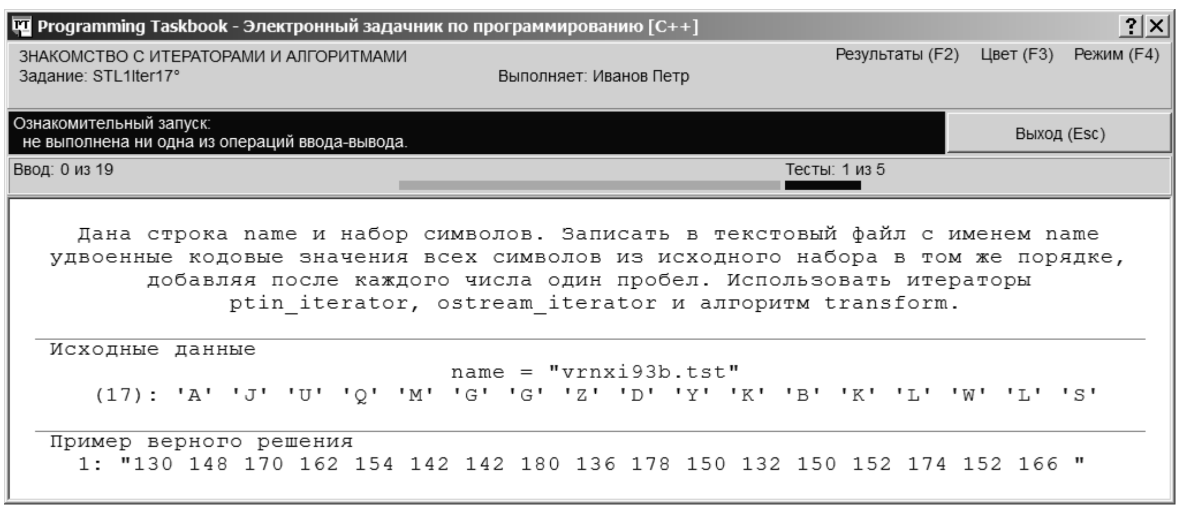


**Рис. 2**. Окно задачника в режиме с динамической компоновкой

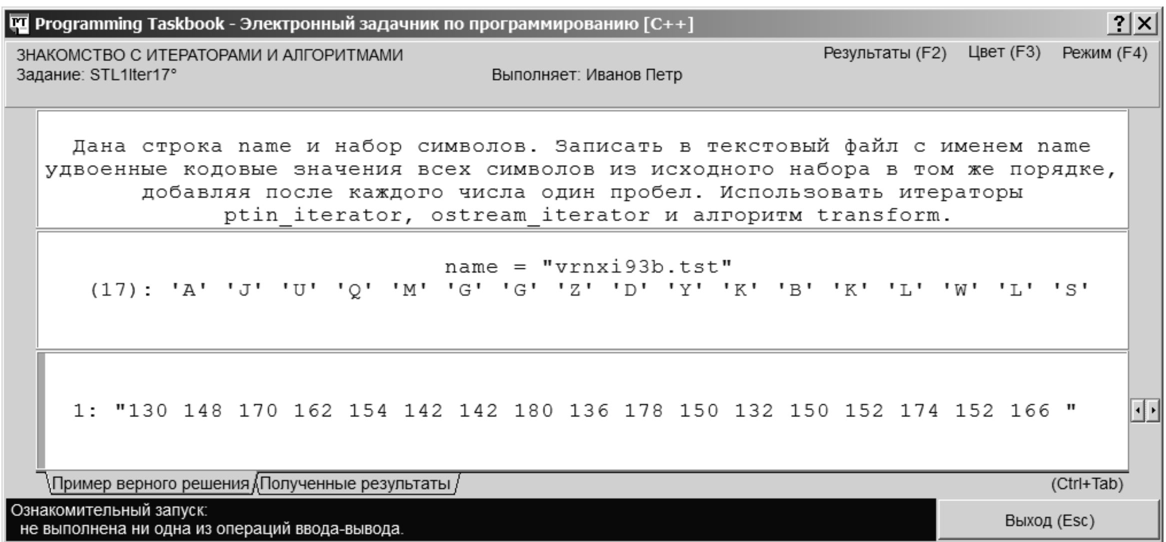


**Рис. 3**. Окно задачника в режиме с фиксированной компоновкой

Поскольку в программе не выполняются действия по вводу и выводу данных, запуск программы считается *ознакомительным*. При ознакомительном запуске в окне задачника отображаются три раздела: с формулировкой задания, исходными данными и примером верного решения.

Все исходные данные выделяются особым цветом, чтобы отличить их от комментариев: если для окна установлен цветовой режим «на белом фоне» (как на приведенных выше рисунках), то данные выделяются синим цветом; в режиме «на черном фоне» данные выделяются желтым цветом. Для переключения цветового режима достаточно нажать клавишу [F3].

Запуск программы позволяет ознакомиться с образцом исходных данных и с соответствующим этим исходным данным примером правильного решения. В нашем случае в набор исходных данных входит имя текстового файла, в который требуется записать полученную последовательность (это имя снабжено комментарием «name =» и заключено в двойные кавычки), количество элементов в исходном символьном наборе (это количество заключено в круглые скобки) и сами символьные элементы набора (каждый из которых заключен в одинарные кавычки). Во всех заданиях, входящих в задачник Programming Taskbook for STL все исходные последовательности, предлагаемые задачником (за исключением тех, которые содержатся в исходных текстовых файлах), задаются аналогичным образом: вначале указывается количество элементов последовательности, а затем – значения самих элементов (на экране количество элементов заключается в скобки и отделяется от списка значений двоеточием), причем значения могут располагаться на нескольких экранных строках.

Символьные данные заключаются в одинарные кавычки, а строковые – в двойные, что соответствует представлению символьных и строковых констант в языке C++. Использование кавычек позволяет, в частности, «увидеть» пустые строки, входящие в наборы исходных или результи-

рующих данных, или строки, начинающиеся или оканчивающиеся пробелами (как в примере верного решения, приведенном на предыдущих рисунках).

В разделе «Пример верного решения» содержится единственная текстовая строка, которая должна быть записана в результирующий файл; эта строка содержит удвоенные коды всех символов из исходного набора, причем после каждого числа располагается по одному пробелу.

Каждая строка текстового файла заключается в кавычки и выводится на отдельной экранной строке; рядом с первой строкой файла указывается ее порядковый номер, равный 1. В разделах исходных данных и результатов содержимое файлов выделяется бирюзовым цветом (цветовое выделение позволяет отличить файловые строки от других данных, а также комментариев). В разделе с правильным решением все данные выводятся серым цветом, чтобы отличить их от «настоящих» данных, найденных учебной программой.

**Примечание 2**. Если в задании используются текстовые файлы, содержащие несколько строк, то в окне задачника по умолчанию отображаются только начальные строки этих файлов, после которых указывается многоточие «…». Для того чтобы отобразить на экране все строки, содержащиеся в текстовом файле, достаточно выполнить двойной щелчок мышью в разделе окна задачника с файловыми данными. Для просмотра заданий, включающих файловые данные большого размера, в задачнике предусмотрен набор дополнительных средств; эти средства подробно описываются в п. 2.5.1 при обсуждении задания STL7Mix4. В заданиях из предыдущих групп (в том числе из группы STL1Iter) файлы либо не используются, либо имеют небольшой размер, поэтому в применении специальных средств для их просмотра нет необходимости.

Для выхода из программы достаточно нажать кнопку «Выход», клавишу [Esc] или клавишу [F5], т. е. ту же клавишу, которая обеспечивает запуск программы из среды Visual Studio (в случае использования среды Code::Blocks окно задачника можно закрыть, нажав клавишу [F9]).

Чтобы более подробно ознакомиться с заданием, можно использовать два специальных режима задачника: *демонстрационный режим* и режим отображения заданий *в формате html*.

Для запуска программы в демонстрационном режиме достаточно дополнить параметр метода Task символом «?» (в нашем случае вызов метода примет вид Task("STL1Iter17?");). Окно задачника в демонстрационном режиме имеет дополнительные кнопки, позволяющие переходить к предыдущему или последующему заданию выбранной группы, а также отображать на экране различные наборы исходных данных и связанные с ними образцы правильного решения (рис. 4).

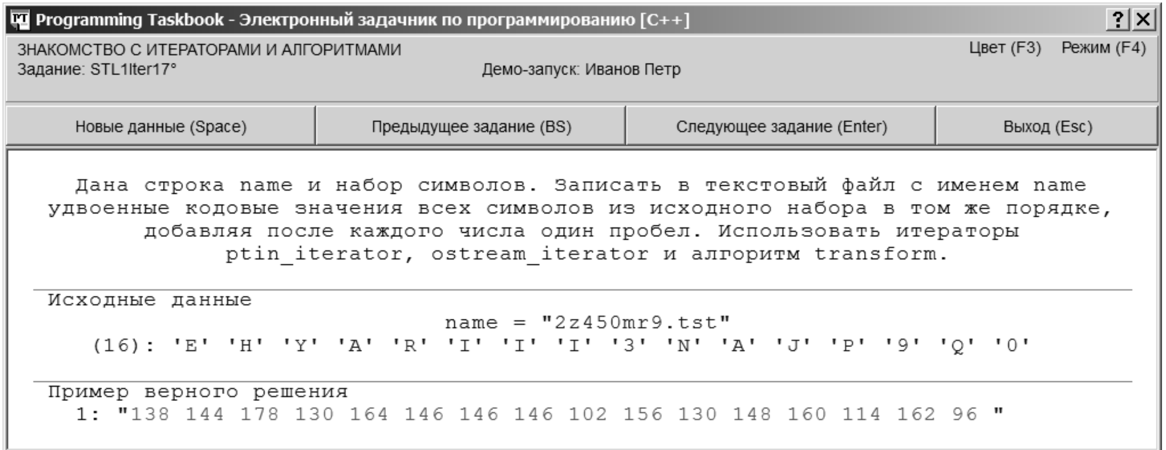


**Рис. 4**. Окно задачника в демонстрационном режиме

Для отображения задания в формате html достаточно дополнить параметр метода Task символом «#» (в нашем случае вызов метода примет вид Task("STL1Iter17#");). При запуске полученной программы будет запущен html-браузер, используемый в системе по умолчанию, и в него загрузится html-страница, содержащая формулировку задания и текст преамбулы к соответствующей группе заданий.

Режим html-страницы удобен в нескольких отношениях. Во-первых, только в этом режиме можно ознакомиться с преамбулами к группе заданий и входящим в нее подгруппам и тем самым получить дополнительные сведения, которые могут оказаться полезными при выполнении заданий. Во-вторых, окно браузера с html-страницей не требуется закрывать для того, чтобы продолжить работу над заданием; таким образом, с помощью этого окна можно в любой момент ознакомиться с формулировкой задания, даже если текущее состояние программы не позволяет откомпилировать ее и запустить на выполнение.

Имеется возможность отобразить в html-режиме *все* задания, входящие в некоторую группу. Для этого в параметре метода Task следует удалить номер задания, оставив только имя группы и символ «#» (например, Task("STL1Iter#");).

Закончив на этом обзор возможностей задачника, предназначенных для формирования проекта-заготовки и ознакомления с выбранным заданием, перейдем к описанию самого процесса решения.

### 2.1.2. Выполнение задания

Начнем выполнение задания с добавления к программе новых директив #include. Поскольку задание связано с записью данных в текстовый файл (и использованием итераторов, связанных с файловыми потоками), подключим заголовок <fstream>. Так как при выполнении задания требуется использовать один из алгоритмов STL (а именно transform), подключим

заголовок <algorithm>. В результате начальная часть файла STL1Iter 17.cpp (до описания функции Solve) примет следующий вид:

```
#include "pt4.h"
using namespace std;
#include <fstream>
#include <algorithm>
```

Заметим, что подключать заголовок <string> не требуется, хотя мы и будем использовать в программе текстовые строки.

Теперь реализуем ввод исходных данных. По условию задачи вначале дается строка – имя результирующего файла. Введем это имя в строковую переменную s, используя поток ввода pt (этот поток определен в заголовочном файле pt4.h и связан с электронным задачником). В результате функция Solve примет следующий вид:

```
void Solve()
{
    Task("STL1Iter17");
    string s;
    pt >> s;
}
```

При запуске полученного варианта программы окно задачника будет содержать сообщение о том, что введены не все требуемые исходные данные, а фон сообщения станет оранжевым (рис. 5).

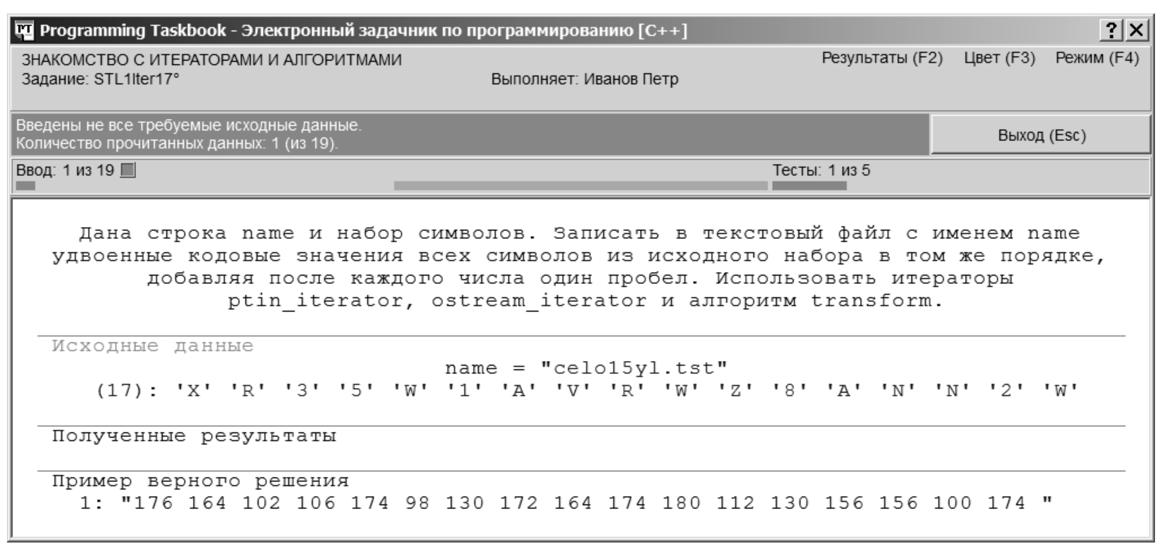


**Рис. 5**. Окно задачника с сообщением об ошибочном решении

Сообщения об ошибках всегда выводятся на фоне, цвет которого является одним из оттенков красного. Например, оранжевый цвет связывается с ситуацией, при которой введено или выведено *недостаточное* количество данных, малиновый цвет – если введено или выведено *избыточное*

количество данных, а фиолетовый – если была попытка ввести или вывести данные *неверного типа*. Таким же цветом выделяется и заголовок раздела окна, связанного с обнаруженной ошибкой (в нашем случае так выделяется заголовок с исходными данными). В разделе индикаторов, расположенном над разделом с формулировкой задания, тоже выводится оранжевый прямоугольный маркер, причем он отображается в левой части раздела (эта часть содержит информацию о вводе данных). Там же указывается, что был фактически прочитан только один элемент исходных данных из имеющихся 19.

Следует обратить внимание на то, что в средней части раздела индикаторов отсутствует информация о выведенных данных. Это связано с тем, что по условию задачи не требуется пересылать результаты задачнику; вместо этого необходимо создать новый файл и записать полученные результаты в созданный файл. В такой ситуации задачник не следит за количеством выведенных данных, а просто проверяет содержимое созданного файла. Впрочем, в нашем случае до проверки результатов дело не дошло, так как раньше была выявлена ошибка, связанная с недостаточным количеством введенных данных.

Для ввода остальных данных можно было бы прочесть количество элементов, а затем в цикле читать сами элементы и сразу их обрабатывать (или сохранять в массиве или другом подходящем контейнере). Однако в задании предлагается использовать для ввода другой способ, основанный на применении *итераторов*, – ведь именно такой способ соответствует «идеологии» стандартной библиотеки STL. Так как исходные данные генерируются задачником, для их ввода надо использовать специальный итератор для потока ввода pt, имеющий имя ptin_iterator. Итератор является шаблонным классом, который надо специализировать типом вводимых элементов; в данном случае это тип char (заметим, что аналогичным образом специализируются и стандартные итераторы файлового ввода).

Кроме того, при создании итератора обычно указывается набор параметров, определяющий свойства итератора. Если требуется создать итератор ptin_iterator, связанный с *началом* последовательности, то необходимо указать целочисленный параметр, равный количеству элементов этой последовательности. Предусмотрено особое значение для этого параметра, равное 0; в этом случае итератор сам определяет количество элементов, считывая его из потока ввода pt *перед* вводом самих элементов. Заметим, что для итераторов файлового ввода, связанных с началом последовательности, также нужно указывать параметр – имя файлового потока (количество элементов в данном случае не указывается, так как итератор файлового ввода читает данные до конца файла).

Итератор ptin_iterator, связываемый с *концом* последовательности, не имеет параметров (как и итератор, связываемый с концом файлового потока).

Итак, для указания начала и конца последовательности символов, получаемой из потока pt, при условии, что перед элементами этой последовательности передается их количество, следует использовать итераторы ptin_iterator<char>(0) и ptin_iterator<char>() соответственно. Чтобы дважды не использовать длинное имя типа, удобно определить *псевдоним* для требуемой специализации шаблона, например:

```
typedef ptin_iterator<char> ptin;
```

Как проверить, что эти итераторы обеспечат правильный ввод исходных данных? Удобным способом является *отладочная печать*, позволяющая выводить информацию в специальный раздел окна задачника (*раздел отладки*). Для отладочной печати предусмотрены функции Show и ShowLine, определенные в файле pt4.h, причем для них имеются шаблонные варианты, обеспечивающие вывод последовательности с использованием ее итераторов (в этом отношении шаблонные варианты методов Show и ShowLine с параметрами-итераторами ведут себя аналогично стандартным алгоритмам из библиотеки STL).

Добавим в конец функции Solve вызов функции Show, обеспечивающий вывод исходной последовательности в раздел отладки. При задании параметров функции Show будем использовать ранее определенный псевдоним ptin для итератора ввода. Получим следующий вариант функции Solve (перед функцией должно располагаться приведенное выше определение псевдонима ptin):

```
void Solve()
{
    Task("STL1Iter17");
    string s;
    pt >> s;
    Show(ptin(0), ptin());
}
```

Заметим, что если бы мы использовали функцию ShowLine, то каждый элемент последовательности выводился бы на новой строке раздела отладки. В вариантах функций Show и ShowLine с параметрами-итераторами можно также указывать необязательный третий параметр – строку-комментарий, которая выводится перед первым элементом последовательности.

После запуска программы окно задачника примет вид, приведенный на рис. 6.

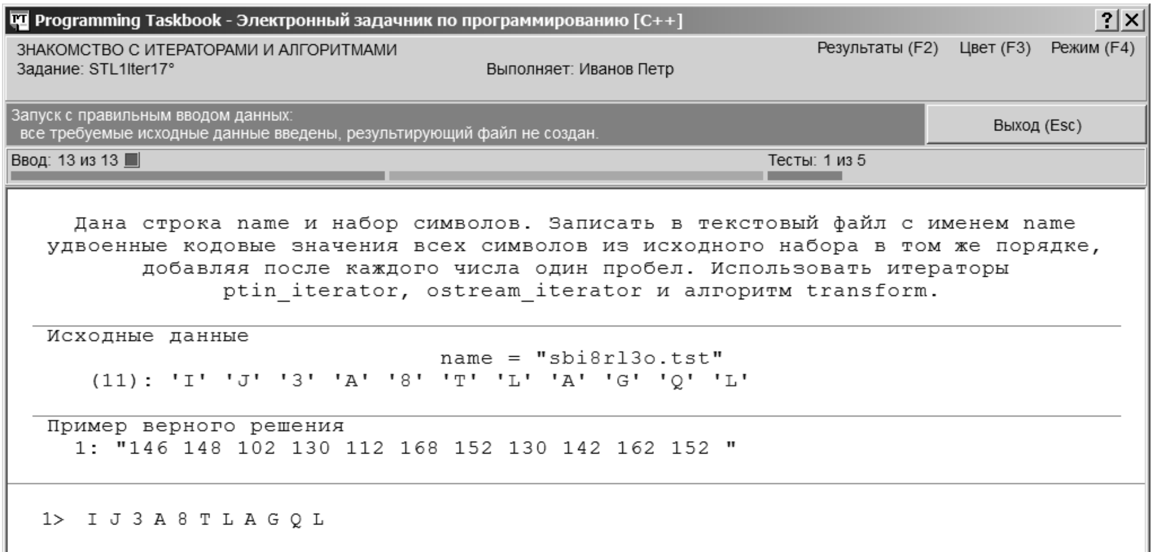


**Рис. 6**. Окно задачника при правильном вводе исходных данных

Теперь наша программа правильно вводит все исходные данные, хотя и не выводит результаты. Можно считать, что мы успешно прошли первый этап решения, связанный с вводом данных. Чтобы отметить этот факт, в разделе индикаторов, связанном с вводом данных, выводится зеленый маркер, а информационная панель содержит текст на светло-синем фоне: «*Запуск с правильным вводом данных: все требуемые исходные данные введены, результирующий файл не создан*». Поскольку результаты не выведены, в окне отсутствует раздел полученных результатов.

Следует также обратить внимание на нижнюю часть окна задачника. В ней появился раздел отладки, где выведены все элементы исходной последовательности.

В нашем варианте программы введенные данные не сохранялись в памяти, а сразу перенаправлялись функции Show для вывода в разделе отладки. Аналогичным образом можно перенаправить эти данные в результирующий текстовый файл. Однако перед записью в файл надо *преобразовать* полученные данные в соответствии с условием задачи. Для преобразования элементов исходной последовательности и получения новой последовательности, содержащей преобразованные элементы, в библиотеке STL имеется алгоритм transform, который, согласно формулировке задачи STL1Iter17, следует использовать при ее решении.

Алгоритм transform реализован в двух вариантах (см. п. 1.3.3); нас интересует вариант, принимающий *одну* входную последовательность:

```
OutIter transform(InIter first, InIter last, OutIter result,
    UnaryOp unop);
```

Первые два параметра этого алгоритма задают начало и конец исходной последовательности (в виде двух итераторов), третий параметр содержит итератор, связанный с началом преобразованной последовательности,

а последний параметр представляет собой *функциональный объект* с одним параметром, определяющий правило, по которому должны преобразовываться элементы исходной последовательности. В простейшем случае в качестве функционального объекта можно использовать указатель на обычную функцию.

Алгоритм transform возвращает итератор, указывающий на конец преобразованной последовательности.

Преобразованную последовательность не обязательно сохранять в памяти. Если использовать в качестве третьего параметра алгоритма transform итератор, связанный с файловым потоком, то преобразованная последовательность будет записана в соответствующий файл. Именно это действие и требуется выполнить в нашем задании.

Итак, для решения задания нам осталось определить функцию (назовем ее f), преобразующую символы в удвоенные значения их кодов, создать файловый поток вывода os (связав его с именем файла s) и вызвать алгоритм transform, передав ему требуемые параметры:

```
int f(char c)
{
    return 2 * c;
}

void Solve()
{
    Task("STL1Iter17");
    string s;
    pt >> s;
    ofstream os(s);
    transform(ptin(0), ptin(), ostream_iterator<int>(os, " "),
        f);
}
```

При описании функции f мы учли тот факт, что в языке C++ (в отличие от многих современных языков программирования) возможно *неявное* преобразование типа char в тип int.

В дополнительном комментарии нуждается второй, необязательный параметр конструктора ostream_iterator: в нем можно указать *разделитель*, который будет автоматически добавляться после записи в поток каждого элемента последовательности. По умолчанию разделитель отсутствует.

Заметим также, что при выполнении задания в системе Visual Studio 2008 в конструкторе класса ofstream необходимо преобразовать строку s к типу const char*, используя соответствующий метод класса string:

```
ofstream os(s.c_str());
```

Последующие версии Visual Studio и компилятор GCC, включенный в среду Code::Blocks 13, поддерживают стандарт C++11, согласно которому указанное преобразование выполнять не требуется.

При запуске полученной программы на экране появится окно прогресса тестирования, в котором отображается число пройденных тестов (рис. 7).

**Рис. 7**. Окно с индикатором прогресса тестирования

Наша программа является правильной, поэтому все пять требуемых тестов будут пройдены успешно, после чего на экране появится окно задачника с сообщением о том, что задание выполнено (рис. 8).

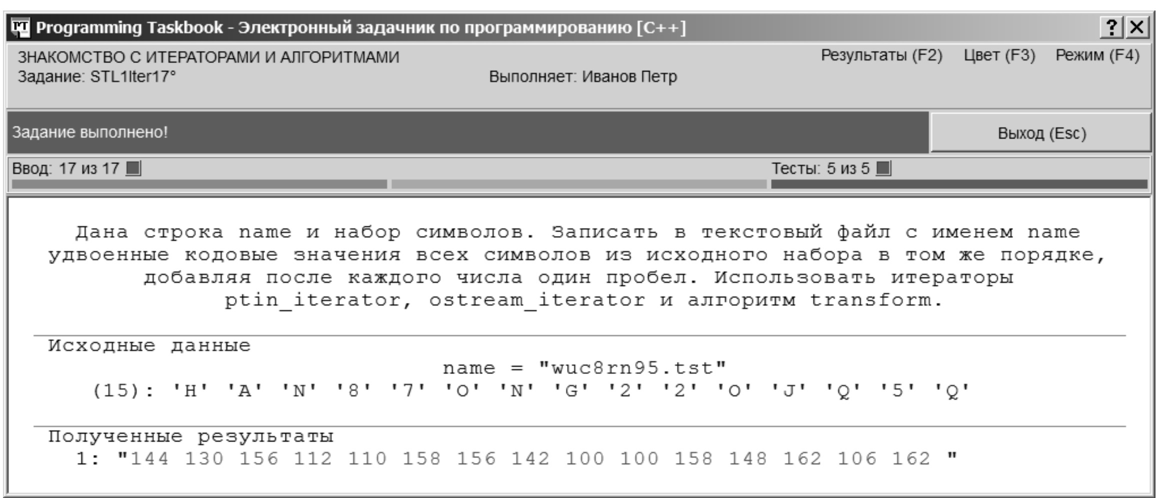


**Рис. 8**. Окно задачника с сообщением об успешном выполнении задания

Не закрывая окна задачника, можно просмотреть историю выполнения задания. Для этого достаточно нажать клавишу [F2] или щелкнуть мышью на метке «Результаты» в правом верхнем углу окна. На экране появится окно еще одного вспомогательного программного модуля задачника – PT4Results, в котором будут перечислены все запуски нашей программы:

```
STL1Iter17  c29/01 15:03 Ознакомительный запуск.--2
STL1Iter17  c29/01 15:10 Введены не все требуемые исходные данные.
STL1Iter17  c29/01 15:14 Запуск с правильным вводом данных.
STL1Iter17  c29/01 15:18 Задание выполнено!
```

Символ «c», указанный перед датой, означает, что задание выполнялось на языке C++.

### 2.1.3. Другой вариант правильного решения

В пересмотренном стандарте C++ ISO/IEC 14882:2011, для которого обычно используется обозначение C++11, появилось замечательная воз-

можность, существенно упрощающая определение и использование функциональных объектов: *лямбда-выражения*.

Лямбда-выражение представляет собой анонимный функциональный объект, определяемый в месте его непосредственного использования. Лямбда-выражения позволяют значительно сократить размеры программ, применяющих STL, и, что не менее важно, сделать эти программы более наглядными.

Лямбда-выражения можно использовать во всех версиях Visual Studio, поддерживаемых задачником, за исключением версии 2008. Их также можно использовать в системе Code::Blocks, если для подключенного к ней компилятора GNU GCC установить опцию компиляции -std=c++11. Начиная с версии задачника 4.15, эта опция устанавливается автоматически в любом проекте, генерируемом задачником для среды Code::Blocks.

Приведем вариант решения, использующий лямбда-выражение (описывать функцию f в данном случае не требуется; лямбда-выражение выделено полужирным шрифтом):

```
void Solve()
{
    Task("STL1Iter17");
    string s;
    pt >> s;
    ofstream os(s);
    transform(ptin(0), ptin(), ostream_iterator<int>(os, " "),
        [](char c){ return 2 * c; });
}
```

Простейший вариант лямбда-выражения (не выполняющего захват внешних переменных) начинается с пустых квадратных скобок, за которыми идет список параметров и тело определяемого функционального объекта, причем если это тело содержит единственный оператор return (как обычно и бывает), то не требуется указывать тип возвращаемого значения.

**Примечание.** В формулировках некоторых заданий группы STL1Iter говорится, что исходные данные размещаются во внешнем текстовом файле, а полученные результаты следует переслать непосредственно задачнику (используя поток вывода pt). В этом случае для ввода надо применять стандартный итератор чтения из файлового потока istream_iterator, а для вывода – итератор ptout_iterator, определенный в файле pt4.h. Возможны и другие комбинации размещения входных и выходных данных. При этом во всех заданиях этой группы достаточно выполнять преобразование входной последовательности в выходную «на лету», применяя подходящий алгоритм и не используя вспомогательные контейнеры.

## 2.2. *Работа с последовательными контейнерами: STL2Seq22*

### 2.2.1. Создание проекта-заготовки и знакомство с заданием

*Контейнеры* являются важнейшими компонентами библиотеки STL. Именно к ним, как правило, применяются алгоритмы с целью анализа содержимого контейнеров, преобразования их элементов или создания новых контейнеров с преобразованными данными. И хотя с помощью потоковых итераторов можно выполнять преобразования наборов данных «на лету», не сохраняя их в контейнерах, подобные действия выполняются сравнительно редко. Мы использовали такой вид преобразований при выполнении заданий группы STL1Iter, предназначенной для начального знакомства с итераторами и алгоритмами. В последующих группах задачника PT for STL, наряду с итераторами и алгоритмами, активно применяются контейнеры. Вторая группа заданий STL2Seq посвящена возможностям *последовательных контейнеров*: векторов, деков и списков.

В группе STL2Seq основное внимание уделяется *функциям-членам* последовательных контейнеров и особенностям их использования. Алгоритмы в заданиях этой группы применяются редко, причем только те, которые уже были рассмотрены в группе STL1Iter. Для детального изучения различных видов алгоритмов предназначена третья группа заданий STL3Alg.

Группа STL2Seq состоит из трех подгрупп. Первая подгруппа посвящена способам заполнения последовательных контейнеров и доступу к их элементам, во второй рассматриваются способы вставки элементов в контейнер, а в третьей – способы их удаления. В первой подгруппе изучаются также особенности *обратных итераторов*, которые придется часто применять при выполнении последующих заданий. В каждой подгруппе задания расположены по возрастанию уровня сложности.

Для того чтобы познакомиться с особенностями выполнения заданий группы STL2Seq, рассмотрим одно из начальных заданий, входящих с третью подгруппу, посвященную удалению элементов:

STL2Seq22. Даны список *L* и вектор *V*; список *L* имеет нечетное количество элементов. Переместить средний элемент списка *L* в конец вектора *V*. Использовать функции-члены push_back и erase.

При генерации проекта-заготовки для данного задания с помощью программы PT4Load будет создан файл STL2Seq22.cpp, который сразу загрузится в редактор используемой среды программирования. Приведем содержимое этого файла:

```
#include "pt4.h"
using namespace std;
```

```
#include <iterator>
#include <vector>
#include <deque>
#include <list>

typedef ptin_iterator<int> ptin;
typedef ptout_iterator<int> ptout;

void Solve()
{
    Task("STL2Seq22");
    list<int>   L(ptin(0), ptin());
    vector<int> V(ptin(0), ptin());

    Show(L.begin(), L.end(), "L: ");
    Show(V.begin(), V.end(), "V: ");
    //copy(L.begin(), L.end(), ptout());
    //copy(V.begin(), V.end(), ptout());
}
```

Заготовка для данного задания является гораздо более развернутой, чем заготовки для заданий группы STL1Iter. В ней, помимо подключения заголовочного файла pt4.h, выполняется подключение ряда заголовочных файлов библиотеки STL и определяются короткие псевдонимы для итераторов чтения и записи, связанных с задачником. Кроме того, в функции Solve не только вызывается функция Task, но и выполняется считывание исходных контейнеров и их отладочная печать. В конце функции Solve располагаются закомментированные операторы, позволяющие переслать преобразованные контейнеры задачнику для их проверки.

Итак, текст заготовки уже содержит фрагменты программы, которые были подробно изучены при выполнении заданий начальной группы STL1Iter (эти фрагменты обеспечивают подключение необходимых заголовочных файлов библиотеки STL и определение псевдонимов итераторов). Заготовка также содержит фрагменты, выполняющие ввод, отладочную печать и вывод результатов. Заметим, что заготовки для первых пяти заданий группы STL2Seq, в которых изучаются действия по вводу и выводу контейнеров, *не содержат* подобных фрагментов и являются столь же краткими, как и заготовки для заданий группы STL1Iter. Таким образом, в заготовку включены те фрагменты программы, которые *уже были изучены при решении предыдущих задач*. Это позволяет сразу приступить к программной реализации алгоритма решения, не отвлекаясь на выполнение стандартных предварительных действий.

При запуске созданного проекта на экране появится окно задачника, примерный вид которого приведен на рис. 9.

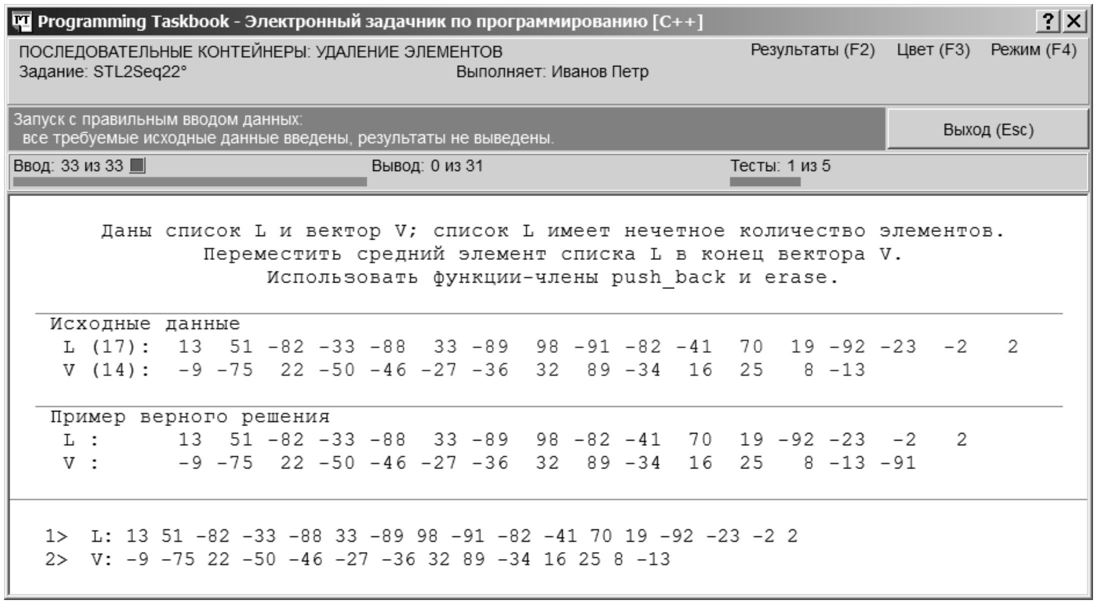


**Рис. 9**. Окно задачника при запуске программы-заготовки для задания STL2Seq22

Содержимое окна показывает, что операторы, входящие в заготовку, обеспечили правильный ввод исходных контейнеров. Нам осталось преобразовать контейнеры в соответствии с условиями задачи. Преобразования следует выполнять перед операторами отладочной печати, поскольку по содержимому раздела отладки мы сможем проверить правильность сделанных преобразований. Убедившись в их правильности, мы раскомментируем два последних оператора заготовки, что позволит переслать преобразованные контейнеры задачнику для их окончательной проверки.

### 2.2.2. Выполнение задания

Прежде всего, нам необходимо получить доступ к среднему элементу списка L. Поскольку итераторы списка не являются итераторами произвольного доступа, мы не можем использовать для этих целей операции индексирования или смещения итератора на произвольное количество элементов. Однако мы можем применить функцию advance из заголовочного файла <iterator>, предварительно определив размер списка с помощью функции size:

```
list<int>::iterator p = L.begin();
advance(p, L.size()/2);
ShowLine(*p);
```

Данный фрагмент следует ввести непосредственно перед операторами отладочной печати, уже имеющимися в тексте заготовки. При запуске полученной программы мы сможем убедиться в том, что средний элемент списка определен нами правильно (рис. 10).

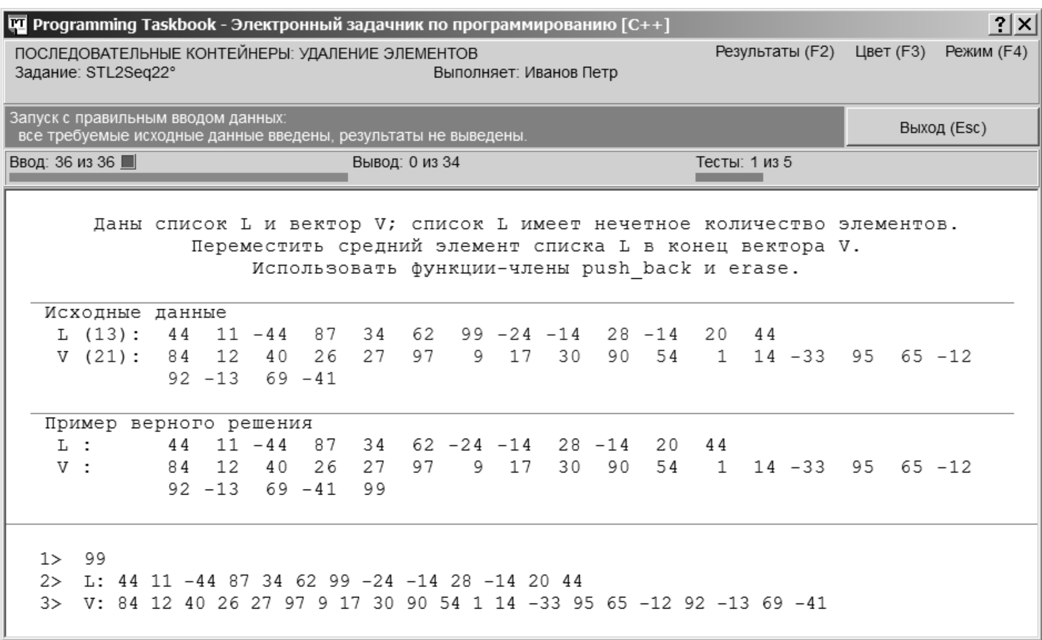


**Рис. 10**. Окно задачника на начальном этапе выполнения задания STL2Seq22

Для добавления этого элемента в конец вектора V следует использовать функцию-член push_back, а для удаления его из списка L – функцию-член erase:

```
V.push_back(*p);
L.erase(p);
```

Запустив полученную программу и сравнив результаты, выведенные в разделе отладки, с примером правильного решения (рис. 11), мы можем убедиться, что контейнеры преобразованы требуемым образом.

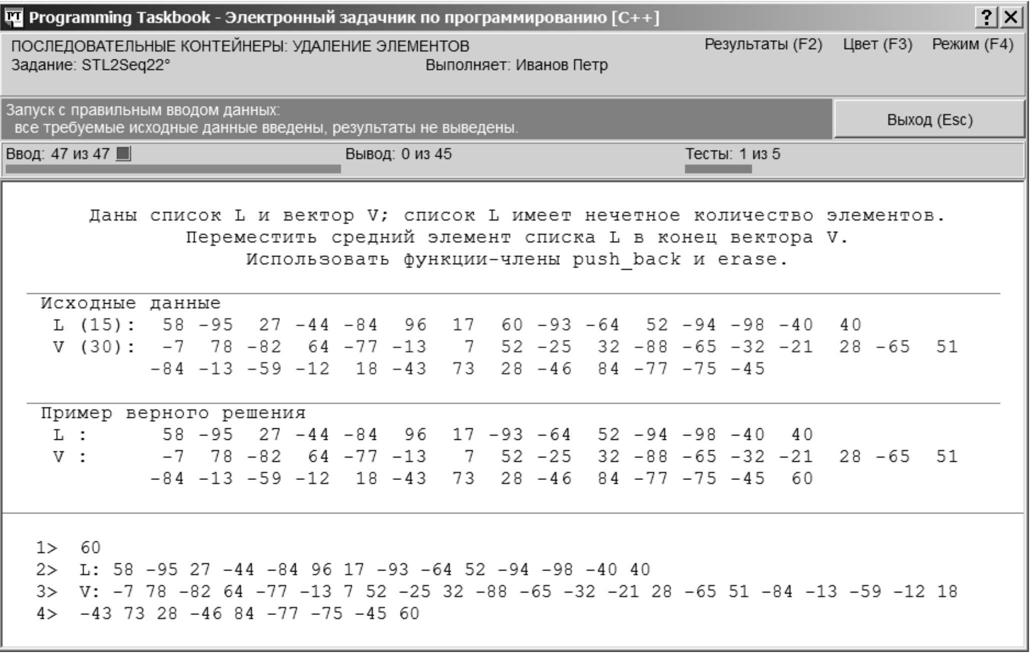


**Рис. 11**. Окно задачника на заключительном этапе выполнения задания STL2Seq22

Нам осталось раскомментировать два последних оператора функции Solve, которые обеспечивают передачу преобразованных контейнеров задачнику для проверки их правильности. Кроме того, мы можем теперь закомментировать или удалить операторы отладочной печати. Приведем окончательный вид функции Solve с решением задачи:

```
void Solve()
{
    Task("STL2Seq22");
    list<int>   L(ptin(0), ptin());
    vector<int> V(ptin(0), ptin());
    list<int>::iterator p = L.begin();
    advance(p, L.size()/2);
    V.push_back(*p);
    L.erase(p);
    copy(L.begin(), L.end(), ptout());
    copy(V.begin(), V.end(), ptout());
}
```

При запуске данного варианта мы получим сообщение о том, что задание выполнено (рис. 12).

**Рис. 12**. Окно задачника с сообщением об успешном выполнении задания STL2Seq22

**Примечание.** В тексте решения можно избавиться от явного указания типа итератора list<int>::iterator, если воспользоваться возможностью *автоматического вывода типа*, появившейся в стандарте C++11. С применением этой возможности оператор, определяющий итератор p, примет следующий вид:

```
auto p = L.begin();
```

Подобный вариант описания переменной p можно применять во всех средах, поддерживаемых задачником PT for STL, за исключением среды Visual Studio 2008.

## 2.3. Использование алгоритмов

### 2.3.1. Преобразование списка с использованием различных видов итераторов: STL3Alg35

Детальному знакомству с *алгоритмами*, входящими в библиотеку STL, посвящена группа STL3Alg. Для заданий этой группы (как и группы STL2Seq) создаются заготовки, уже содержащие фрагменты, обеспечивающие ввод, отладочную печать и вывод используемых в задании контейнеров. Задания в группе STL3Alg разбиты на подгруппы, каждая их которых посвящена определенному набору «родственных» алгоритмов.

В данном пункте мы рассмотрим последнее задание из подгруппы, посвященной модифицирующим алгоритмам. Это задание является одним из наиболее сложных в данной подгруппе.

STL3Alg35. Дан список $L$ с четным количеством элементов $N$. Добавить в начало списка $N/2$ новых элементов со следующими значениями: $A_1 + A_N$, $A_2 + A_{N-1}$, …, $A_{N/2} + A_{N/2+1}$, где $A_1$, $A_2$, …, $A_N$ обозначают исходные элементы списка. Использовать алгоритм transform с обратным итератором и итератором вставки.

Файл STL4Alg35.cpp, созданный программой PT4Load, содержит следующий текст:

```
#include "pt4.h"
using namespace std;

#include <algorithm>
#include <iterator>
#include <vector>
#include <deque>
#include <list>
#include <functional>

typedef ptin_iterator<int> ptin;
typedef ptout_iterator<int> ptout;

void Solve()
{
    Task("STL3Alg35");
    list<int> L(ptin(0), ptin());
```

```
    Show(L.begin(), L.end(), "L: ");
    //copy(L.begin(), L.end(), ptout());
}
```

Примерный вид окна задачника, отображаемого на экране при запуске созданного проекта, приведен на рис. 13.

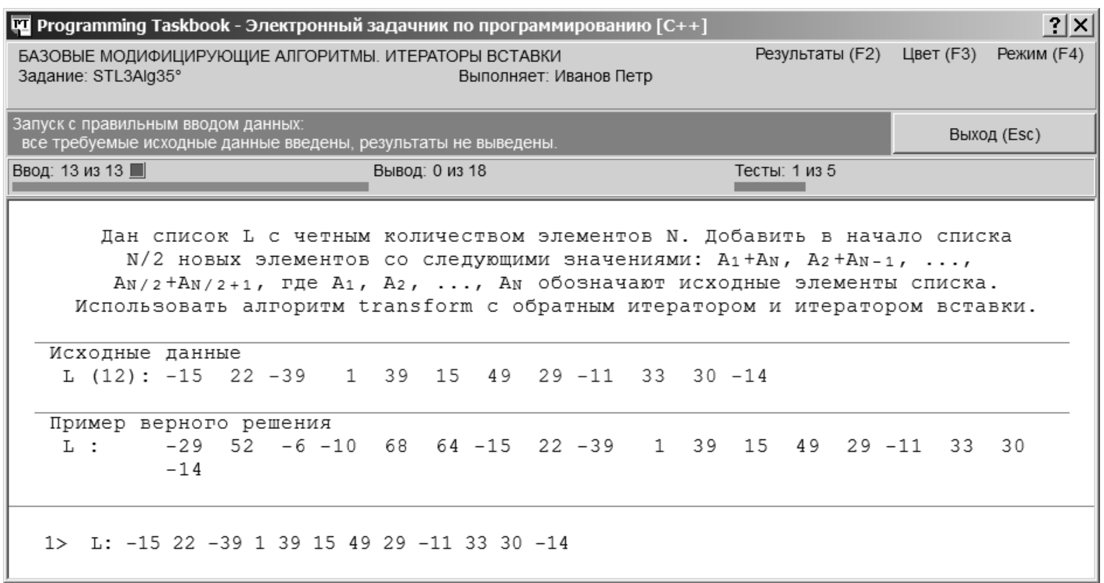


**Рис. 13**. Окно задачника при запуске программы-заготовки для задания STL3Alg35

Анализируя формулировку задания, мы видим, что добавляемый набор элементов может быть получен при суммировании соответствующих элементов первой и второй половины исходного списка, причем элементы второй половины должны перебираться в обратном порядке. Именно такой вид преобразования обеспечивается вариантом алгоритма transform, содержащим два диапазона итераторов чтения (см. п. 1.3.3):

```
OutIter transform(InIter1 first1, InIter1 last1, InIter2 first2,
    OutIter result, BinaryOp binop)
```

При этом в качестве итератора записи result необходимо использовать *итератор вставки*, позволяющий добавлять в контейнер новые элементы. Заметим, что в данном случае вставка элементов в тот же контейнер, из которого берутся исходные данные, не приведет к проблемам, так как в задании используется *список* (контейнер типа list), а добавление новых элементов к списку сохраняет корректными все его итераторы (см. п. 1.2.7).

Вначале мы должны определить итератор, связанный с серединой исходного списка (заметим, что подобное действие мы уже выполняли при решении задачи STL2Seq22):

```
list<int>::iterator p = L.begin();
advance(p, L.size()/2);
```

Теперь итератор p указывает на позицию *за* первой половиной элементов исходного списка L. Таким образом, для определения диапазона, содержащего первую половину элементов списка L (в исходном порядке), нам достаточно использовать итераторы L.begin() и p. Для перебора второй половины элементов в обратном порядке нам надо воспользоваться *обратным итератором* L.rbegin() (см. п. 1.2.9), причем итератор, определяющий конец диапазона, использовать не требуется, так как в алгоритме transform предполагается, что оба диапазона имеют одинаковый размер.

В качестве последнего параметра алгоритма transform – функционального объекта binop – в данном случае можно использовать стандартный фунциональный объект plus, параметризованный типом int: plus<int>() (см. п. 1.4.3).

На начальном этапе решения убедимся, что алгоритм transform выполняет требуемое преобразование. Для этого не обязательно добавлять преобразованные элементы в начало списка; можно просто переслать их задачнику, чтобы он отобразил эти элементы в разделе результатов. Чтобы выполнить подобную пересылку, достаточно указать в алгоритме transform в качестве параметра result *итератор записи* ptout, связанный с потоком вывода pt:

```
transform(L.begin(), p, L.rbegin(), ptout(), plus<int>());
```

При запуске этого варианта решения мы получим результат, приведенный на рис. 14.

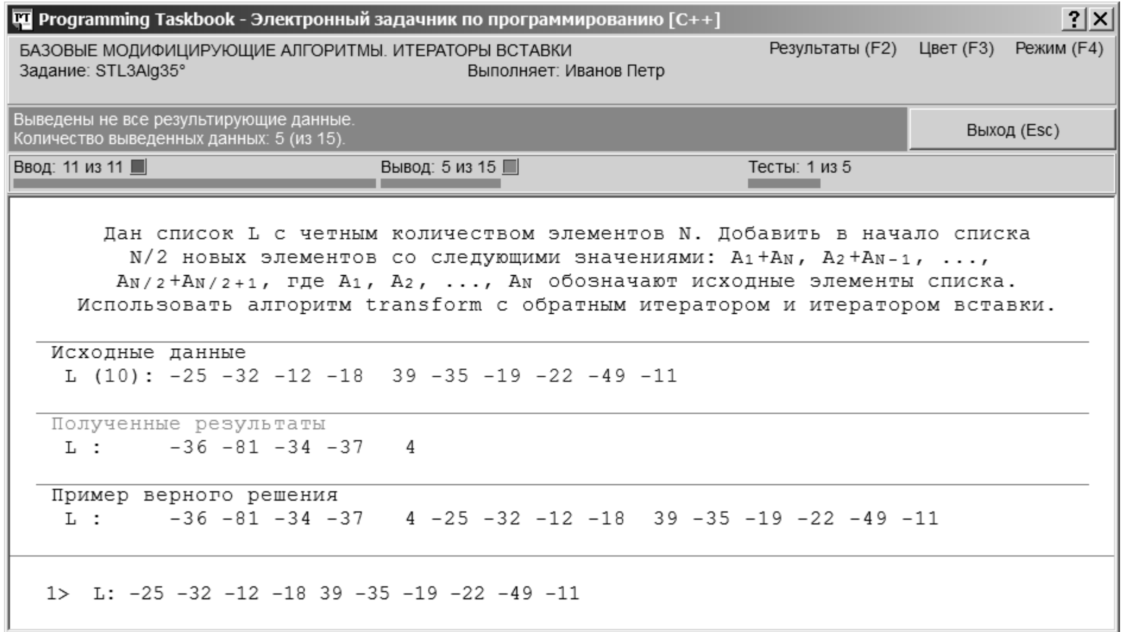


**Рис. 14**. Окно задачника на начальном этапе выполнения задания STL3Alg35

В данном случае, как и следовало ожидать, задачник сообщает о том, что выведены не все результирующие данные. Однако мы видим, что ото-

бражаемые в разделе результатов значения совпадают с теми, которые необходимо добавить в начало списка.

Итак, мы убедились, что алгоритм transform обеспечивает требуемое преобразование элементов. Нам осталось изменить его параметр result таким образом, чтобы преобразованные элементы не пересылались задачнику, а добавлялись в начало списка в том же порядке. Для этого необходимо воспользоваться итератором вставки inserter (см. п. 1.3.5). Приведем новый вариант вызова алгоритма transform, в котором полужирным шрифтом выделен измененный параметр:

```
transform(L.begin(), p, L.rbegin(), inserter(L, L.begin()),
    plus<int>());
```

Запустив новый вариант программы и проанализировав раздел отладки, мы можем убедиться, что преобразованный список содержит именно те элементы, которые требуются в задании (рис. 15).

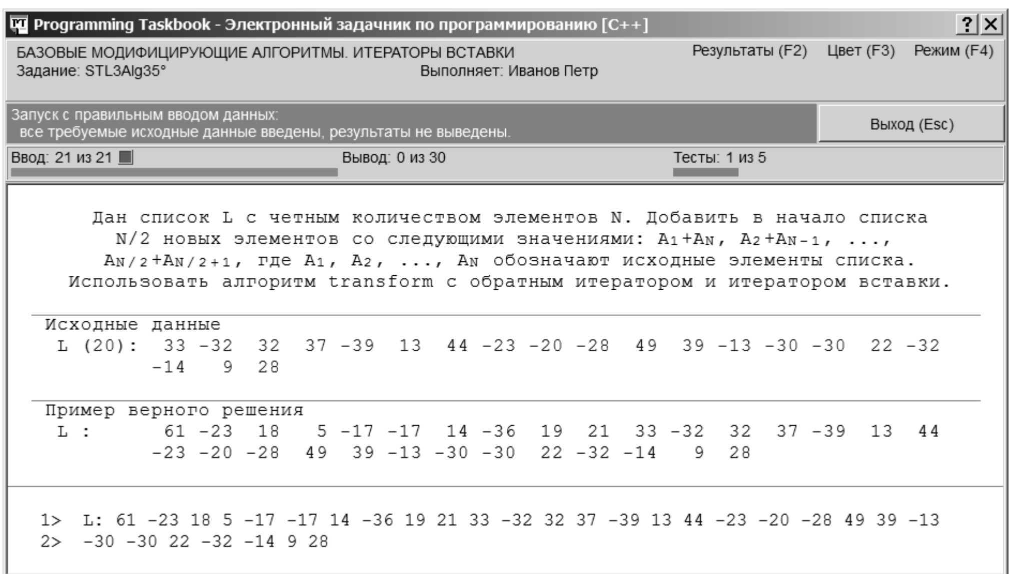


**Рис. 15**. Окно задачника на заключительном этапе выполнения задания STL3Alg35

Нам осталось раскомментировать последний оператор функции Solve, обеспечивающий пересылку элементов преобразованного списка задачнику. Можно также закомментировать или удалить вызов функции Show. Приведем окончательный вид функции Solve:

```
void Solve()
{
    Task("STL3Alg35");
    list<int> L(ptin(0), ptin());
    list<int>::iterator p = L.begin();
    advance(p, L.size()/2);
    transform(L.begin(), p, L.rbegin(), inserter(L, L.begin()),
        plus<int>());
```

```
        copy(L.begin(), L.end(), ptout());
    }
```

После запуска программы мы получим сообщение о том, что задание выполнено. Таким образом, мы смогли выполнить достаточно сложное преобразование исходного списка, использовав единственный обобщенный алгоритм, однако в нем нам потребовалось применить как прямые, так и обратные итераторы, а также итератор вставки.

**Примечание.** Новые возможности, добавленные в язык C++ стандартом C++11, позволяют избавиться не только от явного указания типа итератора list<int>::iterator, но и от стандартного функционального объекта plus<int>(), заменив его *лямбда-выражением* (см. п. 2.1.3). Приведем соответствующий вариант решения:

```
    void Solve()
    {
        Task("STL3Alg35");
        list<int> L(ptin(0), ptin());
        auto p = L.begin();
        advance(p, L.size()/2);
        transform(L.begin(), p, L.rbegin(), inserter(L, L.begin()),
            [](int e1, int e2){ return e1 + e2; });
        copy(L.begin(), L.end(), ptout());
    }
```

### 2.3.2. Преобразование символьных строк: STL4Str27

В заданиях группы STL3Alg всегда указывалось, какие именно алгоритмы следует применять для требуемого анализа или преобразования контейнера. Подобные «подсказки» позволяли выполнять задания оптимальным способом и демонстрировали ситуации, в которых применение того или иного алгоритма является наиболее оправданным.

Группа STL4Str предназначена для закрепления полученных знаний об алгоритмах. В ней требуется выполнять преобразования *символьных строк*, которые тоже могут считаться последовательными контейнерами, поскольку состоят из элементов-символов. При этом способ решения задачи не уточняется: необходимо самостоятельно выбрать оптимальный вариант, используя знания, полученные при выполнении заданий группы STL3Alg.

В качестве примера рассмотрим одно из последних заданий группы STL4Str.

STL4Str27. Дана строка *S*. Используя подходящий обобщенный алгоритм, перегруппировать элементы строки *S*, переместив все цифровые символы в ее начало в том же порядке.

Заготовка для этого задания выглядит следующим образом:

```
#include "pt4.h"
using namespace std;

#include <algorithm>
#include <iterator>
#include <string>

void Solve()
{
    Task("STL4Str27");
}
```

Заготовка содержит директивы подключения всех необходимых заголовочных файлов, однако в функции Solve не выполняется никаких действий, кроме инициализации задания. Это объясняется тем, что для ввода и вывода строк не требуется применять специальные средства стандартной библиотеки, связанные с итераторами, – достаточно воспользоваться потоком ввода-вывода pt. Заметим, что для отладочного вывода строки s также достаточно использовать простейший вариант функции Show: Show(s).

Запустив созданную заготовку, мы, как обычно, увидим в окне задачника образец исходных данных и пример правильного решения (рис. 16).

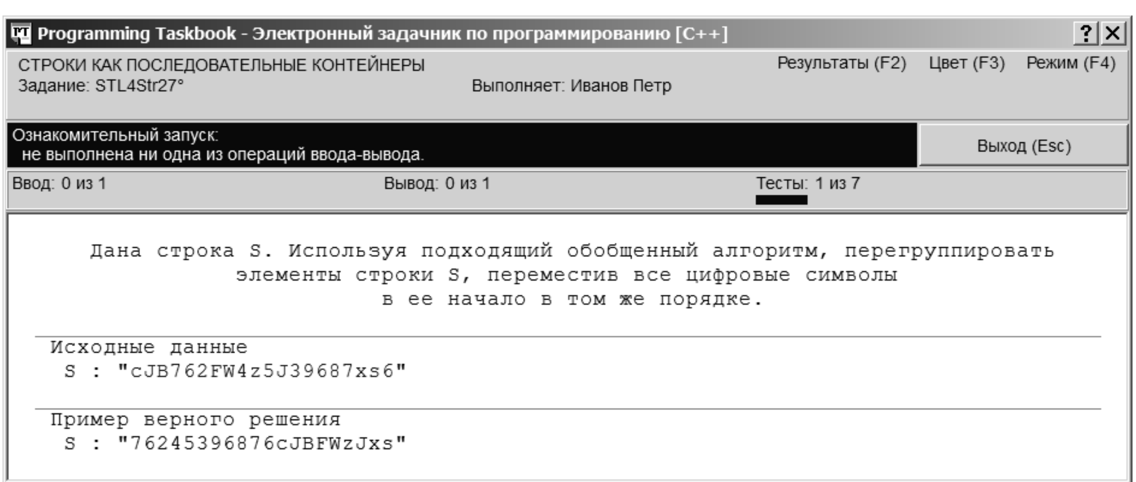


**Рис. 16**. Окно задачника при запуске программы-заготовки для задания STL4Str27

Добавим в функцию Solve операторы, обеспечивающие ввод исходной строки и ее вывод в раздел результатов:

```
string s;
pt >> s;
pt << s;
```

При запуске этого варианта программы в окне задачника будет указано, что введены и выведены все данные, однако полученная строка отличается от требуемой (рис. 17).

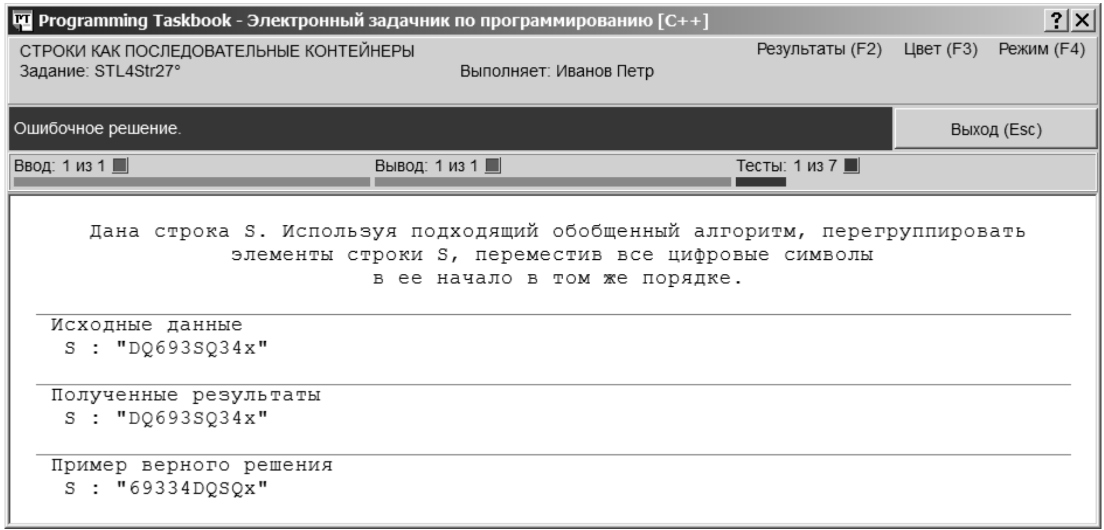


**Рис. 17**. Окно задачника на начальном этапе выполнения задания STL4Str27

Для решения задачи нам необходимо выбрать подходящий алгоритм. В данном случае требуется выполнить *перегруппировку* символов, причем в результате перегруппировки порядок символов в каждой группе должен *сохраниться*. Именно это действие обеспечивает обобщенный алгоритм stable_partition, ранее изученный в подгруппе группы STL3Alg, посвященной сортировке и слиянию:

```
BidiIter stable_partition(BidiIter first, BidiIter last,
    Predicate pred)
```

В качестве функционального объекта-предиката можно использовать следующую функцию:

```
bool p(char c)
{
    return '0' <= c && c <= '9';
}
```

Определив эту функцию, мы укажем ее в алгоритме stable_partition, который надо вызвать между операторами ввода и вывода строки s:

```
stable_partition(s.begin(), s.end(), p);
```

Запустив этот вариант программы, мы получим сообщение о том, что задание выполнено.

**Примечание 1.** Если компилятор поддерживает стандарт C++11, то вместо функции p можно воспользоваться лямбда-выражением:

```
void Solve()
{
```

```
    Task("STL4Str27");
    string s;
    pt >> s;
    stable_partition(s.begin(), s.end(),
        [](char c){ return '0' <= c && c <= '9'; });
    pt << s;
}
```

**Примечание 2.** Если воспользоваться стандартной функцией isdigit, то при решении можно обойтись и без вспомогательной функции, и без лямбда-выражения:

```
    stable_partition(s.begin(), s.end(), isdigit);
```

## *2.4. Использование ассоциативных контейнеров и дополнительных видов функциональных объектов*

### 2.4.1. Группировка данных с применением отображений: STL5Assoc20

Группа STL5Assoc посвящена *ассоциативным контейнерам*. Она состоит из двух подгрупп: в первую подгруппу входят задания, связанные с контейнерами-*множествами* (set и multiset), а во вторую – задания на обработку контейнеров-*отображений* (map и multimap). Во многих заданиях из второй группы требуется выполнить *группировку* или *объединение* исходных наборов данных, поскольку именно такие действия можно эффективно реализовать с помощью различных вариантов контейнеров-отображений.

В качестве иллюстрации рассмотрим одно из заданий второй подгруппы группы STLAssoc, связанных с группировкой данных.

STL5Assoc20. Дан вектор *V*, элементами которого являются английские слова, набранные заглавными буквами. Выполнить группировку элементов вектора *V*, используя в качестве ключа группировки первую букву элемента: в одну группу должны входить все элементы вектора *V*, начинающиеся с одной и той же буквы (сгруппированные элементы должны располагаться в алфавитном порядке с учетом повторений). Представить результат группировки в виде отображения *M*, ключами которого являются ключи группировки, а значениями – мультимножества, содержащие сгруппированные элементы (таким образом, отображение *M* должно иметь тип map<char, multiset<string>>). Вывести полученное отображение (для каждого элемента отображения *M* вначале выводить ключ, а затем элементы связанного с ним мультимножества). Для перебора элементов контейнеров использовать алгоритм for_each или, если компилятор поддерживает стандарт C++11, цикл for по элементам контейнера.

Заготовка для этого задания имеет следующий вид:

```
#include "pt4.h"
using namespace std;

#include <iterator>
#include <vector>
#include <set>
#include <map>
#include <functional>
#include <algorithm>

typedef ptin_iterator<string> ptin;

void Solve()
{
    Task("STL5Assoc20");
    vector<string> V(ptin(0), ptin());
}
```

В ней содержатся директивы подключения всех необходимых заголовочных файлов, и, кроме того, в функции Solve выполняется ввод исходного вектора V. Все последующие действия, связанные с формированием требуемого ассоциативного контейнера и выводом его элементов, надо реализовать самостоятельно.

На рис. 18 приведен вид окна задачника, которое выводится на экран при запуске проекта-заготовки.

Приступим к решению задачи. Нам необходимо выполнить *группировку* исходного набора строк, включив в каждую группу строки, которые начинаются с одного и того же символа. Строки в исходном наборе могут повторяться; в этом случае группа должна содержать все экземпляры повторяющихся строк.

Для выполнения группировки в задании предлагается использовать отображение (контейнер map). При этом в качестве *ключа* отображения надо указать ключ группировки, а в качестве *значения* отображения – контейнер, который наилучшим образом соответствует требуемому варианту группировки. Поскольку в нашем случае надо выполнять группировку *с учетом повторений* и, кроме того, при выводе полученных групп необходимо *упорядочивать их элементы по алфавиту*, наиболее подходящим вариантом для хранения элементов группы является *мультимножество* (контейнер multiset). Таким образом, результат группировки следует представить в виде контейнера типа map<char, multiset<string>>. Заметим, что если бы требовалось, например, выполнять группировку без повторений,

то для хранения элементов группы следовало использовать *множество* (контейнер set), а если бы при группировке надо было сохранять исходный порядок расположения элементов, то группы было бы удобно хранить в виде *векторов* (контейнеров vector). В каждом задании группы STL5Assoc, посвященном группировке или объединению данных, всегда указывается, какой именно вариант ассоциативного контейнера требуется для этого использовать.

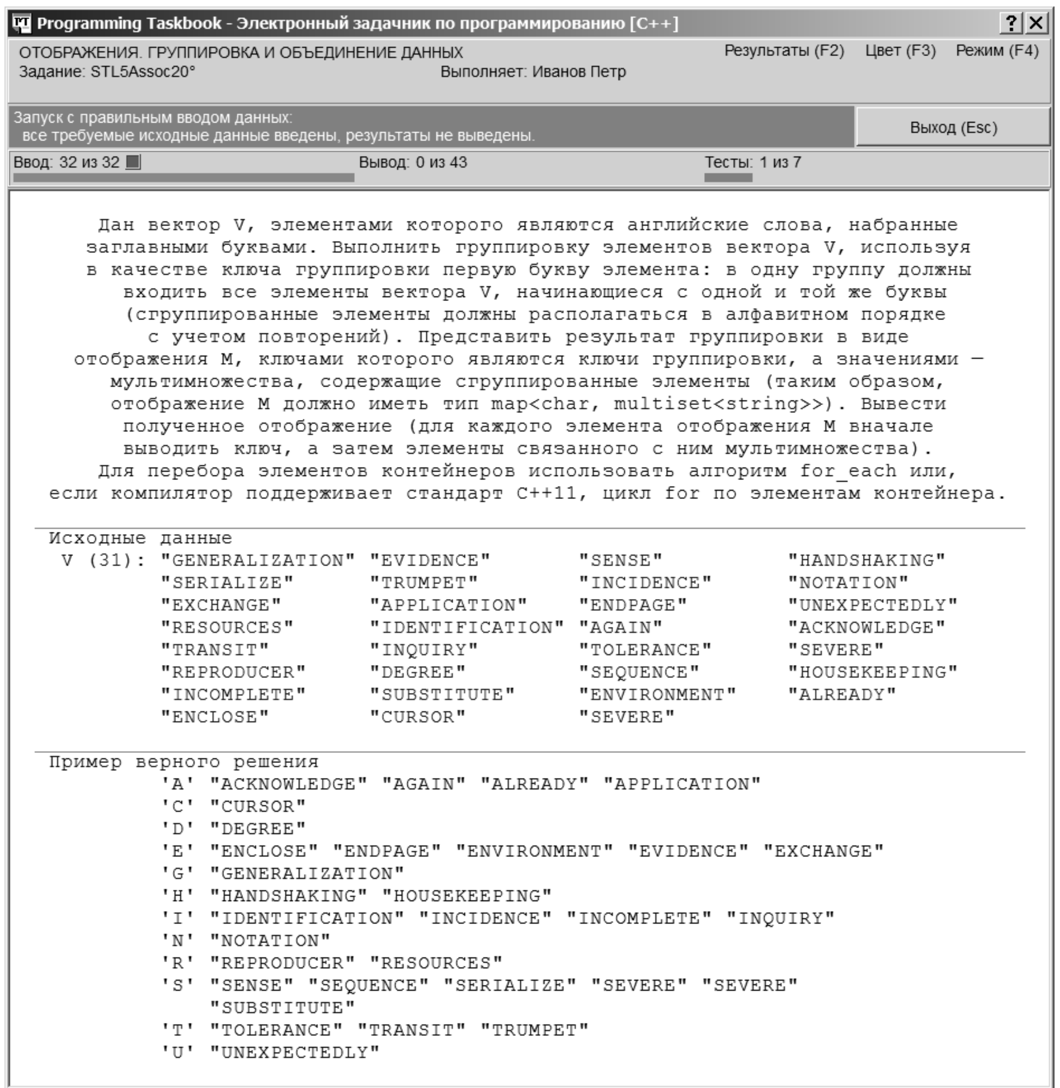


**Рис. 18**. Окно задачника при запуске программы-заготовки для задания STL5Assoc20

В формулировке задания также содержатся указания по организации перебора элементов контейнеров. Для этой цели можно применять алгоритм for_each или, если используемый компилятор поддерживает стандарт C++11, цикл for *по элементам контейнера*. В качестве альтернативы алгоритму for_each можно также использовать цикл for с параметром-итератором.

Вначале решим задачу без применения новых возможностей стандарта C++11. На первом этапе следует сформировать отображение M, содержа-

щее наборы сгруппированных строк. Для этого надо организовать перебор элементов исходного вектора V, добавляя элемент s в то мультимножество, которое соответствует ключу отображения M со значением s[0].

В соответствии с указаниями, приведенными в задании, оформим перебор элементов вектора V в виде алгоритма for_each. В этом случае требуется дополнительно описать функциональный объект, определяющий действие, которое будет выполняться в данном алгоритме. Вместо функционального объекта достаточно описать обычную функцию (назовем ее f1). Поскольку в этой функции необходимо обращаться к контейнеру M, опишем этот контейнер перед функцией. Приведем полученный вариант программы, не указывая начальный фрагмент заготовки (с директивами #include и typedef) и выделяя добавленные операторы полужирным шрифтом:

```
map<char, multiset<string>> M;

void f1(string s)
{
    M[s[0]].insert(s);
}

void Solve()
{
    Task("STL5Assoc20");
    vector<string> V(ptin(0), ptin());
    for_each(V.begin(), V.end(), f1);
}
```

Напомним, что контейнер map позволяет использовать операцию индексирования [ ] по ключу, причем если указанный ключ еще не существует, то автоматически создается элемент контейнера map с данным ключом и пустым содержимым (см. п. 1.2.6).

**Примечание**. Согласно стандарту C++98 при описании шаблонных типов необходимо разделять рядом стоящие символы > хотя бы одним пробелом, например: map<char, multiset<string> >. Однако в стандарте C++11 это требование отменено; кроме того, компиляторы C++ для всех рассматриваемых сред (в том числе и для среды Visual Studio 2008) правильно интерпретируют описания шаблонных типов без дополнительных пробелов.

Было бы желательно просмотреть содержимое полученного отображения M. Это легко сделать с помощью шаблонных реализаций функций Show и ShowLine, которые уже использовалась при выполнении предыдущих заданий. Добавим в конец функции Solve следующий оператор:

```
        ShowLine(M.begin(), M.end());
```

При запуске полученного варианта программы на экране появится окно задачника, примерный вид которого приведен на рис. 19. Для уменьшения размеров окна в нем свернут раздел с формулировкой; подобное действие можно выполнить несколькими способами: или дважды щелкнув мышью на данном разделе, или нажав клавишу [Del], или щелкнув мышью на маркере ×, расположенном в правом верхнему углу раздела с формулировкой (в результате вид маркера изменится на □). Повторное выполнение указанных действий восстанавливает содержимое раздела с формулировкой задания.

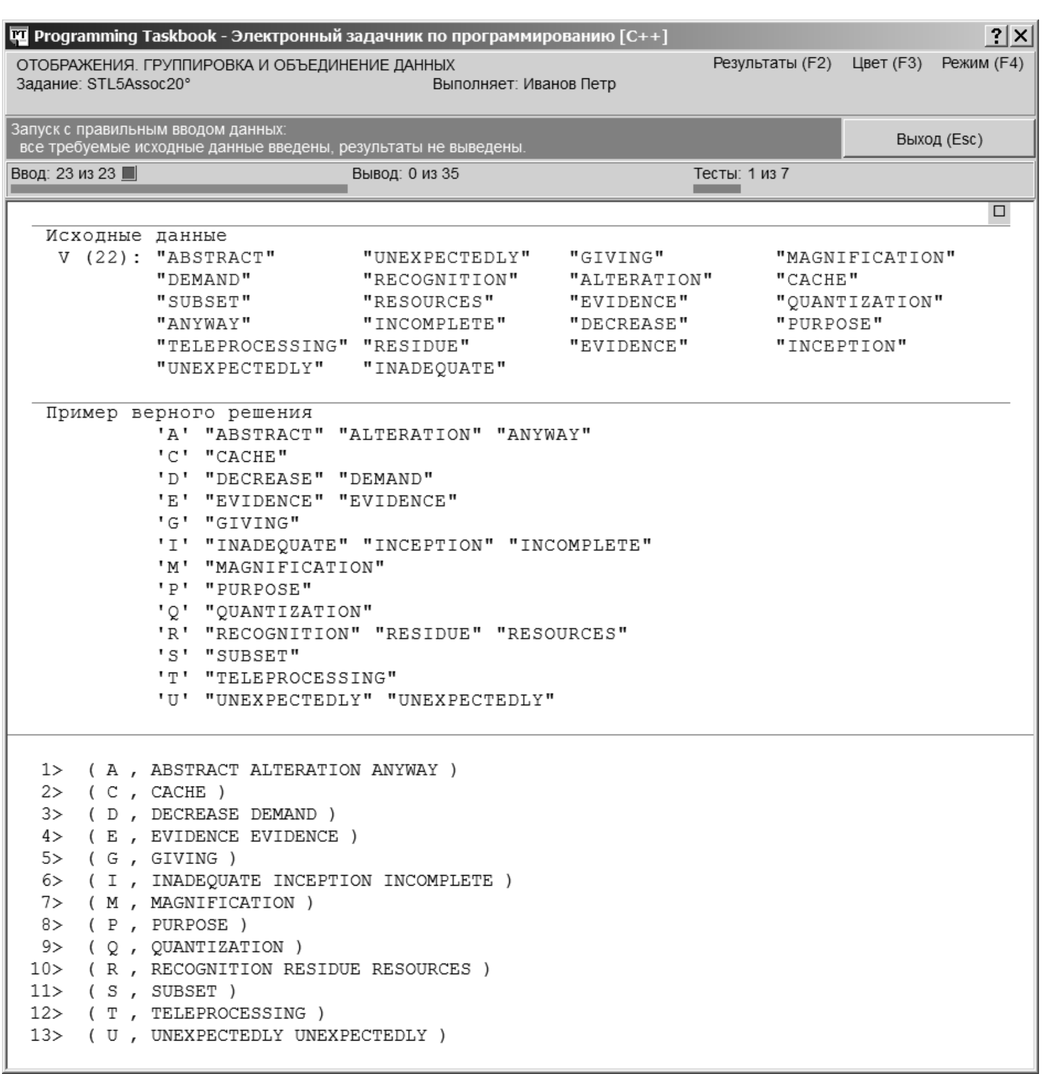


**Рис. 19**. Окно задачника на начальном этапе выполнения задания STL5Assoc20

В разделе отладки выводится содержимое контейнера M. Обратите внимание на то, что функции Show и ShowLine позволяют выводить не только скалярные данные (числа, символы и строки), но и *пары*, т. е. данные типа pair<T1, T2>; при этом перед первым компонентом пары выводится открывающая скобка «(», между первым и вторым компонентом выводится запятая, а после второго компонента – закрывающая скобка «)». Кроме то-

го, функции Show и ShowLine могут выводить данные (в частности, компоненты пары), которые сами представляют собой пары или контейнеры. В нашем случае таким контейнером (типа multiset) является второй компонент пары. Таким образом, функции Show и ShowLine можно использовать для наглядного вывода любых вариантов контейнеров из заданий группы STL5Assoc.

Сравнивая пример правильного решения и содержимым раздела отладки, мы убеждаемся, что группировка выполнена требуемым образом. Осталось вывести содержимое полученного контейнера M в нужном порядке. Для этого мы также используем алгоритм for_each, предварительно описав вспомогательную функцию f2. Приведем окончательный вариант решения, выделив в нем добавленные фрагменты полужирным шрифтом и удалив вызов функции ShowLine:

```
map<char, multiset<string>> M;

void f1(string s)
{
    M[s[0]].insert(s);
}

void f2(pair<char, multiset<string>> p)
{
    pt << p.first;
    copy(p.second.begin(), p.second.end(),
        ptout_iterator<string>());
}

void Solve()
{
    Task("STL5Assoc20");
    vector<string> V(ptin(0), ptin());
    for_each(V.begin(), V.end(), f1);
    for_each(M.begin(), M.end(), f2);
}
```

Для вывода ключа группировки (т. е. первого компонента пары p) мы использовали операцию << для потока pt, а для вывода всех элементов соответствующей группы строк (т. е. второго компонента пары p) мы применили алгоритм copy с итератором записи `ptout_iterator<string>` (напомним, что именно таким образом выводились последовательности в ранее рассмотренных заданиях групп STL2Seq и STL3Alg).

**Примечание**. Чтобы повысить эффективность функции f2, ее заголовок следует оформить следующим образом (добавленные модификаторы выделены полужирным шрифтом):

```
void f2(const pair<char, multiset<string>> &p)
```

После запуска программы и ее автоматической проверки на пяти наборах тестовых данных будет выведено сообщение о том, что задание выполнено.

Завершая обсуждение этого задания, приведем вариант решения, использующий нововведение языка C++, появившееся в стандарте C++11: *цикл for по элементам контейнера*. В этом варианте весь текст решения, включая описание отображения M, размещается в функции Solve (полужирным шрифтом выделены операторы, добавленные к тексту заготовки):

```
void Solve()
{
    Task("STL5Assoc20");
    vector<string> V(ptin(0), ptin());
    map<char, multiset<string>> M;
    for (auto s : V)
        M[s[0]].insert(s);
    for (auto p : M)
    {
        pt << p.first;
        copy(p.second.begin(), p.second.end(),
            ptout_iterator<string>());
    }
}
```

Удобным нововведением языка C++, используемым в заголовке цикла for по элементам контейнера, является также описатель auto, означающий, что тип параметра цикла должен выводиться из типа контейнера. В новом варианте решения мы сохранили имена переменных s и p, напоминающие о том, что элементами исходного вектора V являются *строки*, а элементами отображения M являются *пары*.

**Примечание**. Для повышения эффективности циклов их переменные желательно описать следующим образом (приведем вариант для первого цикла):

```
for (const auto &s : V)
```

Заметим, что если в последнем варианте цикла убрать модификатор const и оставить модификатор &, то цикл можно будет использовать и для *изменения* элементов контейнера V.

### 2.4.2. Дополнительные виды функциональных объектов: STL6Func9

*Функциональные объекты* используются во многих стандартных алгоритмах и, в силу этого, являются неотъемлемой частью библиотеки STL.

При выполнении заданий из предыдущих групп, если они требовали применения функциональных объектов, предлагалось использовать лямбда-выражения или (если компилятор не поддерживает стандарт C++11) обычные функции. Иногда требовалось применять стандартные функциональные объекты less и greater (см., например, STL1Iter21, STL3Alg38, STL5Assoc5). Однако библиотека STL включает и другие стандартные функциональные объекты, описанные в заголовочном файле <functional>. Хотя с появлением в языке C++ лямбда-выражений многие стандартные средства STL, связанные с функциональными объектами, стали существенно менее востребованными, при изучении STL следует познакомиться и с этими средствами (хотя бы для того, чтобы понимать ранее написанный программный код). Знакомству с дополнительными возможностями функциональных объектов посвящена небольшая группа STL6Func, содержащая 14 заданий. Рассмотрим одно из заданий этой группы.

STL6Func9. Определить структуру point с целочисленными членами $x$, $y$ и строковым членом $s$. Предполагается, что строковый член $s$ не содержит пробелов. Отношение порядка для данной структуры определяется следующим образом: $A < B$, если $A.x < B.x$ или ($A.x == B.x$ и $A.y < B.y$). Реализовать это отношение порядка в виде константной функции-члена `bool operator<(const point& b) const`. Кроме того, реализовать операцию `istream& operator>>(istream& is, point& p)`, которая последовательно считывает из входного потока *is* члены $x$, $y$ и $s$ структуры $p$, и операцию `ostream& operator<<(ostream& os, const point& p)`, которая записывает все члены структуры $p$ в выходной поток *os* (члены записываются в том же порядке, в котором считываются; между ними вставляется пробел). Дан текстовый файл с именем *name*, содержащий текстовые представления элементов описанной выше структуры. Используя итератор чтения istream_iterator, заполнить этими данными вектор $V$ с элементами типа point. Используя алгоритм stable_sort, отсортировать полученные данные с учетом заданного отношения порядка и с помощью алгоритма copy и итератора ostream_iterator записать отсортированный вектор в исходный файл, заменив его прежнее содержимое (при этом точки с одинаковыми координатами будут располагаться в том же порядке, что и в исходном файле, поскольку алгоритм stable_sort обеспечивает устойчивую сортировку). Каждая точка должна отображаться на новой строке.

Задание STL6Func9 начинает серию из 6 заданий, в которой в качестве функциональных объектов используются функции-члены специальной

структуры point. Кроме того, в заданиях STL6Func9 и STL6Func10 требуется определить для структуры point дополнительные операции, упрощающие ее использование совместно с итераторами чтения и записи. Заметим, что рассмотренные в STL6Func9 и STL6Func10 возможности будут в дальнейшем активно использоваться при выполнении заданий группы STL7Mix.

Заготовки для заданий группы STL6Func, подобно заготовкам заданий группы STL1Iter, являются очень краткими:

```
#include "pt4.h"
using namespace std;

void Solve()
{
    Task("STL6Func9");
}
```

Вид окна задачника при запуске данной заготовки приведен на рис. 20 (на рисунке не показан раздел отладки с примечаниями к заданию).

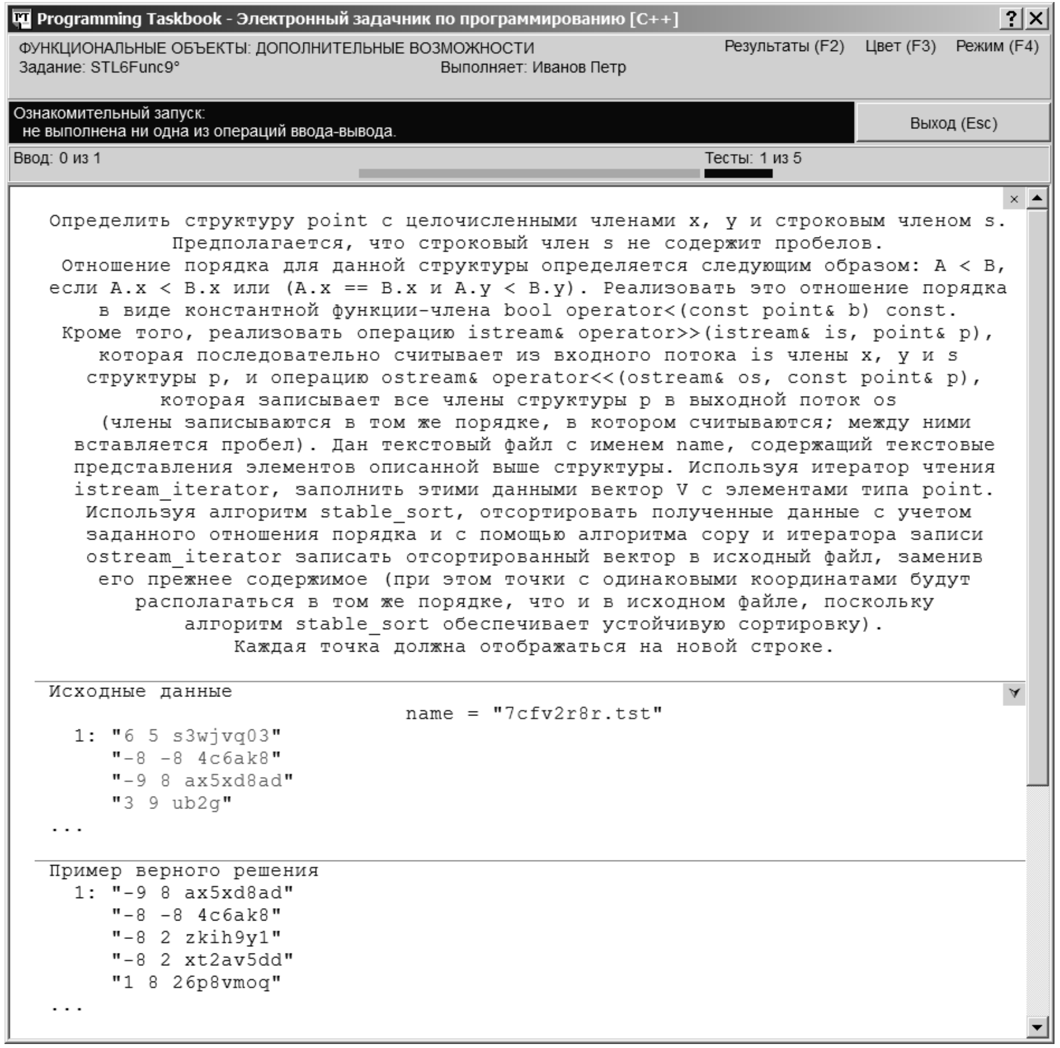


**Рис. 20**. Окно задачника при запуске программы-заготовки для задания STL6Func9

Обратите внимание на то, как в окне задачника отображаются файловые данные. По умолчанию выводятся несколько начальных строк файлов, после которых указывается многоточие «…». Как уже было отмечено при обсуждении задачи STL1Iter17 (см. примечание 2 в п. 2.1.1), для отображения всех файловых данных достаточно выполнить двойной щелчок на одном из разделов окна задачника, содержащих файловые данные (можно также нажать клавишу [Ins] или щелкнуть мышью на маркере , расположенном в правом верхнем углу раздела исходных данных). Более подробно средства просмотра файловых данных описываются в п. 2.5.1.

Приступим к решению задачи. На первом этапе опишем структуру point, определим для нее операцию >> и организуем с ее помощью заполнение вектора данными из исходного файла. Для проверки правильности чтения исходных данных выведем их в разделе отладки окна задачника. Кроме того, удалим из файла его содержимое, чтобы оно не рассматривалось задачником как результат обработки исходного набора данных. Получаем следующую программу (в ней не указаны начальные директивы, входящие в заготовку):

```
#include <fstream>
#include <vector>

struct point
{
    int x;
    int y;
    string s;
};

istream& operator>>(istream& is, point& p)
{
    is >> p.x >> p.y >> p.s;
    return is;
}

void Solve()
{
    Task("STL6Func9");
    string name;
    pt >> name;
    ifstream f1(name);
    vector<point> V((istream_iterator<point>(f1)),
        istream_iterator<point>());
```

```
        f1.close();
        for (auto e : V)
        {
            Show(e.x);
            Show(e.y);
            ShowLine(e.s);
        }
        ofstream f2;
        f2.open(name);
        f2.close();
    }
```

Первый параметр конструктора вектора V взят в дополнительные скобки для того, чтобы оператор описания вектора V не интерпретировался компилятором как объявление прототипа функции V с двумя параметрами – указателями на функции (см. по этому поводу указание к заданию STL2Seq1 в п. 3.2.1).

Мы использовали в программе две возможности, появившиеся в стандарте C++11. Во-первых, в новом стандарте можно указывать параметр типа string в конструкторе потоков и в функции-члене open. Во-вторых, для вывода элементов контейнера в разделе отладки мы применили цикл for по элементам контейнера (ранее подобным циклом мы воспользовались в последнем варианте решения задачи STL5Assoc20, приведенном в конце предыдущего пункта). В старых версиях C++ (в частности, при работе в Visual Studio 2008) данные типа string при использовании в качестве имен файлов необходимо преобразовывать к типу char* с помощью функции-члена c_str, а вместо цикла по элементам контейнера можно применить алгоритм for_each с функциональным объектом (см. первый вариант решения задачи STL5Assoc20 в п. 2.4.1).

**Примечание**. Отладочный вывод контейнеров, содержащих пользовательский тип данных, можно существенно упростить, если определить для этого типа *оператор преобразования к типу* string (см. задание STL6Func10). Если для типа point определен подобный оператор преобразования, то вывод в раздел отладки всех элементов вектора V типа vector<point> (с отображением каждого элемента на отдельной строке) можно выполнить с помощью *единственного* вызова функции ShowLine с двумя параметрами-итераторами: ShowLine(V.begin(), V.end());

При запуске полученной программы на экране появится окно, примерный вид которого приведен на рис. 21. Чтобы уменьшить размеры окна, мы скрыли раздел с формулировкой задания.

Сравнивая содержимое раздела исходных данных и раздела отладки, легко убедиться, что вектор V заполнен правильно. Благодаря очистке со-

держимого исходного файла задачник не стал проверять правильность полученного результата (в тексте на информационной панели лишь отмечено, что результирующий файл является пустым).

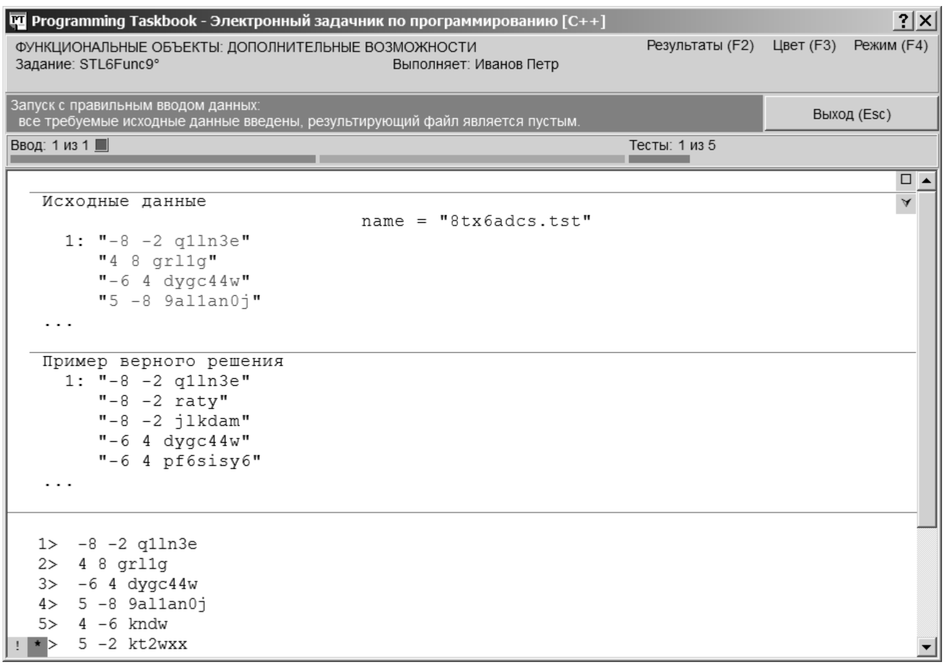


**Рис. 21**. Окно задачника на первом этапе выполнения задания STL6Func9

Обратите также внимание на маркеры ! и *, появившиеся в левом нижнем углу окна. Они позволяют переключаться между несколькими страницами раздела отладки. По умолчанию в разделе отладки отображаются данные, выведенные с помощью функций Show и ShowLine; эта страница раздела отладки связывается с маркером *, который отображается на экране только в случае, если раздел отладки содержит еще одну непустую страницу (связываемую с маркером !) – страницу *дополнительных указаний* или *примечаний* к выполняемому заданию. Если задание содержит подобные элементы, то для их просмотра достаточно либо ввести комбинацию [Shift]+[1] (соответствующую символу «!»), либо нажать клавишу со стрелкой влево или вправо, либо щелкнуть мышью на маркере !. В результате в разделе отладки будет выведено содержимое страницы с дополнительными указаниями или примечаниями (рис. 22).

Вернемся к решению нашей задачи. На следующем этапе нам необходимо выполнить сортировку исходного набора точек в соответствии с порядком, определенным в условии. Для сортировки будем использовать алгоритм stable_sort. В этом алгоритме можно явно указать лямбда-выражение или функцию, которые определяют порядок сортировки. Однако более предпочтительным в нашем случае является определение отношения порядка непосредственно в структуре point в виде операции <. При наличии у типа оперции < именно она используется по умолчанию во всех алгоритмах, связанных с отношением порядка.

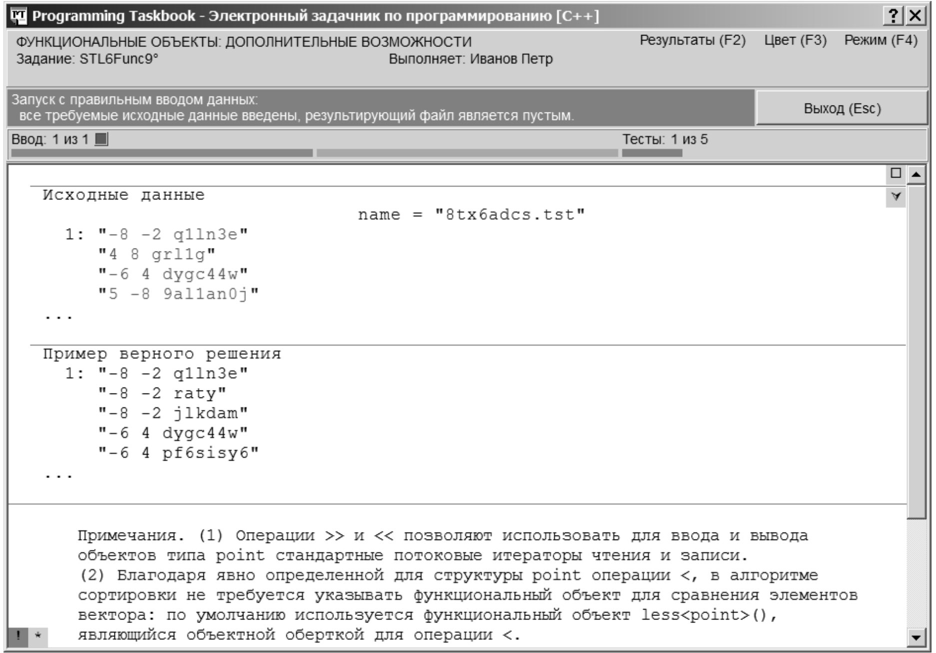


**Рис. 22**. Окно задачника с текстом дополнительных примечаний в разделе отладки

Итак, добавим в программу описание операции < для структуры point, а в функции Solve после заполнения вектора V вызовем для него алгоритм stable_sort, не указывая в нем параметр – функциональный объект. В приведенном ниже тексте программы добавленные фрагменты выделены полужирным шрифтом:

```
#include <fstream>
#include <vector>
#include <algorithm>

struct point
{
    int x;
    int y;
    string s;
};

istream& operator>>(istream& is, point& p)
{
    is >> p.x >> p.y >> p.s;
    return is;
}

bool operator<(const point& p1, const point& p2)
{
    return p1.x < p2.x || p1.x == p2.x && p1.y < p2.y;
}
```

```
void Solve()
{
    Task("STL6Func9");
    string name;
    ifstream f1;
    pt >> name;
    f1.open(name);
    vector<point> V((istream_iterator<point>(f1)),
        istream_iterator<point>());
    f1.close();
    stable_sort(V.begin(), V.end());
    for (auto e : V)
    {
        Show(e.x);
        Show(e.y);
        ShowLine(e.s);
    }
    ofstream f2;
    f2.open(name);
    f2.close();
}
```

Обратите внимание на подключение заголовочного файла <algorithm>, а также на то, что параметры операции < описаны как константные (если модификатор const отсутствует, то описание операции < откомпилируется успешно, однако возникнет ошибка компиляции при вызове алгоритма stable_sort).

При запуске полученной программы мы можем убедиться, что теперь содержимое раздела отладки совпадает с содержимым раздела результатов (рис. 23).

Нам осталось записать преобразованный вектор в файл, предварительно определив операцию << для структуры point. Приведем описание этой операции и окончательный вариант функции Solve, удалив из него фрагмент, обеспечивающий отладочную печать:

```
ostream& operator<<(ostream& os, const point& p)
{
    os << p.x << " " << p.y << " " << p.s;
    return os;
}

void Solve()
{
```

```
    Task("STL6Func9");
    string name;
    ifstream f1;
    pt >> name;
    f1.open(name);
    vector<point> V((istream_iterator<point>(f1)),
        istream_iterator<point>());
    f1.close();
    stable_sort(V.begin(), V.end());
    ofstream f2;
    f2.open(name);
    copy(V.begin(), V.end(), ostream_iterator<point>(f2, "\n"));
    f2.close();
}
```

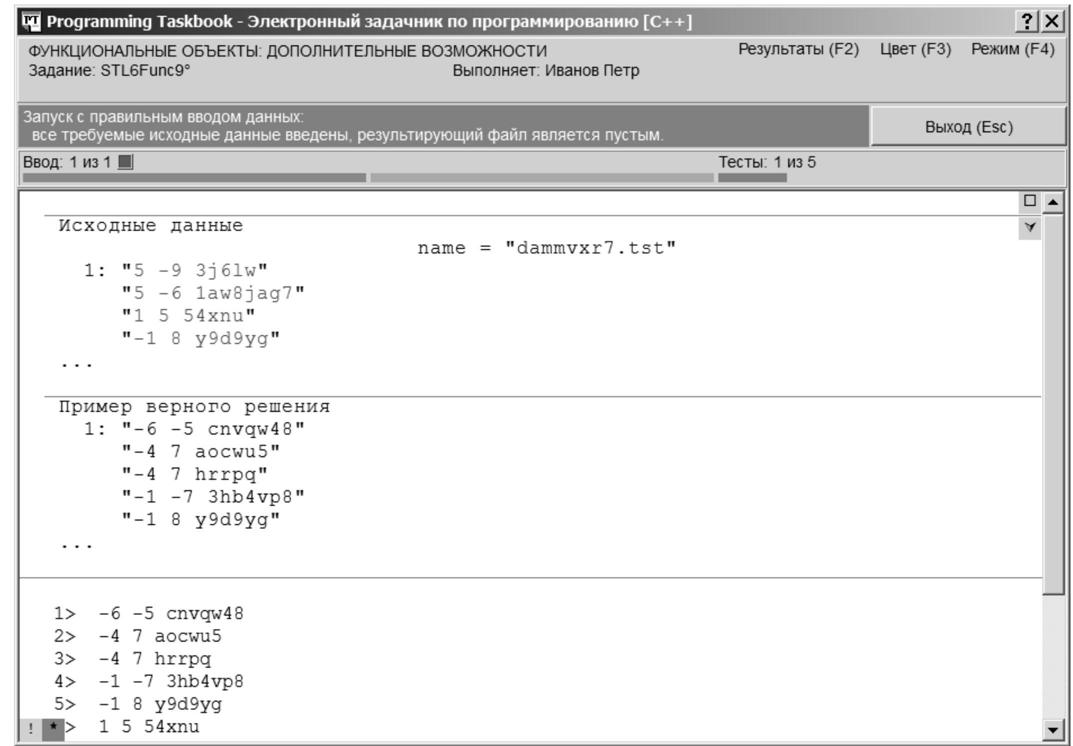


**Рис. 23**. Окно задачника на втором этапе выполнения задания STL6Func9

Для записи элементов вектора в файл мы, как обычно, использовали алгоритм copy с потоковым итератором записи.

После запуска программы и ее автоматической проверки на пяти наборах тестовых данных мы увидим в окне задачника сообщение о том, что задание выполнено.

## *2.5. Применение различных средств стандартной библиотеки C++: STL7Mix4*

### 2.5.1. Создание проекта-заготовки и знакомство с заданием. Дополнительные средства окна задачника, связанные с просмотром файловых данных

Группа STL7Mix является завершающей группой, входящей в задачник Programming Taskbook for STL. Каждая из предыдущих групп задачника была посвящена отдельной части стандартной библиотеки: итераторам, последовательным контейнерам, алгоритмам, строкам, ассоциативным контейнерам, стандартным функциональным объектам. Задания группы STL7Mix предназначены для закрепления навыков *совместного использования* различных компонентов библиотеки STL; при их выполнении требуется самостоятельно выбрать средства библиотеки, обеспечивающие получение требуемого результата. Еще одним отличием заданий последней группы является более сложный вид исходных последовательностей: их элементы представляют собой записи, состоящие из нескольких полей (ранее с подобным видом исходных данных мы встречались в завершающих заданиях группы STL6Func – см. п. 2.4.2). Указанные особенности приближают задания группы STL7Mix к реальным задачам, возникающим при обработке сложных структур данных.

Набор заданий, входящих в группу STL7Mix, ранее был использован в задачнике Programming Taskbook for LINQ [1] в качестве группы LinqObj. Задачник Programming Taskbook for LINQ предназначен для изучения *технологии LINQ*, доступной для языков платформы .NET (C#, VB.NET, PascalABC.NET) и призванной упростить решение тех же задач, для которых была разработана библиотека STL, а именно задач, связанных с *обработкой последовательностей*. Благодаря совпадающим формулировкам задач групп STL7Mix и LinqObj их можно использовать для сравнительного изучения на практике обеих технологий, что позволит глубже понять их основные идеи и особенности. В частности, можно сравнить способы решения задачи STL7Mix4 и LinqObj4, приведенные в справочных системах к этим задачникам. Решения ряда задач из группы LinqObj приведены также в книге [1].

В качестве примера рассмотрим сравнительно простую задачу STL7Mix4, связанную с обработкой отдельной последовательности.

STL7Mix4. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Год> <Номер месяца> <Продолжительность занятий (в часах)> <Код клиента>

Для каждого клиента, присутствующего в исходных данных, определить суммарную продолжительность занятий в течение всех лет (вначале выводить суммарную продолжительность, затем код клиента). Сведения о каждом клиенте выводить на новой строке и упорядочивать по убыванию суммарной продолжительности, а при их равенстве – по возрастанию кода клиента.

Указание. Для группировки по полю «код» используйте вспомогательное отображение.

После создания с помощью модуля PT4Load проекта-заготовки для данной задачи и его загрузки в выбранную среду программирования на экране будет отображен файл STL7Mix4.cpp, в который требуется ввести текст решения. Приведем содержимое этого файла:

```
#include "pt4.h"
using namespace std;

#include <fstream>
#include <sstream>
#include <iomanip>
#include <algorithm>
#include <vector>
#include <set>
#include <map>
struct Data
{
    int code, year, month, len;
    operator string()
    {
        ostringstream os;
        os << "{ code = " << code << ", year = " << year
           << ", month = " << month << ", len = " << len << " }";
        return os.str();
    }
};

istream& operator>>(istream& is, Data& p)
{
    return is >> p.year >> p.month >> p.len >> p.code;
}

void Solve()
```

```
{
    Task("STL7Mix4");
    string name1, name2;
    pt >> name1 >> name2;
    ifstream f1(name1.c_str());
    vector<Data> V((istream_iterator<Data>(f1)),
        istream_iterator<Data>());
    f1.close();
    ShowLine(V.begin(), V.end(), "V: ");

    ofstream f2(name2.c_str());

    f2.close();
}
```

Тексты заготовок для заданий группы STL7Mix являются еще более развернутыми, чем заготовки для предыдущих групп задачника PT for STL. Напомним, что в группе STL1Iter использовались простейшие заготовки, содержащие, кроме минимального количества начальных директив, лишь описание функции Solve с вызовом функции Task. В последующих группах заготовки включали описание используемых контейнеров и даже ввод в них исходных данных. Это избавляло студента от стандартных начальных действий (уже отработанных при выполнении предыдущих заданий) и позволяло сразу сосредоточиться на смысловой части алгоритма.

Подобные цели преследуют и развернутые заготовки группы STL7Mix. Они не просто обеспечивают ввод исходных данных (хранящихся во внешних файлах), но и позволяют сохранить эти данные в векторе, причем каждый элемент данных представляется в виде структуры Data, также описанной в заготовке. Помимо полей в этой структуре определен оператор преобразования структуры в строку, что дает возможность использовать шаблонный вариант функции ShowLine для быстрого вывода полученной последовательности структур в разделе отладки окна задачника (причем вызов этого варианта функции ShowLine уже присутствует в заготовке). Для организации ввода исходных данных с применением итераторов файловых потоков в программе переопределяется операция >> для структуры Data (следует обратить внимание на конструктор вектора V, содержащий два параметра-итератора, первый из которых заключен в дополнительные круглые скобки; причина этого объясняется в указании к заданию STL2Seq1). Наконец, в состав заготовки включены и все операторы, обеспечивающие открытие и закрытие используемых текстовых файлов.

**Примечание**. Как уже отмечалось в п. 2.4.2, в стандарте C++11 появилась возможность использования параметров типа string в конструкторах

потоков; таким образом, в созданной заготовке можно было обойтись без вызова функции c_str. Вызов данной функции оставлен для того, чтобы обеспечить возможность компиляции заготовки в Visual Studio 2008.

Заметим, что с возможностями, использованными в заготовках к заданиям группы STL7Mix, мы уже познакомились при выполнении заданий группы STL6Func (см., в частности, описание процесса выполнения задания STL6Func9 в п. 2.4.2).

Благодаря наличию развернутой заготовки мы можем сразу приступить к реализации содержательной части программы: обработке исходной последовательности (уже доступной в виде вектора типа vector<Data>) с целью получения результирующей последовательности и ее вывода в текстовый файл в нужном формате.

После запуска созданной программы-заготовки мы увидим на экране окно задачника, содержащее формулировку задания, пример исходных данных и правильных результатов, а также строковые представления всех элементов исходного набора, выведенные функцией ShowLine в разделе отладки (рис. 24).

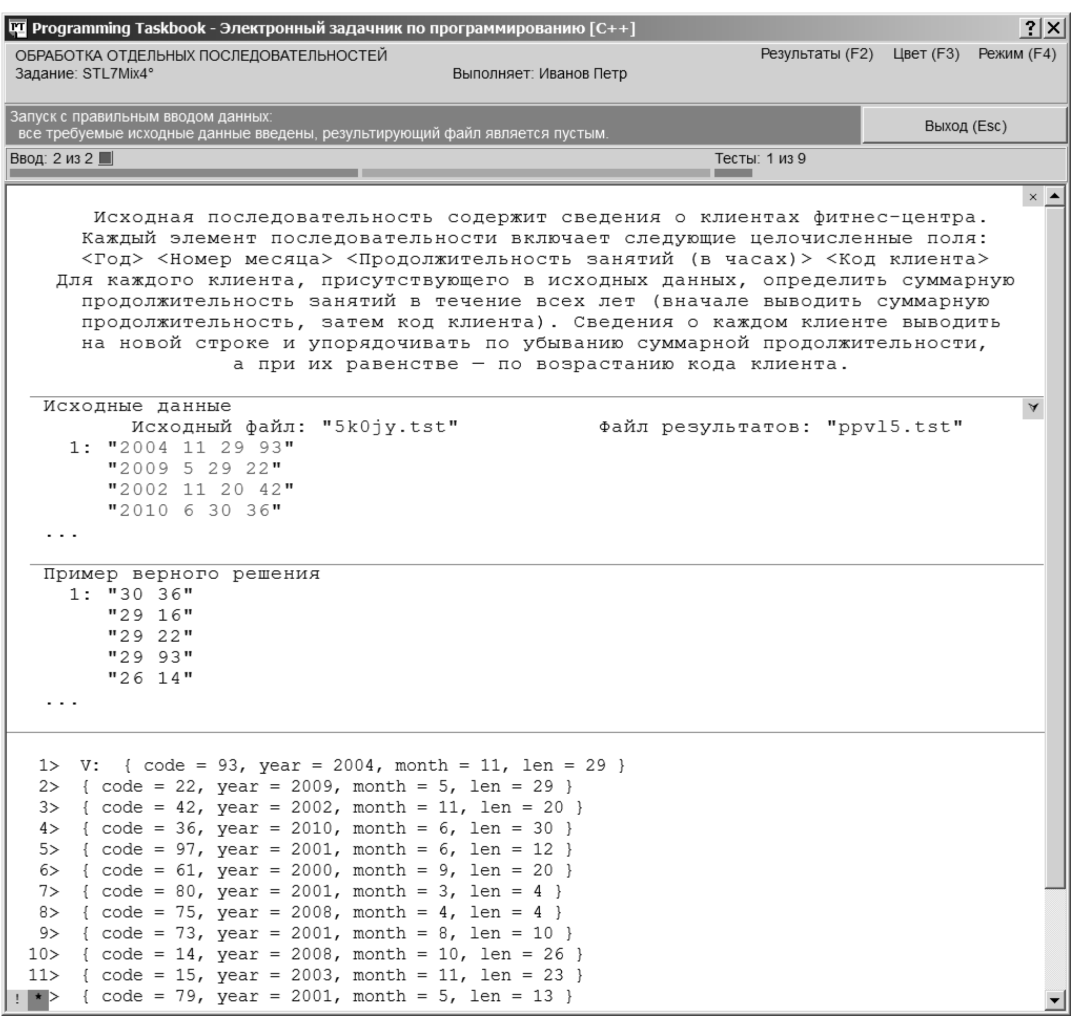


**Рис. 24**. Окно задачника при запуске программы-заготовки для задания STL7Mix4

На информационной панели (светло-синего цвета) отмечается, что все исходные данные введены, а файл результатов является пустым. Таким образом, все готово для реализации алгоритма обработки исходных данных.

Однако прежде, чем переходить к решению задачи, опишем дополнительные возможности окна задачника, которые могут оказаться полезными при анализе исходных и результирующих данных, представленных в виде текстовых файлов большого размера.

Приведенный на рис. 24 вариант окна соответствует режиму *сокращенного отображения файловых данных*, в котором для каждого файла отображается лишь начальная часть его содержимого (от одной до пяти строк). Признаком того, что часть данных отсутствует, является многоточие, размещенное в нижней части тех разделов, в которых отображаются сокращенные данные. Этот режим удобен при первоначальном знакомстве с заданием, поскольку позволяет отобразить в окне сравнительно небольшого размера содержимое всех разделов. При более детальном анализе данных, а также при сравнении полученных ошибочных результатов с примером правильного решения следует использовать режим *полного отображения текстовых файлов*.

Для переключения между сокращенным и полным режимом отображения файловых данных достаточно нажать клавишу [Ins] или выполнить двойной щелчок мышью в одном из разделов с файловыми данными. Можно также щелкнуть на квадратном маркере, который появляется в правом верхнем углу раздела исходных данных, если окно содержит файловые данные. Изображение на этом маркере служит индикатором режима: вариант со стрелкой, направленной вниз, обозначает режим сокращенного отображения (при наведении мыши на маркер в этом случае выводится подсказка «*Развернуть содержимое текстовых файлов (Ins)*»); вариант со стрелкой, направленной вверх, обозначает режим полного отображения (с ним связана подсказка «*Свернуть содержимое текстовых файлов (Ins)*»).

На рис. 25 приведен вид окна в режиме полного отображения текстовых файлов. В этом режиме порядковый номер указывается перед каждой файловой строкой. При закрытии окна текущий режим отображения запоминается и при последующих запусках программы восстанавливается.

В случае, когда размеров окна недостаточно для отображения всех данных, окно снабжается полосой прокрутки, и, кроме того, в нем отображаются дополнительные маркеры (все эти дополнительные элементы окна имеются на рис. 25). Прокрутку содержимого окна проще всего выполнять с помощью клавиш [Home], [End], [Up], [Down], [PgUp], [PgDn], а также используя колесико мыши.

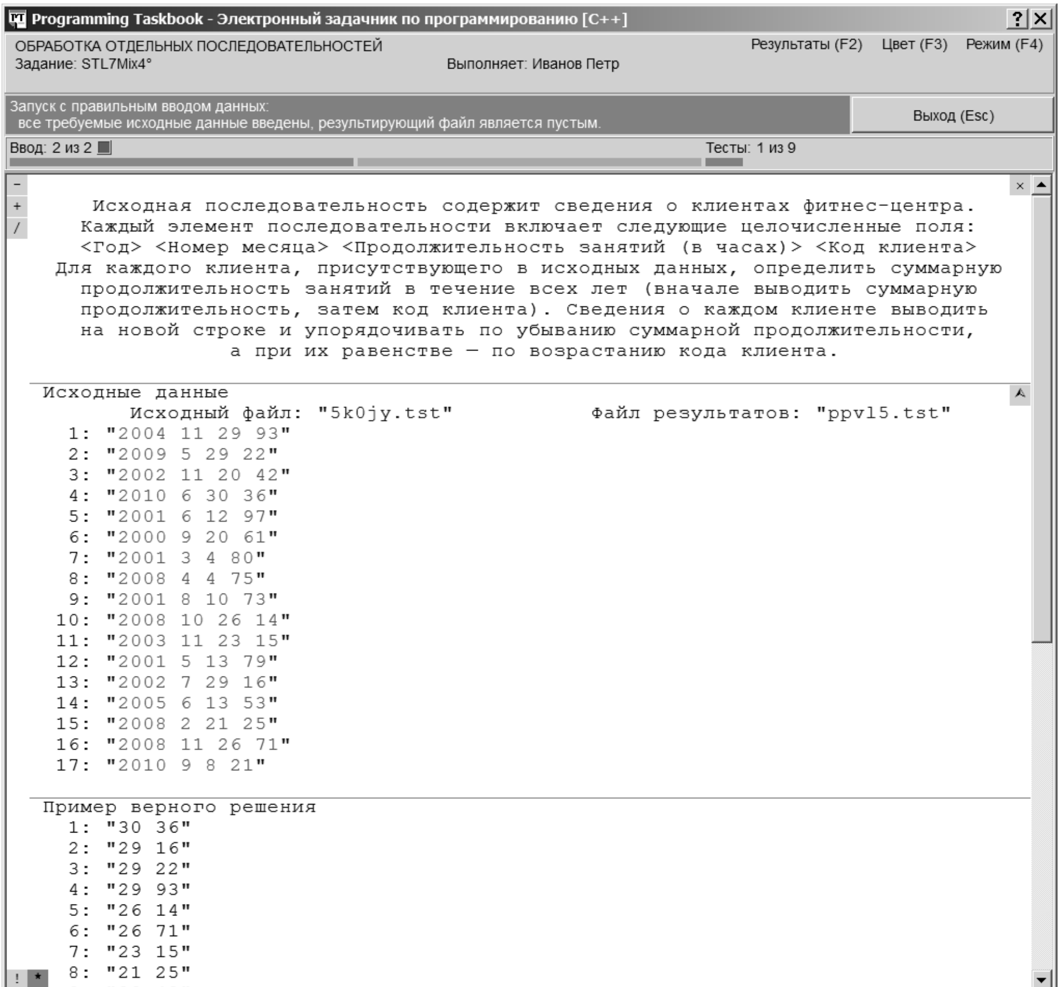


**Рис. 25**. Окно задачника в режиме полного отображения текстовых файлов

Группа маркеров в левом верхнем углу раздела с формулировкой задания предназначена для быстрого отображения различных разделов, связанных с заданием: щелчок на маркере + обеспечивает переход к началу следующего раздела, щелчок на маркере – – к началу предыдущего раздела. Перебор разделов выполняется циклически. Маркер / позволяет переключаться между разделами с результатами и с примером правильного решения, если окно содержит оба этих раздела. Вместо щелчка на указанных маркерах достаточно нажать соответствующую клавишу: [+], [–] или [/].

Назначение маркера × было описано в п. 2.4.1, а маркеров ! и * – в п. 2.4.2: первый из них позволяет скрыть раздел с формулировкой задания, а два последних обеспечивают переключение между страницами раздела отладки; при этом маркер * связан со страницей, на которой отображаются отладочные данные, а маркер ! – со страницей, содержащей дополнительные указания или примечания к заданию. На рис. 26 приводится пример окна в режиме сокращенного отображения файловых данных с дополнительным указанием для задания STL7Mix4.

**Рис. 26**. Окно задачника с текстом указания в разделе отладки

## 2.5.2. Выполнение задания

Завершив обзор дополнительных возможностей окна задачника и ознакомившись с заданием STL7Mix4, приступим к его выполнению.

В процессе обработки исходной последовательности (которая уже хранится в векторе V типа vector<Data>) мы должны, во-первых, выполнить *группировку* исходных данных по полю code (код клиента), определив для каждого клиента суммарную продолжительность его занятий, и, во-вторых, *отсортировать* данные, полученные в результате группировки, по набору ключей: главный ключ – суммарная длительность (сортируется по убыванию), подчиненный ключ – код клиента (сортируется по возрастанию для одинаковых значений главного ключа).

Способы группировки данных, основанные на контейнерах STL, подробно анализировались в заданиях группы STL5Assoc (см. п. 3.5.2, а также решение задачи STL5Assoc20 в п. 2.4.1). Один из эффективных способов группировки основан на использовании *отображения* (map); именно об этом способе говорится в указании к нашему заданию (рис. 26).

Определим отображение M с ключом – кодом клиента (целого типа) и значением – суммарной длительностью занятий (также целого типа):

```
map<int, int> M;
```

Для заполнения этого отображения мы должны перебрать все элементы вектора V и для каждого элемента e выполнить следующий оператор:

```
M[e.code] += e.len;
```

В данном операторе используется следующая особенность операции индексирования для отображения (см. п. 1.2.6): если элемент отображения с указанным ключом не существует, то он автоматически создается, и его значению присваивается значение по умолчанию для данного типа (в нашем случае – число 0).

Перебор элементов вектора V можно выполнить различными способами. Опишем два наиболее коротких и наглядных.

Первый из этих способов основан на применении алгоритма for_each с лямбда-выражением в качестве функционального объекта:

```
for_each(V.begin(), V.end(),
    [&M](Data e){ M[e.code] += e.len; });
```

Следует обратить внимание на первый компонент лямбда-выражения: [&M]. Он означает, что данное лямбда-выражение *захватывает* внешнюю переменную M, причем захват выполняется *по ссылке*, что позволяет изменять эту переменную внутри лямбда-выражения.

Во втором способе используется специальный вариант цикла for, предназначенный для перебора элементов некоторого контейнера (ранее этот вариант цикла использовался в п. 2.4 при решении задач STL5Assoc20 и STL6Func9):

```
for (auto e : V)
    M[e.code] += e.len;
```

Вместо описателя auto, означающего, что компилятор должен сам вывести тип параметра цикла e, исходя из типа элементов контейнера V, можно указать явный описатель Data, однако вариант с auto является более предпочтительным в силу своей универсальности.

После выполнения группировки желательно ознакомиться с ее результатами. Проще всего вывести полученные данные в раздел отладки. Чтобы не загромождать этот раздел лишними сведениями, удалим вызов функции ShowLine, добавленный в программу-заготовку при ее создании. Для вывода в раздел отладки элементов отображения M достаточно использовать аналогичный вызов функции ShowLine с параметрами-итераторами:

```
ShowLine(M.begin(), M.end());
```

Приведем новый вариант функции Solve, выделив в нем полужирным шрифтом добавленные операторы:

```
void Solve()
{
```

```
    Task("STL7Mix4");
    string name1, name2;
    pt >> name1 >> name2;
    ifstream f1(name1.c_str());
    vector<Data> V((istream_iterator<Data>(f1)),
        istream_iterator<Data>());
    f1.close();
    map<int, int> M;
    for (auto e : V)
        M[e.code] += e.len;
    ShowLine(M.begin(), M.end());
    ofstream f2(name2.c_str());

    f2.close();
}
```

При запуске этого варианта программы содержимое отображения `M` будет выведено в разделе отладки в виде пар чисел (ключ, значение) (рис. 27). Сравнивая эти данные с примером верного решения, легко убедиться, что группировка выполнена правильно (обратите внимание на то, что в разделе с примером верного решения *вначале* указываются значения, а *затем* ключи).

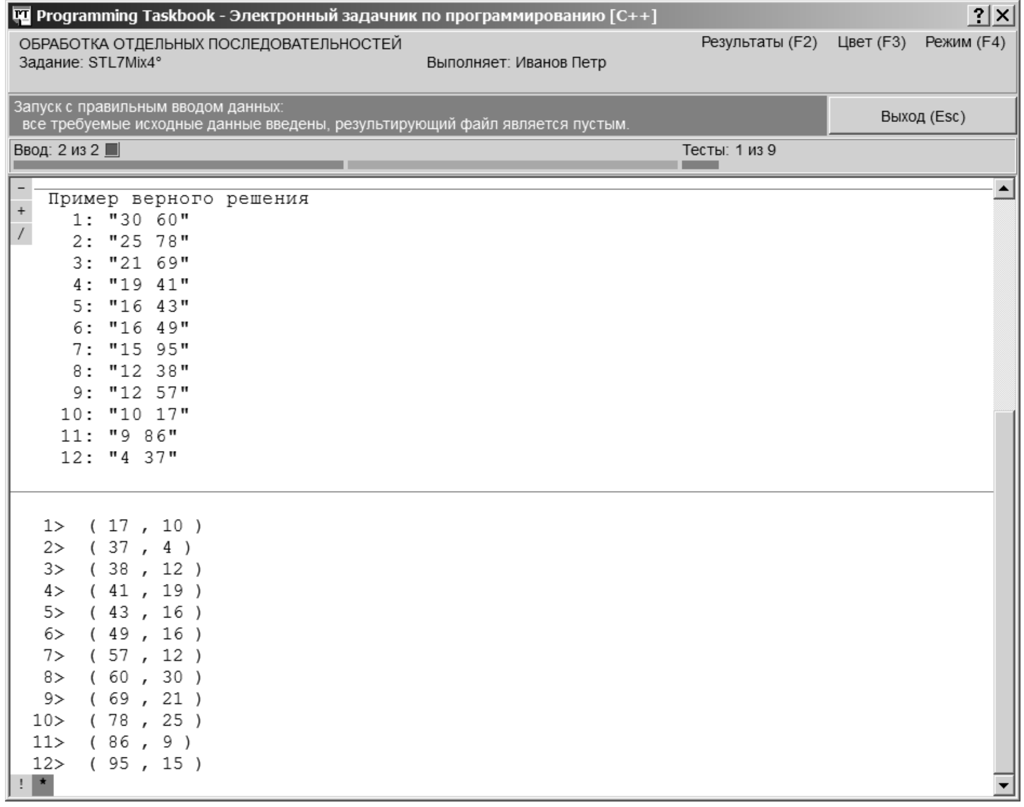


**Рис. 27**. Окно задачника после первого этапа решения задачи STL7Mix4

Итак, мы успешно прошли первый этап решения, выполнив группировку исходных данных. Осталось отсортировать полученные данные в соответствии с условиями задачи.

Так как в контейнере map порядок следования элементов фиксирован и определяется значениями ключей (по умолчанию элементы располагаются по возрастанию ключей – см. рис. 27), для изменения порядка элементов надо скопировать их в контейнер, для которого можно использовать алгоритм сортировки, – например, в новый вектор V1.

Тип элементов вектора V1 должен совпадать с типом элементов отображения M, т. е. pair<int, int> (для этого типа удобно определить короткий псевдоним p); чтобы заполнить вектор данными, достаточно использовать конструктор с двумя параметрами-итераторами:

```
typedef pair<int, int> p;
vector<p> V1(M.begin(), M.end());
```

Для заполненного вектора V1 надо вызвать алгоритм сортировки, передав в него в качестве последнего параметра функциональный объект, определяющий отношение сравнения для пары pair<int, int>. В данном случае, поскольку элементы вектора V1 уже упорядочены по возрастанию ключей (хранящихся в поле first), удобно применить *устойчивый* алгоритм сортировки stable_sort (см. п. 1.3.3):

```
stable_sort(V1.begin(), V1.end(),
    [](p e1, p e2){ return e1.second > e2.second; });
```

Этот вызов алгоритма обеспечивает сортировку элементов по убыванию поля second, причем относительное расположение элементов с *одинаковыми* полями second не изменяется (благодаря устойчивому характеру сортировки). Таким образом, мы получаем такой порядок, который требуется в задаче: элементы располагаются по убыванию поля second (т. е. суммарной длительности), а элементы с одинаковым полем second располагаются по возрастанию поля first (т. е. кода клиента).

Осталось вывести отсортированный вектор V1. Для этого можно использовать либо алгоритм for_each, либо цикл for. Воспользуемся вариантом цикла for, обеспечивающим перебор элементов последовательности (этот цикл необходимо расположить между оператором описания потока f2 и оператором его закрытия f2.close()):

```
for (auto e : V1)
    f2 << e.second << " " << e.first << endl;
```

После запуска данного варианта программы и успешного прохождения требуемых девяти тестов будет выведено сообщение о том, что задание выполнено.

Приведем функцию Solve с одним из вариантов правильного решения, убрав из нее все операторы отладочной печати (полужирным шрифтом выделены операторы, которые не содержались в исходном тексте заготовки):

```
void Solve()
{
    Task("STL7Mix4");
    string name1, name2;
    pt >> name1 >> name2;
    ifstream f1(name1.c_str());
    vector<Data> V((istream_iterator<Data>(f1)),
        istream_iterator<Data>());
    f1.close();

    map<int, int> M;
    for (auto e : V)
        M[e.code] += e.len;
    typedef pair<int, int> p;
    vector<p> V1(M.begin(), M.end());
    stable_sort(V1.begin(), V1.end(),
        [](p e1, p e2){ return e1.second > e2.second; });

    ofstream f2(name2.c_str());
    for (auto e : V1)
        f2 << e.second << " " << e.first << endl;
    f2.close();
}
```

### 2.5.3. Другие варианты решения

Решение было бы более наглядным, если бы поля элементов вектора V1 имели, подобно полям исходного вектора V, значащие имена, например code (код) и sumlen (суммарная длительность). Этого можно добиться, если ввести вспомогательный тип-структуру с указаными полями. Кроме того, в новом типе (назовем его res) можно определить дополнительные функции-члены и операции, которые упростят его использование.

Итак, добавим в программу следующие описания:

```
struct res
{
    int code, sumlen;

    res(pair<int, int> p)
    {
```

```
            code = p.first;
            sumlen = p.second;
        }

        bool operator<(res b) const
        {
            return this->sumlen > b.sumlen;
        }
    };

    ostream& operator<<(ostream& os, const res& r)
    {
        os << r.sumlen << " " << r.code;
        return os;
    }
```

В структуре res, помимо двух полей, определены конструктор, позволяющий неявно преобразовать пару pair<int, int> в элемент типа res, и операция <, позволяющая сравнивать элементы типа res (с учетом условий решаемой задачи мы считаем, что элемент a типа res «меньше» элемента b типа res, если a.sumlen > b.sumlen). Кроме того, мы определили для типа res операцию <<.

Используя тип res и связанные с ним операции, мы можем получить более наглядный вариант решения задачи, избавившись, к тому же, от лямбда-выражения в алгоритме сортировки: если для сортируемых элементов определена операция <, то в алгоритме сортировки можно не указывать функциональный объект – по умолчанию для сравнения элементов будет применяться операция < (см. решение задачи STL6Func9, приведенное в п. 2.4.2):

```
    void Solve()
    {
        Task("STL7Mix4");
        string name1, name2;
        pt >> name1 >> name2;
        ifstream f1(name1.c_str());
        vector<Data> V((istream_iterator<Data>(f1)),
            istream_iterator<Data>());
        f1.close();

        map<int, int> M;
        for (auto e : V)
            M[e.code] += e.len;
```

```
        vector<res> V1(M.begin(), M.end());
        stable_sort(V1.begin(), V1.end());

        ofstream f2(name2.c_str());
        copy(V1.begin(), V1.end(), ostream_iterator<res>(f2, "\n"));
        f2.close();
    }
```

Для возможности заполнения вектора V1 данными, взятыми из отображения M, необходимо, чтобы пару pair<int, int> можно было неявно преобразовать в тип res. Мы обеспечили эту возможность, определив для типа res соответствующий конструктор.

Благодаря наличию операции << для типа res, мы можем использовать алгоритм copy со стандартным потоковым итератором записи для вывода полученных результатов в текстовый файл.

В заключение приведем варианты решения, в которых не применяются новые средства, появившиеся в стандарте C++11. Эти варианты можно использовать, например, при выполнении задания в среде Visual Studio 2008. В первом варианте за основу возьмем программу приведенную в п. 2.5.2. В ней достаточно изменить цикл for, дважды использованный для перебора элементов вектора, и избавиться от лямбда-выражения в алгоритме stable_sort. Вместо лямбда-выражения можно указать обычную функцию, описав ее перед функцией Solve, а в качестве альтернативы цикла можно использовать цикл for с итераторами обрабатываемой последовательности (единственным усложнением будет необходимость явного указания типа параметра-итератора). Получаем следующий вариант решения (полужирным шрифтом выделены фрагменты, не входящие в исходную заготовку):

```
    typedef pair<int, int> p;

    bool comp(p e1, p e2)
    {
        return e1.second > e2.second;
    }

    void Solve()
    {
        Task("STL7Mix4");
        string name1, name2;
        pt >> name1 >> name2;
        ifstream f1(name1.c_str());
        vector<Data> V((istream_iterator<Data>(f1)),
```

```
        istream_iterator<Data>());
    f1.close();

    map<int, int> M;
    for (vector<Data>::iterator i = V.begin(); i != V.end(); ++i)
        M[i->code] += i->len;
    vector<p> V1(M.begin(), M.end());
    stable_sort(V1.begin(), V1.end(), comp);

    ofstream f2(name2.c_str());
    for (vector<p>::iterator i = V1.begin(); i != V1.end(); ++i)
        f2 << i->second << " " << i->first << endl;
    f2.close();
}
```

Вариант решения, использующий вспомогательный тип res, также можно преобразовать к виду, допускающему компиляцию в системе Visual Studio 2008. Лямбда-выражения в нем не применяются, поэтому единственный оператор, требующий замены, – это цикл for для перебора элементов последовательности, который надо заменить на цикл for с итераторами. Для успешной компиляции исправленного варианта необходимо также изменить описание параметра p в заголовке конструктора res:

```
    res(pair<const int, int> p)  // добавлен модификатор const
```

# Раздел 3. Учебные задачи

Задачи, которые приводятся в данном разделе, входят в состав электронного задачника Programming Taskbook for STL (PT for STL). Большинство групп заданий связано с определенным набором объектов стандартной библиотеки шаблонов: итераторами, последовательными или ассоциативными контейнерами, алгоритмами, функциональными объектами. При этом в каждой последующей группе используются объекты, изученные при выполнении заданий из предыдущих групп. Кроме того, предусмотрены группы, предназначенные для повторения ранее изученного материала.

Начальной группой является **STL1Iter** (см. п. 3.1), посвященная знакомству с *итераторами* и *алгоритмами*. Она содержит 24 задания, которые сформулированы таким образом, чтобы для обработки исходных данных не требовалось привлекать контейнеры. В этой группе активно используются стандартные потоковые итераторы чтения и записи istream_iterator и ostream_iterator, а также специальные итераторы ptin_iterator и ptout_iterator, реализованные в электронном задачнике и обладающие всеми свойствами стандартных потоковых итераторов. В дальнейшем итераторы ptin_iterator и ptout_iterator применяются для организации ввода-вывода в большинстве заданий всех групп, входящих в PT for STL. Что касается алгоритмов, то в группе STL1Iter используются только те, которые связаны с базовыми операциями копирования и преобразования, а также с генерацией последовательностей: count, count_if, copy, remove_copy, remove_copy_if, replace_copy, replace_copy_if, transform, merge, fill_n, generate_n.

В некоторых заданиях вместо применения алгоритмов требуется использовать цикл с параметром-итератором. При использовании алгоритмов с функциональными объектами рекомендуется применять *лямбда-выражения*, введенные в стандарте C++11 и доступные во всех средах программирования, поддерживаемых задачником (за исключением Visual Studio 2008). Возможен и традиционный подход, при котором в качестве функциональных объектов указываются предварительно описанные функции. В более сложных ситуациях необходимо применять лямбда-выражения с захваченными переменными или структуры с дополнительными полями и операцией () (см. STL1Iter8 и STL1Iter9).

Следующая группа, имеющая название **STL2Seq** (см. п. 3.2), посвящена стандартным *последовательным контейнерам*: *вектору* (vector), *деку* (deck) и *списку* (list). Она содержит 34 задания и включает три подгруппы, каждая из которых связана с определенным набором операций для контейнеров: в заданиях первой подгруппы требуется заполнять контейнеры и обращаться к их элементам, в заданиях второй – выполнять вставку элементов, а в заданиях третьей – выполнять удаление. Кроме того, уже в пер-

вой подгруппе требуется применять *обратные итераторы* последовательностей, которые в дальнейшем используются во многих заданиях.

Основным объектом изучения в данной группе являются функции-члены контейнеров; алгоритмы в этой группе практически не используются. В ряде заданий особое внимание уделяется различиям в обработке контейнеров, в частности, тому обстоятельству, что для списка (в отличие от вектора и дека) не реализованы итераторы произвольного доступа.

Третья группа **STL3Alg** (см. п. 3.3) целиком посвящена *алгоритмам*. Она содержит 64 задания, в которых уже изученные последовательные контейнеры требуется преобразовать с помощью того или иного стандартного алгоритма. Ввиду большого количества алгоритмов они разбиты на категории, с каждой из которых связывается определенная подгруппа группы STL3Alg. Подгруппы и связанные с ними алгоритмы перечислены в таблице 6.

*Таблица 6*

**Подгруппы группы STL3Alg**

| **Название подгруппы** | **Изучаемые алгоритмы (в порядке появления в задачах подгруппы)** | **Число задач** |
|---|---|---|
| Алгоритмы поиска | find, find_if, find_first_of, search_n, search, adjacent_find | 15 |
| Базовые модифицирующие алгоритмы | fill, fill_n, generate, generate_n, swap_ranges, rotate, rotate_copy, reverse, reverse_copy, replace_if, replace_copy_if, remove_copy_if, remove, remove_if, unique, unique_copy, transform | 20 |
| Сортировка и слияние | nth_element, max_element, min_element, partial_sort, partial_sort_copy, partition, stable_partition, stable_sort, sort, inplace_merge | 13 |
| Перестановки и работа с кучей | next_permutation, prev_permutation, make_heap, pop_heap, push_heap | 6 |
| Численные алгоритмы | accumulate, adjacent_difference, inner_product, partial_sum | 10 |

С учетом алгоритмов, рассмотренных ранее, группа STL3Alg охватывает все категории стандартных алгоритмов, за исключением теоретико-множественных алгоритмов (которые рассматриваются позже, совместно с ассоциативными контейнерами). Наряду с алгоритмами при выполнении заданий требуется применять необходимые функции-члены последовательных контейнеров, а также различные виды итераторов, в том числе *обратные итераторы* (изученные в группе STL2Seq) и *итераторы вставки*, которые вводятся в рассмотрение в подгруппе, связанной с базовыми модифицирующими алгоритмами (см. п. 3.3.2).

Поскольку целью группы STL3Alg является ознакомление со стандартными алгоритмами и типичными ситуациями, в которых их следует применять, в каждом задании этой группы явно указывается, какие средства (алгоритмы, функции-члены, итераторы) необходимо использовать для требуемого преобразования исходных наборов данных. Во всех заданиях требуется преобразовывать последовательности целых чисел.

После выполнения заданий трех первых групп студент уже имеет некоторый опыт в применении основных объектов библиотеки STL. Поэтому следующая группа **STL4Str** (см. п. 3.4) предназначена, прежде всего, для закрепления полученного опыта. Эта группа содержит 28 заданий, связанных с обработкой *текстовых строк* (string). С точки зрения библиотеки STL текстовые строки являются последовательными контейнерами, содержащими символы, поэтому к ним можно применять все средства данной библиотеки, связанные с обработкой последовательных контейнеров. Так как целью заданий группы STL4Str является повторение изученного материала, в их формулировках не уточняется, каким именно способом можно получить требуемый результат: необходимо самостоятельно выбрать оптимальный вариант решения.

Очередная группа **STL5Assoc** (см. п. 3.5) посвящена *ассоциативным контейнерам*: *множествам* (set), *отображениям* (map) и их вариантам, допускающим совпадение элементов множества или ключей отображения (multuset и multimap). Она включает 36 заданий и состоит из двух подгрупп. В первой подгруппе рассматриваются контейнеры set и multiset, а также *теоретико-множественные алгоритмы* (includes, set_intersection, set_difference, set_union, set_symmetric_difference). Во второй рассматриваются контейнеры map и multimap и различные варианты реализации преобразований последовательностей, связанных с их *группировкой* и *объединением*. Особое внимание к подобным преобразованиям объясняется тем, что в терминах этих преобразований можно описать многие задачи на обработку сложных наборов данных, причем именно ассоциативные контейнеры предоставляют средства для их эффективной реализации.

Группа **STL6Func** (см. п. 3.6) содержит 14 заданий, связанных с различными аспектами использования *функциональных объектов* (*объектов-функций*): применение стандартных функциональных объектов и определение новых, работа с *функциональными адаптерами* (в частности, с *инверторами* и *связывателями*), преобразование функций-членов в функциональные объекты. Формулировки заданий ориентированы на применение функциональных объектов, предусмотренных стандартом C++11, хотя допустимо применять и функциональные объекты из предыдущего стандарта; особенности этих «старых» объектов описываются в примечаниях к заданиям.

На этом изучение средств библиотеки STL завершается, и студенту предлагается продемонстрировать полученные знания, выполнив задания из итоговой группы **STL7Mix** (см. п. 3.7). Эта группа состоит из 100 заданий, в каждом из которых требуется обработать структуру данных, состоящую из нескольких числовых и строковых полей и хранящуюся в файле. В завершающей части этой группы, состоящей из 30 заданий, требуется обработать несколько взаимосвязанных наборов данных. Как и в случае группы STL4Str, в формулировках заданий не уточняется, какие средства стандартной библиотеки следует использовать для получения требуемого результата. Однако в большинстве задач требуется выполнять группировку или объединение данных, что предполагает применение ассоциативных контейнеров. Ранее уже отмечалось (см. п. 2.5.1), что формулировки заданий данной группы повторяют формулировки заданий группы LinqObj из электронного задачника по LINQ-технологиям Programming Taskbook for LINQ [1]. Использование одного и того же набора заданий для двух разных программных технологий объясняется тем, что и библиотека STL, и технология LINQ предназначены для решения одинаковых задач, связанных с обработкой наборов данных: STL предоставляет соответствующие средства для языка C++, а LINQ – для языков платформы .NET (C#, VB.NET, PascalABC.NET).

Таким образом, последовательное выполнение заданий из всех групп, приведенных в данном разделе, позволит ознакомиться с основными средствами библиотеки STL и проверить их работу на практике, после чего закрепить навыки их использования при решении достаточно сложных задач. В каждой группе и ее подгруппах можно выделить наборы однотипных заданий одинакового уровня сложности, что дает возможность преподавателю при организации лабораторных занятий предлагать каждому студенту свой вариант индивидуальных заданий, достаточный для изучения всех объектов STL.

## *3.1. Знакомство с итераторами и алгоритмами*

Для всех исходных наборов данных вначале указывается их размер (т. е. количество элементов), а затем значения элементов.

Во всех заданиях группы STL1Iter обработка данных выполняется без использования контейнеров: исходные наборы считываются итератором чтения и сразу передаются требуемому алгоритму, а полученные результаты немедленно выводятся в файл или пересылаются задачнику с помощью соответствующего итератора записи.

Если алгоритм требует применения функционального объекта, то в случае компиляторов, поддерживающих стандарт C++11, следует использовать лямбда-выражения.

В большинстве ситуаций достаточно применять лямбда-выражения без захвата переменных или, если компилятор не поддерживает стандарт C++11, – обычные функции (см. указание к STL1Iter2). Более сложные задания требуют применения лямбда-выражений с захваченными внешними переменными или структур с дополнительными полями и операцией () (см. указания к STL1Iter8 и STL1Iter9).

---

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи STL1Iter17, приведенным в п. 2.1.

---

STL1Iter1. Дан набор целых чисел. Найти количество нулей в исходном наборе. Использовать итератор ptin_iterator и алгоритм count.

STL1Iter2. Дан текстовый файл с именем *name*, содержащий строковые представления вещественных чисел. Найти количество положительных чисел в исходном файле. Использовать итератор istream_iterator и алгоритм count_if.

Указание. Если компилятор поддерживает стандарт C++11, то в качестве функционального объекта используйте лямбда-выражение `[ ](double e){ return e > 0; }`, указав его в списке параметров алгоритма.

В противном случае используйте функцию f, описав ее как `bool f(double e){ return e > 0; }`.

STL1Iter3. Дан текстовый файл с именем *name*, содержащий английские слова. Найти количество слов длины 6. Использовать итератор istream_iterator и алгоритм count_if.

Указание. Рекомендации по использованию функциональных объектов приведены в указании к STL1Iter2.

STL1Iter4. Дан набор вещественных чисел. Вывести все элементы этого набора в том же порядке. Использовать итераторы ptin_iterator, ptout_iterator и алгоритм copy.

STL1Iter5. Дан текстовый файл с именем *name*, содержащий строковые представления целых чисел. Вывести все числа из файла в том же порядке. Использовать итераторы istream_iterator, ptout_iterator и алгоритм copy.

STL1Iter6. Дана строка *name* и набор символов. Записать в текстовый файл с именем *name* исходный набор символов в том же порядке, добавляя после каждого символа пробел. Использовать итераторы ptin_iterator, ostream_iterator и алгоритм copy.

STL1Iter7. Дан текстовый файл с именем *name1*, содержащий строковые представления целых чисел, и строка *name2*. Записать в текстовый файл с именем *name2* все ненулевые числа из исходного файла в том

же порядке, располагая каждое число на новой строке. Использовать итераторы istream_iterator, ostream_iterator и алгоритм remove_copy.

STL1Iter8. Дано целое число *K* (> 0), текстовый файл с именем *name1*, содержащий английские слова, и строка *name2*. Записать в текстовый файл с именем *name2* все слова из исходного файла, длина которых не превосходит *K*, сохранив исходный порядок их следования и располагая каждое слово на новой строке. Использовать итераторы istream_iterator, ostream_iterator и алгоритм remove_copy_if.

Указание. Если компилятор поддерживает стандарт C++11, то в качестве функционального объекта используйте лямбда-выражение с захваченной внешней переменной k, указав это лямбда-выражение в списке параметров алгоритма: [k](string e){ return e.length() > k; }.

В противном случае опишите структуру f, которая содержит член len, конструктор с параметром, задающим начальное значение члена len, и операцию (), определенную следующим образом: bool operator()(string e) { return e.length() > len; }. В алгоритме в качестве функционального объекта надо указать конструктор f(k), инициализирующий член len значением k.

STL1Iter9. Дан текстовый файл с именем *name*, содержащий строковые представления целых чисел. Вывести числа из исходного файла с нечетными порядковыми номерами (т. е. первое число, третье число и т. д.). Использовать итераторы istream_iterator, ptout_iterator и алгоритм remove_copy_if.

Указание. Если компилятор поддерживает стандарт C++11, то в качестве функционального объекта используйте лямбда-выражение с внешней переменной num, захваченной по ссылке: [&num](int e){ return num++ % 2 == 0; } (перед вызовом алгоритма переменная num должна быть инициализирована значением 1).

В противном случае опишите структуру f, которая содержит член num, конструктор с параметром, задающим начальное значение члена num, и операцию (), определенную следующим образом: bool operator()(int e){ return num++ % 2 == 0; }. В алгоритме в качестве функционального объекта надо указать конструктор f(1), инициализирующий член num значением 1.

STL1Iter10. Дан набор вещественных чисел, содержащий не менее двух элементов. Вывести числа из исходного набора с четными порядковыми номерами (т. е. второе число, четвертое число и т. д.). Использовать итераторы ptin_iterator, ptout_iterator и алгоритм remove_copy_if.

STL1Iter11. Решить задачу STL1Iter7, используя вместо алгоритма remove_copy цикл for с параметром-итератором.

Указание. Опишите итератор out типа ostream_iterator и организуйте цикл for с параметром-итератором in типа istream_iterator и операцией инкремента in++. В цикле выполняйте проверку введенного числа *in и в случае, если оно удовлетворяет требуемому условию, выполняйте оператор out = *in. Выражение *in можно использовать многократно; оно всегда будет соответствовать последнему прочитанному элементу данных (чтение очередного элемента выполняется при создании объекта istream_iterator и при выполнении операции инкремента).

STL1Iter12. Решить задачу STL1Iter8, используя вместо алгоритма remove_copy_if цикл for с параметром-итератором.

Указание. Ср. с задачей STL1Iter11.

STL1Iter13. Решить задачу STL1Iter9, используя вместо алгоритма remove_copy_if цикл for с параметром-итератором.

Указание. Опишите итератор out типа ptout_iterator и организуйте цикл for с параметром-итератором in типа istream_iterator и операцией инкремента in++. В цикле выполняйте оператор out = *in++.

STL1Iter14. Решить задачу STL1Iter10, используя вместо алгоритма remove_copy_if цикл for с параметром-итератором.

Указание. Ср. с задачей STL1Iter13. В данном случае надо использовать цикл с параметром in типа ptin_iterator.

STL1Iter15. Дана строка *name* и набор целых чисел. Записать в текстовый файл с именем *name* все числа из исходного набора в том же порядке, заменяя каждое число 0 на число 10 и добавляя после каждого числа два пробела. Использовать итераторы ptin_iterator, ostream_iterator и алгоритм replace_copy.

STL1Iter16. Дан набор символов. Вывести все символы из исходного набора в том же порядке, заменяя цифровые символы на символ подчеркивания. Использовать итераторы ptin_iterator, ptout_iterator и алгоритм replace_copy_if.

STL1Iter17. Дана строка *name* и набор символов. Записать в текстовый файл с именем *name* удвоенные кодовые значения всех символов из исходного набора в том же порядке, добавляя после каждого числа один пробел. Использовать итераторы ptin_iterator, ostream_iterator и алгоритм transform.

Примечание. Решение данной задачи приведено в п. 2.1.

STL1Iter18. Дана строка *name* и целое число *K* (> 0). Записать в текстовый файл с именем *name* *K* символов «*». Использовать итератор ostream_iterator и алгоритм fill_n.

STL1Iter19. Даны вещественные числа *A*, *D* и целое число *N*. Вывести *N* первых членов арифметической прогрессии с первым элементом *A* и разностью *D*. Использовать итератор ptout_iterator и алгоритм generate_n.

Указание. Рекомендации по использованию функциональных объектов приведены в указании к STL1Iter9. В данном случае лямбда-выражение должно содержать внешнюю переменную d, захваченную по значению, и еще одну вспомогательную вещественную переменную, захваченную по ссылке и соответствующую текущему элементу прогрессии. Если в качестве функционального объекта используется структура, то она должна содержать два члена (которые соответствуют разности и текущему элементу прогрессии).

STL1Iter20. Дана строка *name* и целое число *N* ($1 \le N \le 26$). Записать в текстовый файл с именем *name* *N* начальных прописных букв латинского алфавита. Использовать итератор ostream_iterator и алгоритм generate_n.

Указание. Рекомендации по использованию функциональных объектов приведены в указании к STL1Iter9.

STL1Iter21. Даны два текстовых файла с именами *name1* и *name2*, содержащие строковые представления целых чисел, причем в каждом файле числа располагаются по убыванию. Вывести все числа из исходных файлов в виде единой последовательности, упорядоченной по убыванию. Использовать итераторы istream_iterator, ptout_iterator и алгоритм merge с параметром – функциональным объектом.

Указание. В данном случае не требуется применять ни лямбда-выражение, ни специально описанный функциональный объект. Достаточно указать стандартный функциональный объект greater<int>(), реализующий операцию > («больше»). Парным к нему является функциональный объект less, который реализует операцию < («меньше») (операция < используется по умолчанию во многих алгоритмах).

STL1Iter22. Даны два текстовых файла с именами *name1* и *name2*, содержащие английские слова, причем в каждом файле слова располагаются по возрастанию длины, а слова равной длины – в лексикографическом порядке. Вывести все слова из исходных файлов в виде единой последовательности, упорядоченной таким же образом, как и исход-

ные файлы. Использовать итераторы istream_iterator, ptout_iterator и алгоритм merge с параметром – функциональным объектом.

STL1Iter23. Даны два текстовых файла с именами *name1* и *name2*, содержащие одинаковое количество строковых представлений вещественных чисел. Вывести разности $B_1 - A_1$, $B_2 - A_2$, …, $B_N - A_N$, где $N$ – количество чисел в каждом файле, $A_1$, $A_2$, …, $A_N$ – числа из файла *name1*, а $B_1$, $B_2$, …, $B_N$ – числа из файла *name2*. Использовать итераторы istream_iterator, ptout_iterator и алгоритм transform.

STL1Iter24. Дан текстовый файл с именем *name1*, содержащий английские слова, строка *name2* и набор английских слов, размер которого не превосходит количество слов, содержащихся в файле *name1*. Записать в текстовый файл с именем *name2* все слова из исходного набора, дополнив каждое слово символом «-» (дефис) и словом из файла *name1* с тем же порядковым номером (если файл *name1* содержит больше слов, чем исходный набор, то лишние слова в файле игнорируются). Каждое дополненное слово записывать в файл с новой строки. Использовать итераторы ptin_iterator, istream_iterator, ostream_iterator и алгоритм transform.

## *3.2. Последовательные контейнеры*

Во всех заданиях данной группы, кроме пяти начальных, заготовки решения уже содержат операторы, обеспечивающие заполнение исходных контейнеров и отладочную печать преобразованных контейнеров. Кроме того, заготовки содержат закомментированные операторы вывода полученных контейнеров в раздел результатов.

Например, если в задании требуется преобразовать вектор целых чисел, то заготовка решения имеет следующий вид:

```
typedef ptin_iterator<int> ptin;
typedef ptout_iterator<int> ptout;
vector<int> V(ptin(0), ptin());

Show(V.begin(), V.end(), "V: ");
//copy(V.begin(), V.end(), ptout());
```

Решение следует ввести перед оператором отладочной печати Show, после чего раскомментировать вызов copy для вывода результатов и их автоматической проверки.

Во всех заданиях элементами контейнеров являются целые числа.

---

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи STL2Seq22, приведенным в п. 2.2.

---

### 3.2.1. Заполнение контейнеров и доступ к элементам. Обратные итераторы

STL2Seq1. Дан текстовый файл с именем *name*, содержащий строковые представления целых чисел. Заполнить вектор *V* числами из исходного файла и вывести элементы вектора в исходном порядке. Для заполнения вектора использовать итератор istream_iterator и конструктор вектора, для вывода элементов вектора использовать алгоритм copy, применяя его к итераторам полученного вектора и итератору ptout_iterator.

Указание. Для инициализации вектора V данными из файла f с применением итераторов ввода наиболее естественной представляется следующая конструкция:

```
vector<int> V(istream_iterator<int>(f), istream_iterator<int>());
```

Однако в силу особенностей лексического разбора языка C++ данная конструкция будет интерпретироваться как объявление прототипа функции V с двумя параметрами – указателями на функции. Простейшим вариантом решения проблемы является заключение одного из параметров конструктора вектора в дополнительные круглые скобки. Можно также ввести вспомогательную переменную для одного или обоих итераторов:

```
istream_iterator<int> eof;
vector<int> V(istream_iterator<int>(f), eof);
```

или

```
istream_iterator<int> fi(f), eof;
vector<int> V(fi, eof);
```

STL2Seq2. Дан набор целых чисел. Заполнить список *L* исходными числами и вывести элементы списка *L* вначале в исходном, а затем в обратном порядке. Для заполнения списка использовать итератор ptin_iterator и конструктор списка, для вывода элементов списка дважды применить алгоритм copy к итераторам полученного списка и итератору ptout_iterator, причем для вывода элементов в обратном порядке использовать обратные итераторы, возвращаемые функциями-членами rbegin и rend.

Указание. Если при использовании конструктора итератора ptin_iterator указывать его параметр в виде константы, то проблемы, описанные в указании к задаче STL2Seq1, не возникают (если же используется переменная, то возникает аналогичная проблема, решить которую можно так, как описано в указании к задаче STL2Seq1, или

превратив переменную в выражение: вместо x указать, например, x + 0).

STL2Seq3. Дан набор целых чисел с четным количеством элементов. Заполнить вектор *V* исходными числами и вывести вначале вторую половину элементов вектора *V*, а затем первую половину (в каждой половине порядок элементов не изменять). Для заполнения вектора использовать итератор ptin_iterator и конструктор вектора, для вывода элементов дважды применить алгоритм copy к итераторам полученного вектора и итератору ptout_iterator. При задании некоторых итераторов вектора использовать выражение V.size()/2 и операцию «+».

STL2Seq4. Дан набор целых чисел с четным количеством элементов. Заполнить дек *D* исходными числами и вывести первую половину элементов дека *D* в обратном порядке, а затем – вторую половину (также в обратном порядке). Для заполнения дека использовать итератор ptin_iterator и конструктор дека, для вывода элементов дважды применить алгоритм copy к итераторам полученного дека и итератору ptout_iterator. Использовать обратные итераторы дека; при определении некоторых из них использовать выражение D.size()/2 и операцию «+».

STL2Seq5. Дан набор целых чисел, количество которых делится на 3. Заполнить список *L* исходными числами и вывести вначале первую треть элементов списка *L* в исходном порядке, затем вторую треть элементов в обратном порядке, а затем последнюю треть (также в обратном порядке). Для заполнения списка использовать итератор ptin_iterator и конструктор списка, для вывода элементов трижды применить алгоритм copy к итераторам полученного списка и итератору ptout_iterator. Использовать как прямые, так и обратные итераторы; при определении некоторых из них использовать выражение L.size()/3 и функцию advance.

STL2Seq6. Даны вектор *V*, дек *D* и список *L*. Каждый исходный контейнер содержит не менее трех элементов, количество элементов является нечетным. Удвоить значения первого, среднего и последнего элемента каждого из исходных контейнеров. Для доступа к первому и последнему элементу любого контейнера использовать функции-члены front и back. Для доступа к среднему элементу вектора и дека использовать операцию индексирования []. Для доступа к среднему элементу списка использовать функцию advance для итераторов и операцию разыменования итератора *.

STL2Seq7. Даны вектор *V*, дек *D* и список *L*. Каждый исходный контейнер содержит не менее двух элементов, количество элементов является

четным. Поменять значения двух средних элементов каждого из исходных контейнеров. Использовать алгоритм swap (не путать его с одноименной функцией-членом контейнера).

### 3.2.2. Вставка элементов

STL2Seq8. Дан вектор *V* с четным количеством элементов. Добавить в середину вектора 5 нулевых элементов. Использовать один вызов функции-члена insert.

STL2Seq9. Дан дек *D* с нечетным количеством элементов *N* ($\geq 5$). Добавить в начало дека пять его средних элементов в исходном порядке. Использовать один вызов функции-члена insert.

STL2Seq10. Дан список *L*, количество элементов которого делится на 3. Добавить в конец списка первую треть его исходных элементов в обратном порядке. Использовать один вызов функции-члена insert.

STL2Seq11. Даны вектор *V* и список *L*. Каждый исходный контейнер содержит не менее 5 элементов. Вставить после элемента списка с порядковым номером 5 первые 5 элементов вектора в обратном порядке. Использовать один вызов функции-члена insert.

STL2Seq12. Даны дек *D* и список *L*. Каждый исходный контейнер содержит не менее 5 элементов. Вставить перед пятым с конца элементом списка последние 5 элементов дека в обратном порядке. Использовать один вызов функции-члена insert.

STL2Seq13. Даны вектор *V* и дек *D*, имеющие четное количество элементов. Добавить в конец вектора первую половину элементов дека (в исходном порядке), а в начало дека – вторую половину исходных элементов вектора (в обратном порядке). Использовать два вызова функции-члена insert.

STL2Seq14. Дан вектор *V*. Вставить после каждого элемента исходного вектора число −1. Использовать функцию-член insert в цикле с параметром-итератором.

Указание. Организуйте перебор элементов вектора в цикле с параметром-итератором i. Вставку выполняйте в позицию ++i, обязательно присваивая параметру i значение, возвращаемое функцией-членом insert.

STL2Seq15. Дан дек *D* с четным количеством элементов. Вставить перед каждым элементом из первой половины исходного дека число −1. Использовать функцию-член insert в цикле с числовым параметром.

Указание. Используйте цикл с числовым параметром, повторяющийся *N*/2 раз (где *N* – исходный размер дека). Свяжите вспомогательный

итератор i с началом второй половины элементов дека. В цикле вызывайте функцию-член insert с первым параметром, равным --i, обязательно присваивая возвращаемое значение итератору i.

STL2Seq16. Решить задачу STL2Seq15, используя функцию-член insert в цикле по обратному итератору.

Указание. Организуйте цикл for с параметром r – обратным итератором для перебора элементов первой половины исходного дека в обратном порядке. После вставки нового элемента функцией insert (с первым параметром --r.base()) следует использовать возвращаемое значение функции insert для обновления значения итератора r. Поскольку функция insert возвращает обычный итератор, необходимо выполнить явное приведение этого итератора к типу vector<int>::reverse_iterator (если компилятор поддерживает стандарт C++11, то в качестве имени типа можно указать decltype(r)). При такой организации цикла в его заголовке не следует указывать операцию инкремента ++r.

STL2Seq17. Дан список *L*. Вставить перед каждым элементом исходного списка число −1. Использовать функцию-член insert в цикле с параметром-итератором.

Указание. См. указание к задаче STL2Seq14. Вставку следует выполнять в позицию i, причем не следует присваивать параметру i значение, возвращаемое функцией insert (иначе произойдет зацикливание). Вставка новых элементов в список (в отличие от вставки новых элементов в вектор или дек) не делает недействительными итераторы, указывающие на последующие элементы списка.

STL2Seq18. Дан список *L* с четным количеством элементов. Вставить после каждого элемента из первой половины исходного списка число −1. Использовать функцию-член insert в цикле с параметром-итератором.

Указание. См. указание к задаче STL2Seq15. Для задания начального значения итератора i используйте функцию advance. В данном случае первым параметром функции-члена insert должно быть выражение i--, причем сохранять значение, возвращенное функцией insert, не требуется.

STL2Seq19. Решить задачу STL2Seq18, используя функцию-член insert в цикле по обратному итератору.

Указание. См. указание к задаче STL2Seq16. Для определения диапазона итераторов используйте функцию advance. Первым параметром функции-члена insert должно быть выражение r.base(). В случае списков в результате подобной вставки обратный итератор r будет возвра-

щать значение вставленного элемента. Если, как в задаче STL2Seq16, сохранять возвращаемое значение функции insert в итераторе r (с применением явного приведения типа), то в результате итератор r будет возвращать значение элемента, предшествующего вставленному (т. е. расположенного слева от вставленного). Поэтому в конце итерации цикла необходимо дополнительно выполнить операцию ++r. В данной программе, в отличие от решения задачи STL2Seq16, можно не сохранять значение, возвращаемое функцией insert, но в этом случае в конце цикла требуется дважды инкрементировать итератор r (например, указывая в заголовке цикла выражение ++r, ++r).

### 3.2.3. Удаление элементов

STL2Seq20. Дан дек *D* с нечетным количеством элементов *N* (≥ 3). Удалить средний элемент дека. Использовать функцию-член erase.

STL2Seq21. Дан вектор *V* с нечетным количеством элементов *N* (≥ 5). Удалить три средних элемента вектора. Использовать один вызов функции-члена erase.

STL2Seq22. Даны список *L* и вектор *V*; список *L* имеет нечетное количество элементов. Переместить средний элемент списка *L* в конец вектора *V*. Использовать функции-члены push_back и erase.

Примечание. Решение данной задачи приведено в п. 2.2.

STL2Seq23. Даны список *L* и дек *D*; дек *D* имеет четное количество элементов. Переместить первую половину элементов дека *D* в начало списка *L*. Использовать один вызов функций-членов insert и erase.

STL2Seq24. Даны списки $L_1$ и $L_2$; список $L_1$ имеет нечетное количество элементов. Переместить средний элемент списка $L_1$ в конец списка $L_2$. Использовать один вызов функции-члена splice.

STL2Seq25. Даны списки $L_1$ и $L_2$; список $L_2$ имеет четное количество элементов. Переместить первую половину элементов списка $L_2$ в начало списка $L_1$. Использовать один вызов функции-члена splice.

STL2Seq26. Даны списки $L_1$ и $L_2$, имеющие четное количество элементов. Поменять местами первую половину исходного списка $L_1$ и вторую половину исходного списка $L_2$. Использовать два вызова функции-члена splice.

STL2Seq27. Дан вектор *V*. Удалить все элементы исходного вектора с нечетными порядковыми номерами (считая, что начальный элемент вектора имеет порядковый номер 1). Использовать функцию-член erase в цикле с параметром-итератором.

Указание. Организуйте перебор элементов вектора в цикле с параметром-итератором i, увеличивая параметр в заголовке цикла. Удаление выполняйте в позиции i, обязательно присваивая параметру i значение, возвращаемое функцией-членом erase. В конце каждой итерации цикла дополнительно проверяйте, достигнут ли конец вектора.

STL2Seq28. Дан дек *D* с количеством элементов, кратным 4. Удалить в первой половине исходного дека все элементы с четными порядковыми номерами (считая, что начальный элемент дека имеет порядковый номер 1). Использовать функцию-член erase в цикле с числовым параметром.

Указание. Используйте цикл с числовым параметром, повторяющийся *N*/4 раз (где *N* – исходный размер дека). Свяжите вспомогательный итератор i с началом дека. В цикле вызывайте функцию-член erase с параметром, равным ++i, обязательно присваивая возвращаемое значение итератору i.

STL2Seq29. Решить задачу STL2Seq28, используя функцию-член erase в цикле по обратному итератору.

Указание. Организуйте цикл с параметром r – обратным итератором для перебора элементов первой половины исходного дека в обратном порядке (параметр r должен увеличиваться в заголовке цикла). После удаления элемента функцией-членом erase (с параметром --r.base()) следует обязательно использовать возвращаемое значение функции erase для обновления значения итератора r. Поскольку функция erase возвращает обычный итератор, необходимо выполнить явное приведение этого итератора к типу deque<int>::reverse_iterator (если компилятор поддерживает стандарт C++11, то в качестве имени типа можно указать decltype(r)).

STL2Seq30. Дан список *L*. Удалить все элементы исходного списка с четными порядковыми номерами (считая, что начальный элемент списка имеет порядковый номер 1). Использовать функцию-член erase в цикле с параметром-итератором.

Указание. См. указание к задаче STL2Seq27. В данном случае в заголовке цикла по итератору i не нужно использовать операцию инкремента, а удаление следует выполнять в позиции ++i, присваивая параметру i значение, возвращаемое функцией-членом erase.

STL2Seq31. Дан список *L* с количеством элементов, кратным 4. Удалить в первой половине исходного списка все элементы с нечетными порядковыми номерами (считая, что начальный элемент списка имеет

порядковый номер 1). Использовать функцию-член erase в цикле с параметром-итератором.

Указание. См. указание к задаче STL2Seq28. В данном случае параметром функции-члена erase должно быть выражение i++, причем сохранять значение, возвращенное функцией erase, не следует; вместо этого в конце каждой итерации требуется выполнять дополнительный инкремент: ++i.

STL2Seq32. Решить задачу STL2Seq31, используя функцию-член erase в цикле по обратному итератору.

Указание. См. указание к задаче STL2Seq29. Для определения диапазона итераторов используйте функцию advance. В данном случае в качестве параметра функции-члена erase можно использовать выражение --(++r).base(), причем сохранять значение, возвращаемое функцией erase, не требуется (не требуется также выполнять инкремент итератора r в заголовке цикла).

STL2Seq33. Дан список $L$ с элементами $A_1$, $A_2$, $A_3$, …, $A_{N-1}$, $A_N$ ($N$ – четное). Изменить порядок элементов в списке на следующий: $A_N$, $A_1$, $A_{N-1}$, $A_2$, $A_{N-2}$, …, $A_{N/2}$, $A_{N/2-1}$. Для этого использовать два итератора i и r, связав их с первым и последним элементом списка. В цикле, который должен повторяться $N$/2 раз, вызывать функцию-член splice с первым параметром i++ и третьим параметром r--.

STL2Seq34. Даны два списка $L_1$ и $L_2$ с одинаковым количеством элементов $N$. Получить в списке $L_2$ комбинированный набор элементов исходных списков вида $A_1$, $B_1$, $A_2$, $B_2$, $A_3$, $B_3$, …, $A_N$, $B_N$, где $A_I$ обозначают элементы исходного списка $L_1$, а $B_I$ – элементы исходного списка $L_2$. Для этого использовать итераторы i1 (для списка $L_1$) и i2 (для списка $L_2$), связав их с первым элементом соответствующего списка. В цикле, который должен повторяться $N$ раз, вызывать функцию-член splice для списка $L_2$ с первым параметром ++i2 и третьим параметром i1++.

## *3.3. Обобщенные алгоритмы*

Во всех заданиях данной группы заготовки решения уже содержат операторы, обеспечивающие заполнение исходных контейнеров. Если в задании требуется преобразовать содержимое исходного контейнера, то заготовка содержит отладочную печать преобразованного контейнера. Кроме того, если в задании требуется вывести преобразованный контейнер, заготовка содержит закомментированный оператор вывода требуемого контейнера (для вывода используется алгоритм copy).

Например, если в задании требуется преобразовать вектор целых чисел, то заготовка решения имеет следующий вид:

```
typedef ptin_iterator<int> ptin;
typedef ptout_iterator<int> ptout;
vector<int> V(ptin(0), ptin());

Show(V.begin(), V.end(), "V: ");
//copy(V.begin(), V.end(), ptout());
```

Решение следует ввести перед оператором отладочной печати Show, после чего раскомментировать вызов copy для вывода результатов и их автоматической проверки.

Если тип элементов контейнера в задании не указан, то предполагается, что элементами являются целые числа.

---

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи STL3Alg35, приведенным в п. 2.3.1.

---

### 3.3.1. Алгоритмы поиска

**STL3Alg1**. Дан вектор *V*. Удалить второй из элементов вектора, равных нулю. Если нулевых элементов меньше двух, то вектор не изменять. Использовать два вызова алгоритма find и функцию-член erase.

**STL3Alg2**. Дан дек *D*. Удалить последний нулевой элемент дека. Если нулевых элементов нет, то дек не изменять. Использовать алгоритм find с обратными итераторами и функцию-член erase.

**Указание.** Алгоритм find может возвращать обратный итератор, однако функция-член erase не позволяет его использовать для удаления элемента. Необходимо перейти от обратного итератора r к связанному с ним обычному итератору, используя функцию-член обратного итератора r.base(). При этом надо учитывать, что функция r.base() возвращает итератор, связанный с элементом, *следующим* за тем, с которым связан обратный итератор r. Поэтому в функции-члене erase необходимо указать один из двух вариантов: --r.base() или (++r).base() (предполагается, что r – это обратный итератор, который вернул алгоритм find, и этот итератор отличен от rend).

**STL3Alg3**. Дан список *L*. Удалить первый и последний нулевой элемент списка. Если нулевых элементов нет, то список не изменять, если нулевой элемент всего один, то удалить только его. Использовать два вызова алгоритма find и два вызова функции-члена erase.

**Указание.** Имеющуюся в классе list функцию-член remove в данном случае использовать нельзя, так как она удаляет *все* элементы с определенным значением. См. также указание к задаче STL3Alg2.

STL3Alg4. Дан список *L*. Удалить все элементы списка, расположенные между первым и вторым отрицательным элементом (не включая сами отрицательные элементы). Если список не содержит отрицательных элементов, то не изменять его, если отрицательный элемент всего один, то удалить все элементы, следующие за этим отрицательным элементом. Использовать два вызова алгоритма find_if и один вызов функции-члена erase.

STL3Alg5. Дан список *L*, содержащий как отрицательные, так и положительные элементы. Вставить нулевой элемент после первого отрицательного элемента и перед последним положительным элементом. Использовать два вызова алгоритма find_if и два вызова функции-члена insert.

Указание. По поводу использования алгоритма find_if с обратным итератором см. указание к задаче STL3Alg2.

STL3Alg6. Даны вектор *V* и список *L*; вектор *V* имеет четное количество элементов. Продублировать последний элемент списка, совпадающий с каким-либо элементом из первой половины исходного вектора. Если список не содержит требуемых элементов, то не изменять его. Использовать алгоритм find_first_of и функцию-член insert для списка.

STL3Alg7. Дан вектор *V* с четным количеством элементов. Добавить нулевой элемент перед последним элементом в первой половине вектора, совпадающим с каким-либо элементом из второй половины этого же вектора. Если вектор не содержит требуемых элементов, то не изменять его. Использовать алгоритм find_first_of и функцию-член insert.

STL3Alg8. Дано целое число *K* (> 0) и вектор *V*, содержащий только нули и единицы. Удалить в векторе *V* последний набор из *K* подряд расположенных нулей (если в этом наборе имеется больше *K* нулей, то требуется удалить только последние *K* из них). Если вектор не содержит требуемого набора нулей, то не изменять его. Использовать алгоритм search_n и функцию-член erase, а также обратные итераторы.

STL3Alg9. Дано целое число *K* (> 0) и вектор *V*. Продублировать в векторе *V* первый набор из *K* подряд расположенных положительных чисел, вставив после этого набора его копию (если в наборе имеется больше *K* положительных чисел, то лишние числа во внимание не принимаются, и дублируется только начальная часть набора, содержащая *K* чисел). Если вектор не содержит требуемого набора чисел, то не изменять его. Использовать алгоритм search_n с параметром – функциональным объектом.

STL3Alg10. Даны список *L* и дек *D*; дек *D* содержит четное количество элементов. Продублировать в деке *D* первый набор элементов, расположенный в его второй половине и совпадающий с элементами списка *L*, взятыми в обратном порядке. Например, для списка 1, 2 и дека 2, 1, 1, 2, 3, 2, 1, 1, 2, 1 дек должен быть преобразован следующим образом: 2, 1, 1, 2, 3, 2, 1, (2, 1,) 1, 2, 1 (вставленные элементы заключены в скобки). Если дек не содержит требуемого набора чисел, то не изменять его. Использовать алгоритм search и функцию-член insert.

STL3Alg11. Даны вектор *V* и дек *D*; дек *D* содержит четное количество элементов. Удалить в векторе *V* последний набор элементов, в котором четные и нечетные числа располагаются в том же порядке, что и в первой половине дека *D*. Например, для дека 1, 2, 3, 4, 5, 6 и вектора 11, 14, (15, 16, 17,) 17, 18, 10 в векторе должны быть удалены элементы, заключенные в скобки. Если вектор не содержит требуемого набора чисел, то не изменять его. Использовать алгоритм search с параметром – функциональным объектом и функцию-член erase.

STL3Alg12. Решить задачу STL3Alg11, используя алгоритм find_end с параметром – функциональным объектом и функцию-член erase.

STL3Alg13. Дан список *L*. Удвоить значения в последней паре соседних совпадающих элементов. Например, список 1, 2, 2, 3, 3, 1, 2, 2, 2, 4 должен быть преобразован следующим образом: 1, 2, 2, 3, 3, 1, 2, 4, 4, 4. Если список не содержит соседних совпадающих элементов, то не изменять его. Использовать алгоритм adjacent_find и обратные итераторы.

STL3Alg14. Дан вектор *V*. Обнулить первую пару соседних элементов, имеющих одинаковую четность. Например, список 1, 2, 3, 4, 6, 8, 3, 1 должен быть преобразован следующим образом: 1, 2, 3, 0, 0, 8, 3, 1. Если список не содержит соседних элементов с одинаковой четностью, то не изменять его. Использовать алгоритм adjacent_find с параметром – функциональным объектом.

STL3Alg15. Дан дек *D*. Удвоить значения всех пар соседних элементов, исходные значения которых отличаются только знаком (преобразованные элементы не должны повторно анализироваться при последующих вызовах алгоритма). Например, дек 1, −1, 2, −3, 3, 3, −3, 6 должен быть преобразован следующим образом: 2, −2, 2, −6, 6, 6, −6, 6. Использовать алгоритм adjacent_find с параметром – функциональным объектом в цикле по итератору дека.

### 3.3.2. Базовые модифицирующие алгоритмы. Итераторы вставки

STL3Alg16. Даны числа $A$ и $B$ и векторы $V_1$ и $V_2$, каждый из которых содержит не менее 10 элементов. Заполнить первые 5 элементов каждого вектора значениями $A$, а последние 5 – значениями $B$. При преобразовании вектора $V_1$ использовать два вызова алгоритма fill, при преобразовании вектора $V_2$ использовать два вызова алгоритма fill_n.

STL3Alg17. Даны числа $A$ и $B$ и векторы $V_1$ и $V_2$. Добавить в начало каждого вектора 5 элементов со значениями $A$, а в конец – 5 элементов со значениями $B$. При преобразовании вектора $V_1$ использовать два вызова алгоритма fill_n с функциями inserter и back_inserter (данные функции возвращают итераторы вставки), при преобразовании вектора $V_2$ использовать два вызова функции-члена insert.

Замечание. Второй способ является более эффективным.

STL3Alg18. Дано число $N$ (> 0) и дек $D$, содержащий не менее $2N$ элементов. Заполнить первые $N$ элементов дека последовательностью чисел 1, 2, …, $N$, а последние $N$ элементов дека – последовательностью $N$, $N-1$, …, 2, 1. Использовать два вызова алгоритма generate с одинаковым параметром – функциональным объектом, а также обратные итераторы.

STL3Alg19. Дано число $N$ (> 0) и дек $D$. Добавить в начало дека последовательность чисел 1, 2, …, $N$, а в его конец – последовательность $N$, $N-1$, …, 2, 1. Использовать два вызова алгоритма generate_n с одинаковым параметром – функциональным объектом, а также итераторы вставки, возвращаемые функциями front_inserter и back_inserter.

STL3Alg20. Даны списки $L_1$ и $L_2$, каждый из которых содержит четное количество элементов. Поменять местами первую и вторую половину каждого списка (например, список 1, 2, 3, 4 должен быть преобразован следующим образом: 3, 4, 1, 2). Для первого списка использовать алгоритм swap_ranges, для второго – алгоритм rotate. Использовать также функцию advance.

STL3Alg21. Дано число $K$ ($0 < K < 10$) и списки $L_1$ и $L_2$, каждый из которых содержит не менее 10 элементов. Выполнить для списка $L_1$ циклический сдвиг элементов вправо на $K$ позиций, а для списка $L_2$ – циклический сдвиг влево на $K$ позиций. Использовать алгоритм rotate и функцию advance.

STL3Alg22. Дано число $K$ ($0 < K < 5$) и список $L$, содержащий не менее 10 элементов. Набор из первых 5 элементов списка скопировать в конец списка, выполнив для этой копии циклический сдвиг на $K$ позиций вправо, а набор из последних 5 элементов исходного списка ско-

пировать в начало списка, выполнив для этой копии циклический сдвиг на $K$ позиций влево. Использовать два вызова алгоритма rotate_copy и итераторы вставки, а также функцию advance.

STL3Alg23. Даны списки $L_1$ и $L_2$, у каждого из которых количество элементов делится на 4. В первом списке инвертировать (расположить в обратном порядке) первую половину элементов, во втором списке – вторую половину. Для первого списка использовать алгоритм swap_ranges и обратные итераторы, для второго – алгоритм reverse. Использовать также функцию advance.

STL3Alg24. Даны списки $L_1$ и $L_2$, каждый из которых содержит четное количество элементов. В каждом списке продублировать первую половину, добавив ее элементы в конец списка в обратном порядке. Для первого списка использовать алгоритм reverse_copy и итератор вставки, для второго – функцию-член insert и обратные итераторы. Использовать также функцию advance.

Замечание. Второй способ является более эффективным.

STL3Alg25. Дан вектор $V$ с четным количеством элементов. В первой половине исходного вектора заменить все отрицательные числа на $-1$, а во второй – все положительные числа на 1. Использовать два вызова алгоритма replace_if с различными параметрами – функциональными объектами.

STL3Alg26. Дан список $L$ с четным количеством элементов. Скопировать в конец списка все элементы, расположенные в его первой половине, заменив при этом отрицательные элементы на нули и расположив скопированные элементы в обратном порядке. Использовать алгоритм replace_copy_if, итератор вставки и обратные итераторы, а также функцию advance.

STL3Alg27. Дан дек $D$ с четным количеством элементов. Решить для него задачу STL3Alg26. Поскольку операция вставки делает все итераторы дека недействительными, для решения задачи использовать вспомогательный дек $D_0$. Инициализировать дек $D_0$ первой половиной элементов дека $D$, после чего применить алгоритм replace_copy_if, используя дек $D_0$ как источник, а дек $D$ как приемник данных.

STL3Alg28. Дан список $L$ с четным количеством элементов. Скопировать в начало списка все положительные элементы, расположенные в его второй половине, сохранив для них исходный порядок следования. Использовать алгоритм remove_copy_if и итератор вставки, а также функцию advance.

STL3Alg29. Дан вектор $V$ с четным количеством элементов. Решить для него задачу STL3Alg28. Поскольку операция вставки в начало вектора делает все его итераторы недействительными, для решения задачи использовать вспомогательный вектор $V_0$. Инициализировать вектор $V_0$ второй половиной элементов вектора $V$, после чего применить алгоритм remove_copy_if, используя вектор $V_0$ как источник, а вектор $V$ как приемник данных.

STL3Alg30. Дан вектор $V$ с четным количеством элементов. Удалить все нулевые элементы, расположенные во второй половине исходного вектора. Использовать алгоритм remove и функцию-член erase.

Указание. Алгоритм remove и другие алгоритмы, связанные с удалением элементов, не удаляют требуемые элементы, а лишь *перемещают* их в конец исходного диапазона и возвращают итератор, указывающий на начало диапазона с перемещенными элементами. Для фактического удаления перемещенных элементов после выполнения алгоритма необходимо вызвать функцию-член erase.

STL3Alg31. Дан вектор $V$ с четным количеством элементов. Удалить все четные элементы, расположенные в первой половине исходного вектора. Использовать алгоритм remove_if и функцию-член erase.

Указание. См. указание к задаче STL3Alg30.

STL3Alg32. Дан вектор $V$. В каждой группе подряд расположенных элементов с одинаковой четностью удалить все элементы, кроме начального. Использовать алгоритм unique с параметром – функциональным объектом и функцию-член erase.

Указание. См. указание к задаче STL3Alg30.

STL3Alg33. Дан дек $D$. Из каждой группы подряд расположенных положительных, отрицательных или нулевых элементов выбрать последний и скопировать выбранные элементы в том же порядке в начало дека. Например, дек 1, 2, −3, 4, 0, 0, −5, −6, 7, 8, 9 должен быть преобразован следующим образом: (2, −3, 4, 0, −6, 9,) 1, 2, −3, 4, 0, 0, −5, −6, 7, 8, 9 (добавленные элементы заключены в скобки). Использовать вспомогательный дек $D_0$ (инициализировав его с помощью дека $D$), алгоритм unique_copy с функциональным объектом и обратными итераторами, а также подходящий итератор вставки (в алгоритме дек $D_0$ должен быть источником, а дек $D$ – приемником данных).

STL3Alg34. Дан дек $D$ с четным количеством элементов. Заменить в нем вторую половину элементов на удвоенные значения элементов первой половины, расположив эти удвоенные значения в обратном порядке. Использовать алгоритм transform с обратным итератором.

STL3Alg35. Дан список $L$ с четным количеством элементов $N$. Добавить в начало списка $N/2$ новых элементов со следующими значениями: $A_1 + A_N$, $A_2 + A_{N-1}$, …, $A_{N/2} + A_{N/2+1}$, где $A_1$, $A_2$, …, $A_N$ обозначают исходные элементы списка. Использовать алгоритм transform с обратным итератором и итератором вставки.

Примечание. Решение данной задачи приведено в п. 2.3.1.

### 3.3.3. Сортировка и слияние

STL3Alg36. Дан вектор $V$ с нечетным количеством элементов $N$ ($\geq 3$). Определить значения трех средних элементов вектора после того, как вектор будет отсортирован (по возрастанию), и вывести их в порядке возрастания. Использовать алгоритмы nth_element (для нахождения центрального элемента), max_element (для нахождения элемента, предшествующего центральному) и min_element (для нахождения элемента, следующего за центральным).

STL3Alg37. Дан вектор $V$, содержащий не менее 3 элементов. Определить значения трех начальных элементов вектора после того, как вектор будет отсортирован (по возрастанию), и вывести их в порядке возрастания. Использовать один вызов алгоритма partial_sort и алгоритм copy для вывода требуемых элементов.

STL3Alg38. Дан вектор $V$, содержащий не менее 3 элементов. Определить значения трех конечных элементов вектора после того, как вектор будет отсортирован (по возрастанию) и вывести их в порядке убывания. Использовать один вызов алгоритма partial_sort с параметром – функциональным объектом greater и алгоритм copy для вывода требуемых элементов.

STL3Alg39. Дан вектор $V$ с четным количеством элементов. Добавить в его конец первую половину элементов, которые содержал бы отсортированный по возрастанию вариант исходного вектора. Использовать функцию-член insert для увеличения количества элементов вектора на требуемую величину и алгоритм partial_sort_copy.

STL3Alg40. Дан список $L$. Определить количество четных и нечетных чисел в исходном списке (вначале вывести количество четных, затем количество нечетных чисел). Использовать алгоритм partition и два вызова функции distance для итераторов.

STL3Alg41. Дан список $L$. Перегруппировать элементы списка, расположив в нем вначале положительные, а затем неположительные элементы (порядок расположения элементов в каждой группе должен совпадать с исходным). Использовать алгоритм stable_partition.

STL3Alg42. Дан список *L*. Перегруппировать элементы списка, расположив в нам вначале отрицательные, затем нулевые, а затем положительные элементы (порядок расположения элементов в каждой группе должен совпадать с исходным). Использовать два вызова алгоритма stable_partition.

STL3Alg43. Дан список *L* с четным количеством элементов. Перегруппировать элементы списка, расположив в нем вначале все четные элементы из первой половины исходного списка, затем все нечетные элементы, а затем – все четные элементы из второй половины списка (порядок расположения элементов в каждой группе должен совпадать с исходным). Использовать два вызова алгоритма stable_partition.

STL3Alg44. Дан дек *D*. Решить для него задачу STL3Alg42, используя один вызов алгоритма stable_sort с параметром – функциональным объектом.

STL3Alg45. Дан дек *D*, элементами которого являются английские слова. Отсортировать его элементы по возрастанию их длин, а элементы одинаковой длины – в алфавитном порядке. Использовать алгоритм sort для сортировки по алфавиту и затем алгоритм stable_sort с параметром – функциональным объектом для сортировки по длине. Выводить новое содержимое дека после применения каждого алгоритма.

STL3Alg46. Дан дек *D*, элементами которого являются английские слова. Отсортировать его элементы по убыванию их длин, а элементы одинаковой длины – в алфавитном порядке. Использовать единственный вызов алгоритма sort с параметром – функциональным объектом, включающим как сравнение строк, так и сравнение их длин.

STL3Alg47. Дан вектор *V* с четным количеством элементов. Известно, что первая половина вектора уже отсортирована по возрастанию. Отсортировать все элементы вектора по возрастанию, выполнив вначале сортировку его второй половины алгоритмом sort, а затем слияние обеих половин алгоритмом inplace_merge. Выводить новое содержимое вектора *V* после применения каждого алгоритма.

STL3Alg48. Дан вектор *V*, количество элементов которого делится на 3. Известно, что каждая треть вектора уже упорядочена по возрастанию. Решить для исходного вектора задачу STL3Alg42, используя два вызова алгоритма inplace_merge с одним и тем же параметром – функциональным объектом.

### 3.3.4. Перестановки и работа с кучей

STL3Alg49. Дан вектор *V*, элементы которого не упорядочены по убыванию. Получить все перестановки исходных элементов, которые боль-

ше исходного вектора в лексикографическом порядке и вывести полученные перестановки по возрастанию их лексикографического порядка (при этом последняя выведенная перестановка должна содержать элементы, упорядоченные по убыванию). Использовать в цикле алгоритм next_permutation.

STL3Alg50. Дан вектор $V$, элементы которого не упорядочены по убыванию. Решить задачу STL3Alg49, используя в цикле алгоритм prev_permutation с параметром – функциональным объектом greater.

STL3Alg51. Дан вектор $V$, содержащий не менее 3 элементов. Решить для него задачу STL3Alg38, используя один вызов алгоритма make_heap, цикл из трех итераций, в котором вызывается алгоритм pop_heap, и алгоритм copy (с обратными итераторами) для вывода требуемых элементов.

STL3Alg52. Дан вектор $V$, содержащий не менее 3 элементов. Решить для него задачу STL3Alg37, используя один вызов алгоритма make_heap, цикл из трех итераций, в котором вызывается алгоритм pop_heap, и алгоритм copy (с обратными итераторами) для вывода требуемых элементов.

Указание. Используйте варианты алгоритмов с параметром – функциональным объектом.

STL3Alg53. Дан дек $D_1$, содержащий не менее 3 элементов, и дек $D_2$, содержащий ровно 3 элемента. Определить три максимальных элемента среди всех элементов набора, полученного из дека $D_1$ в результате удаления трех его максимальных элементов и добавления к нему всех элементов из дека $D_2$. Вывести требуемые элементы в порядке убывания. Использовать алгоритмы make_heap, pop_heap и push_heap, а также алгоритм copy (с обратными итераторами) для вывода требуемых элементов. Размеры исходных деков не изменять, алгоритм make_heap использовать один раз.

Указание. Алгоритм make_heap используется для преобразования элементов первого дека в кучу. Затем выполняется цикл с тремя вызовами алгоритма pop_heap для удаления из кучи трех максимальных элементов, цикл с тремя вызовами алгоритма push_heap для добавления в кучу элементов из второго дека и еще один цикл с тремя вызовами алгоритма pop_heap для определения требуемых элементов. Алгоритм copy вызывается после цикла.

STL3Alg54. Дан дек $D_1$, содержащий не менее 3 элементов, и дек $D_2$, содержащий ровно 3 элемента. Определить три минимальных элемента среди всех элементов набора, полученного из дека $D_1$ в результате

удаления трех его минимальных элементов и добавления к нему всех элементов из дека $D_2$. Вывести требуемые элементы в порядке возрастания. Использовать алгоритмы make_heap, pop_heap и push_heap, а также алгоритм copy (с обратными итераторами) для вывода требуемых элементов. Размеры исходных деков не изменять, алгоритм make_heap использовать один раз.

Указание. См. указания к задачам STL3Alg52 и STL3Alg53.

### 3.3.5. Численные алгоритмы

STL3Alg55. Дан вектор *V* с четным количеством элементов. Найти сумму элементов из первой и из второй половины вектора. Использовать два вызова алгоритма accumulate.

STL3Alg56. Дан вектор *V*. Найти сумму отрицательных и сумму положительных элементов вектора. Использовать два вызова алгоритма accumulate с параметрами – функциональными объектами.

STL3Alg57. Дан вектор *V*, элементами которого являются английские слова. Получить строку, содержащую начальные символы всех элементов вектора, расположенные в обратном порядке. Использовать алгоритм accumulate с параметром – функциональным объектом.

STL3Alg58. Дан вектор *V*, элементами которого являются английские слова. Получить строку, содержащую конечные символы всех элементов вектора, расположенные в исходном порядке. Использовать алгоритм accumulate с параметром – функциональным объектом.

STL3Alg59. Дан вектор *V*, элементами которого являются английские слова. Получить строку, являющуюся суммой строк, описанных в задачах STL3Alg57 и STL3Alg58 (в указанном порядке). Использовать один вызов алгоритма accumulate с параметром – функциональным объектом.

STL3Alg60. Дан список *L*. Получить вектор *V* вещественных чисел, содержащий значения среднего арифметического для всех пар соседних элементов исходного списка (количество элементов вектора *V* должно быть на 1 меньше количества элементов списка *L*). Например, для исходного списка 1, 3, 4, 6 полученный вектор должен содержать значения 2.0, 3.5, 5.0. Использовать алгоритм adjacent_difference с итератором вставки и функциональным объектом, а также функцию-член erase для вектора *V*.

STL3Alg61. Дан список *L*, элементами которого являются английские слова. Получить дек *D* со строковыми элементами, каждый из которых строится по паре соседних элементов исходного списка *L* следующим образом: последняя буква правого элемента пары приписывается

справа к первой букве левого элемента пары. Количество элементов дека $D$ должно быть на 1 меньше количества элементов списка $L$. Например, для исходного списка ABC, DEF, KLM, XYZ полученный дек должен содержать строки AF, DM, KZ. Использовать алгоритм adjacent_difference с итератором вставки и функциональным объектом, а также функцию-член erase для дека $D$.

STL3Alg62. Даны векторы $V_1$ и $V_2$ одинакового размера $N$, каждый из которых содержит координаты точки в $N$-мерном пространстве с евклидовой метрикой $r(x,y) = (\Sigma(x_i - y_i)^2)^{1/2}$ (сумма берется по всем $i$ от 0 до $N-1$). Найти значение квадрата метрики для двух данных точек. Использовать алгоритм inner_product с двумя функциональными объектами (в качестве первого из них можно указать стандартный объект plus).

STL3Alg63. Даны векторы $V_1$ и $V_2$ одинакового размера $N$, каждый из которых содержит координаты точки в $N$-мерном пространстве с метрикой $r(x,y) = \max\{|x_i - y_i|\}$ (максимум берется по всем $i$ от 0 до $N-1$). Найти значение метрики для двух данных точек. Использовать алгоритм inner_product с двумя функциональными объектами.

STL3Alg64. Дано целое число $N$ ($1 \le N \le 16$). Получить вектор $V$ вещественных чисел, равных факториалам всех чисел от 1 до $N$ (вещественные числа используются для того, чтобы избежать целочисленного переполнения). Для этого последовательно вызвать конструктор вектора $V$, заполняющий его единицами, алгоритм partial_sum (с параметром – функциональным объектом) для получения в векторе всех чисел от 1 до $N$ и алгоритм partial_sum (с параметром – функциональным объектом, в качестве которого достаточно использовать стандартный объект multiplies) для получения в векторе требуемых факториалов.

## 3.4. Строки как последовательные контейнеры

Во всех заданиях данной группы требуется обработать строку, рассматривая ее как последовательный контейнер и используя конструкторы класса string, его функции-члены insert и erase и/или подходящие обобщенные алгоритмы. Вспомогательные строки при решении задач не использовать.

---

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи STL4Str27, приведенным в п. 2.3.2.

---

STL4Str1. Дано целое положительное число $N$ и символ $C$. Используя конструктор класса string, заполнить строку $S$, записав в нее $N$ копий символа $C$.

**STL4Str2.** Дана строка $S_0$ четной длины. Используя конструктор класса string с параметрами-итераторами, записать в строку $S$ вторую половину символов строки $S_0$ в обратном порядке.

**STL4Str3.** Дана строка $S$ четной длины $N$ ($N \geq 4$). Дважды используя подходящий обобщенный алгоритм, поменять местами первый и последний символ, а также два средних символа строки $S$.

**STL4Str4.** Дано целое число $K$ (> 0) и строка $S$ четной длины. Используя один вызов функции-члена insert с параметром-итератором, добавить в середину строки $S$ $K$ символов «*».

**STL4Str5.** Дано целое число $K$ (> 0) и строка $S$, длина которой больше $K$. Используя один вызов функции-члена insert с параметрами-итераторами, добавить в конец строки $S$ ее первые $K$ символов в обратном порядке.

**STL4Str6.** Дано целое число $K$ (> 0) и строки $S_1$ и $S_2$. Строка $S_1$ содержит не менее $K$ символов, а строка $S_2$ – не менее двух символов. Используя два вызова функции-члена insert с параметрами-итераторами, преобразовать строку $S_2$, вставив после ее первого символа первые $K$ символов строки $S_1$ в исходном порядке, а перед ее последним символом – последние $K$ символов строки $S_1$ в обратном порядке.

**STL4Str7.** Дана строка $S$. Используя функцию-член insert в цикле с параметром-итератором, вставить перед каждым символом строки $S$ символ «*».

**STL4Str8.** Дана строка $S$, содержащая не менее двух символов. Используя функцию-член insert в цикле с параметром-итератором, вставить после каждого символа строки $S$ с четным порядковым номером три символа «*» (например, строка «abcde» должна быть преобразована в «ab***cd***e»).

**STL4Str9.** Дана строка $S$ четной длины. Используя функцию-член insert в цикле с параметром-итератором, продублировать каждый символ из первой половины строки $S$ (например, строка «abcdef» должна быть преобразована в «aabbccdef»).

**STL4Str10.** Дана строка $S$ нечетной длины $N$ ($N \geq 5$). Используя один вызов функции-члена erase с параметрами-итераторами, удалить три средних символа строки.

**STL4Str11.** Дана строка $S$, содержащая не менее трех символов. Используя функцию-член erase в цикле с параметром-итератором, удалить каждый третий символ строки $S$.

STL4Str12. Дана строка $S$, содержащая не менее трех символов. Используя подходящий обобщенный алгоритм в цикле, переместить каждый третий символ строки $S$ в ее конец, располагая перемещенные символы в исходном порядке.

STL4Str13. Дана строка $S$, длина которой делится на 3. Используя подходящий обобщенный алгоритм, переместить вторую треть символов строки $S$ в ее конец (с сохранением порядка символов).

STL4Str14. Дана строка $S$, содержащая не менее трех символов «*». Используя три вызова подходящего обобщенного алгоритма и два вызова функции-члена erase, удалить в строке $S$ второй и последний из этих символов.

STL4Str15. Дана строка $S$. Используя два вызова подходящего обобщенного алгоритма и один вызов функции-члена erase, удалить все символы, расположенные в строке $S$ между первой и последней цифрой. Если цифр меньше двух, то строку не изменять.

STL4Str16. Дана строка $S$ четной длины. Используя в цикле подходящий обобщенный алгоритм и функцию-член erase, удалить во второй половине строки $S$ все символы, совпадающие с символами из ее первой половины.

STL4Str17. Дана строка $S$. Используя подходящий обобщенный алгоритм и функцию-член erase, удалить все соседние цифровые и буквенные символы, оставив в каждом наборе подряд идущих цифр или букв только начальный символ. Например, строка «123abc456de» должна быть преобразована к виду «1a4d». Считать, что строка $S$ содержит только латинские буквенные символы.

STL4Str18. Дано целое число $K$ (> 0) и строка $S$, длина которой превосходит число $K$. Используя подходящий обобщенный алгоритм и функцию-член insert, продублировать в строке $S$ первую из подстрок, содержащих $K$ подряд расположенных цифр. Если подстрока содержит более $K$ цифр, то продублировать $K$ начальных цифр этой подстроки. Если требуемых подстрок нет, то строку не изменять.

STL4Str19. Даны строки $S_1$ и $S_2$, длина строки $S_1$ не превосходит длины строки $S_2$. Используя подходящий обобщенный алгоритм и функцию-член erase, удалить в строке $S_2$ последнюю подстроку, совпадающую с инвертированной строкой $S_1$ (строку $S_1$ не изменять). Если требуемых подстрок нет, то строку $S_2$ не изменять.

STL4Str20. Дано число $N$ ($1 \le N \le 26$) и строка $S$. Используя два вызова подходящего обобщенного алгоритма с итератором вставки, добавить в начало и в конец строки $S$ по $N$ первых заглавных букв латинского

алфавита, расположив эти буквы в конце строки по алфавиту, а в начале – в порядке, обратном алфавитному. Например, при $N = 3$ и $S$ = «567» требуется получить строку «CBA567ABC».

STL4Str21. Дано целое число $K$ (> 0) и строка $S$ четной длины, большей $2K$. Используя два вызова подходящего обобщенного алгоритма, выполнить в первой половине строки $S$ циклический сдвиг символов на $K$ позиций влево, а в ее второй половине – циклический сдвиг символов на $K$ позиций вправо.

STL4Str22. Дана строка $S$ четной длины. Используя два вызова подходящего обобщенного алгоритма, заменить в первой половине строки $S$ все цифровые символы на символ «*», а в ее второй половине – все заглавные латинские буквы на символ «_».

STL4Str23. Дана строка $S$. Используя один вызов подходящего обобщенного алгоритма, изменить регистр всех буквенных символов строки $S$ на противоположный, а цифровые символы заменить на символ «*».

STL4Str24. Даны строки $S_1$ и $S_2$ одинаковой длины. Используя подходящий обобщенный алгоритм, преобразовать строку $S_2$, заменив на символ «*» те ее символы, которые не совпадают с символами строки $S_1$, расположенными в тех же позициях. Например, в случае $S_1$ = «ab2CD» и $S_2$ = «as2Cd» строка $S_2$ должна быть преобразована к виду «a*2C*».

STL4Str25. Дана строка $S$. Используя подходящий обобщенный алгоритм, скопировать в конец строки $S$ все содержащиеся в ней цифровые символы в том же порядке.

STL4Str26. Дана строка $S$. Используя подходящий обобщенный алгоритм и функцию-член erase, удалить в строке $S$ все цифровые символы.

STL4Str27. Дана строка $S$. Используя подходящий обобщенный алгоритм, перегруппировать элементы строки $S$, переместив все цифровые символы в ее начало в том же порядке.

Примечание. Решение данной задачи приведено в п. 2.3.2.

STL4Str28. Дана строка $S$ четной длины. Используя два вызова подходящего обобщенного алгоритма, отсортировать первую половину строки $S$ по убыванию кодов символов, а вторую – по их возрастанию.

## *3.5. Ассоциативные контейнеры*

Во всех заданиях данной группы заготовки решения уже содержат операторы, обеспечивающие заполнение исходных контейнеров. Если тип элементов контейнера не указан, то предполагается, что элементами являются целые числа.

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи STL5Assoc20, приведенным в п. 2.4.1.

### 3.5.1. Множества. Теоретико-множественные алгоритмы

STL5Assoc1. Дан вектор $V$ с четным количеством элементов. Если все значения, содержащиеся во второй половине вектора, входят хотя бы один раз в его первую половину, то вывести true, иначе вывести false. Использовать алгоритм includes, применив его к двум *множествам* (контейнерам типа set), созданным на основе вектора $V$.

STL5Assoc2. Дан вектор $V_0$, целое число $N$ (> 0) и набор векторов $V_1$, …, $V_N$. Известно, что размер вектора $V_0$ не превосходит размера любого из векторов $V_1$, …, $V_N$. Найти количество векторов $V_I$, $I = 1, …, N$, в которых содержатся все элементы вектора $V_0$ (без учета их повторений). Использовать алгоритм includes, применяя его в цикле к двум множествам, одно из которых создано на основе вектора $V_0$, а другое на очередной итерации содержит элементы очередного из векторов $V_I$, $I = 1, …, N$.

STL5Assoc3. Дан вектор $V_0$, целое число $N$ (> 0) и набор векторов $V_1$, …, $V_N$. Известно, что размер вектора $V_0$ не превосходит размера любого из векторов $V_1$, …, $V_N$. Найти количество векторов $V_I$, $I = 1, …, N$, в которых содержатся все элементы вектора $V_0$ (с учетом их повторений). Использовать алгоритм includes, применяя его в цикле к двум *мультимножествам* (контейнерам типа multiset), одно из которых создано на основе вектора $V_0$, а другое на очередной итерации содержит элементы очередного из векторов $V_I$, $I = 1, …, N$.

STL5Assoc4. Дана строка *name* и вектор $V$ с четным количеством элементов. Найти все различные числа, которые одновременно входят и в первую, и во вторую половину исходного вектора, и записать их в текстовый файл с именем *name* в возрастающем порядке, добавляя после каждого числа символ пробела. Использовать алгоритм set_intersection для двух вспомогательных множеств и итератора ostream_iterator.

STL5Assoc5. Дана строка *name* и вектор $V$ с четным количеством элементов. Найти все различные числа, которые входят во вторую половину исходного вектора и при этом отсутствуют в первой половине. Записать найденные числа в текстовый файл с именем *name* в убывающем порядке, выводя каждое число на новой строке. Использовать алгоритм set_difference для двух вспомогательных множеств и итератора ostream_iterator. Чтобы обеспечить вывод чисел в нужном порядке,

при создании множеств и в алгоритме использовать функциональный объект greater.

STL5Assoc6. Даны векторы $V_1$ и $V_2$. Найти все числа (с учетом повторений), которые входят хотя бы в один из исходных векторов, и вывести их в порядке возрастания; при этом если, например, некоторое число входит в один из векторов 3 раза, а в другой 5 раз, то его надо вывести 5 раз. Использовать алгоритм set_union для двух вспомогательных мультимножеств и итератора ptout_iterator.

STL5Assoc7. Даны векторы $V_1$ и $V_2$ с различным количеством элементов. Найти все числа (с учетом повторений), которые входят в один из исходных векторов и отсутствуют в другом, и вывести их в порядке убывания; при этом если, например, некоторое число входит в один из векторов 3 раза, а в другой 5 раз, то его надо вывести 2 раза. Использовать алгоритм set_symmetric_difference для двух вспомогательных мультимножеств и итератора ptout_iterator. Чтобы обеспечить вывод чисел в нужном порядке, при создании множеств и в алгоритме использовать функциональный объект greater.

STL5Assoc8. Дан вектор $V$, содержащий не менее трех различных чисел. Вывести все его различные элементы, кроме максимального и минимального, в порядке убывания. Использовать вспомогательное множество и алгоритм copy с обратными итераторами, указывающими на предпоследний и первый элементы множества, и с итератором ptout_iterator.

STL5Assoc9. Решить задачу STL5Assoc8, не используя вспомогательное множество. Вместо него последовательно использовать для исходного вектора алгоритмы sort, unique и copy.

STL5Assoc10. Дан вектор $V$, содержащий не менее трех различных чисел. Вывести все его элементы (с учетом повторений), кроме минимального и максимального, в порядке возрастания. Использовать вспомогательное мультимножество, его функции-члены lower_bound и upper_bound и алгоритм copy с итератором ptout_iterator.

STL5Assoc11. Решить задачу STL5Assoc10, не используя вспомогательное мультимножество. Вместо него последовательно использовать для исходного вектора алгоритмы sort, lower_bound, upper_bound и copy.

STL5Assoc12. Дан вектор $V$. Определить количество повторений каждого числа в векторе $V$ и вывести все различные элементы вектора $V$ вместе с количеством их повторений (в порядке возрастания значений элементов); количество повторений выводить сразу после значения соответствующего элемента. Использовать вспомогательное мультимно-

жество и цикл, в котором вызывается функция-член upper_bound для мультимножества и функция difference для его итераторов.

STL5Assoc13. Решить задачу STL5Assoc12, не используя вспомогательное мультимножество и функцию difference. Вместо этого использовать для исходного вектора алгоритм sort и в цикле алгоритм upper_bound и операцию разности для итераторов вектора.

STL5Assoc14. Решить задачу STL5Assoc12, не используя функцию upper_bound и цикл. Вместо этого использовать вспомогательное мультимножество *M* и вспомогательное множество *S* (создав их на основе исходного вектора) и алгоритм for_each для множества *S* с параметром – функциональным объектом, в котором использовать функцию-член count мультимножества *M*.

### 3.5.2. Отображения. Группировка и объединение данных

STL5Assoc15. Дан вектор *V*. Определить количество повторений каждого числа в векторе *V* и вывести все различные элементы вектора *V* вместе с количеством их повторений (в порядке возрастания значений элементов); количество повторений выводить сразу после значения соответствующего элемента. Использовать вспомогательное *отображение* *M* (класс map), ключами которого являются различные элементы вектора *V*, а значениями – количество повторений этих элементов. При заполнении отображения *M* не использовать условные конструкции (достаточно операций индексирования [] и инкремента). Элементы вектора *V* (при заполнении отображения *M*) и элементы отображения *M* (при выводе полученных результатов) перебирать в цикле с параметром-итератором соответствующего контейнера.

STL5Assoc16. Решить задачу STL5Assoc15, используя для перебора элементов вектора *V* и отображения *M* вызовы алгоритма for_each или, если компилятор поддерживает стандарт C++11, циклы for по элементам контейнера.

STL5Assoc17. Дан вектор *V*, элементами которого являются английские слова, набранные заглавными буквами. Определить суммарную длину слов, начинающихся с одной и той же буквы, и вывести все различные буквы, с которых начинаются элементы вектора *V*, вместе с суммарной длиной этих элементов (в алфавитном порядке букв); длину выводить сразу после соответствующей буквы. Использовать вспомогательное отображение *M*, ключами которого являются начальные буквы элементов вектора *V*, а значениями – суммарная длина этих элементов. При заполнении отображения *M* не использовать условные конструкции (достаточно операций индексирования [], инкремента и функции-члена size для строк). Элементы вектора *V* (при заполнении

отображения *M*) и элементы отображения *M* (при выводе полученных результатов) перебирать в цикле с параметром-итератором соответствующего контейнера.

STL5Assoc18. Решить задачу STL5Assoc17, используя для перебора элементов вектора *V* и отображения *M* вызовы алгоритма for_each или, если компилятор поддерживает стандарт C++11, циклы for по элементам контейнера.

STL5Assoc19. Дан вектор *V*. Выполнить *группировку* элементов вектора *V*, используя в качестве ключа группировки последнюю (т. е. правую) цифру элемента: в одну группу должны входить все элементы вектора *V*, оканчивающиеся на одну и ту же цифру (сгруппированные элементы должны располагаться в том же порядке, в котором они располагались в исходном векторе). Представить результат группировки в виде отображения *M*, ключами которого являются ключи группировки, а значениями – векторы, содержащие сгруппированные элементы (таким образом, отображение *M* должно иметь тип map<int, vector<int>>). Вывести полученное отображение (для каждого элемента отображения *M* вначале выводить ключ, а затем элементы связанного с ним вектора). Для перебора элементов контейнеров использовать циклы с параметрами-итераторами.

STL5Assoc20. Дан вектор *V*, элементами которого являются английские слова, набранные заглавными буквами. Выполнить группировку элементов вектора *V*, используя в качестве ключа группировки первую букву элемента: в одну группу должны входить все элементы вектора *V*, начинающиеся с одной и той же буквы (сгруппированные элементы должны располагаться в алфавитном порядке с учетом повторений). Представить результат группировки в виде отображения *M*, ключами которого являются ключи группировки, а значениями – мультимножества, содержащие сгруппированные элементы (таким образом, отображение *M* должно иметь тип map<char, multiset<string>>). Вывести полученное отображение (для каждого элемента отображения *M* вначале выводить ключ, а затем элементы связанного с ним мультимножества). Для перебора элементов контейнеров использовать алгоритм for_each или, если компилятор поддерживает стандарт C++11, цикл for по элементам контейнера.

Примечание. Решение данной задачи приведено в п. 2.4.1.

STL5Assoc21. Дан вектор *V*. Выполнить группировку элементов вектора *V*, используя в качестве ключа группировки последнюю (т. е. правую) цифру элемента: в одну группу должны входить все элементы вектора *V*, оканчивающиеся на одну и ту же цифру (сгруппированные

элементы должны располагаться в том же порядке, в котором они располагались в исходном векторе). Представить результат группировки в виде *мультиотображения M* (класса multimap), ключами которого являются ключи группировки, т. е. последние цифры элементов вектора *V*, а значениями – элементы вектора, оканчивающиеся на соответствующую цифру (таким образом, отображение *M* должно иметь тип multimap<int, int>). Вывести полученное отображение (для каждого элемента отображения *M* вначале выводить ключ, а затем связанный с ним элемент вектора *V*; ключи могут повторяться). Для перебора элементов контейнеров использовать циклы с параметрами-итераторами.

STL5Assoc22. Дан вектор *V*, элементами которого являются английские слова, набранные заглавными буквами. Выполнить группировку элементов вектора *V*, используя в качестве ключа группировки последнюю букву элемента: в одну группу должны входить все элементы вектора *V*, оканчивающиеся одной и той же буквой (сгруппированные элементы должны располагаться в порядке, обратном порядку их расположения в исходном векторе). Представить результат группировки в виде мультиотображения *M*, ключами которого являются ключи группировки, т. е. последние буквы элементов вектора *V*, а значения-ми – элементы вектора, оканчивающиеся на соответствующую букву (таким образом, отображение *M* должно иметь тип multimap<char, string>). Вывести полученное отображение (для каждого элемента отображения *M* вначале выводить ключ, а затем связанный с ним элемент вектора *V*; ключи могут повторяться). Для перебора элементов контейнеров использовать алгоритм for_each или, если компилятор поддерживает стандарт C++11, цикл for по элементам контейнера.

Указание. Для размещения элементов в группе в порядке, обратном исходному порядку их расположения в векторе, достаточно при формировании мультиотображения *M* перебирать элементы вектора *V* в обратном порядке (используя обратные итераторы). Требуемый результат можно получить и при прямом переборе элементов вектора, если использовать вариант функции-члена M.insert с первым параметром-итератором («подсказкой»), в качестве которого указывать возвращаемое значение функции-члена M.lower_bound для соответствующего ключа.

STL5Assoc23. Дан вектор *V*. В каждой группе его элементов, оканчивающихся на одну и ту же цифру, найти сумму значений этих элементов, за исключением начального элемента группы (предполагается, что элементы группы располагаются в том же порядке, что и в исходном векторе). Если группа состоит из единственного элемента, то сумма

должна равняться 0. Для каждой группы вывести соответствующую ей цифру и найденную сумму, упорядочивая выводимые пары по возрастанию цифр. Использовать вспомогательное отображение *M* и вариант группировки, описанный в задаче STL5Assoc19. Для нахождения сумм использовать алгоритм accumulate.

STL5Assoc24. Решить задачу STL5Assoc23, используя вспомогательное мультиотображение *M* и вариант группировки, описанный в задаче STL5Assoc21. Для нахождения сумм использовать алгоритм accumulate.

STL5Assoc25. Решить задачу STL5Assoc23, не прибегая к этапу предварительной группировки. Использовать отображение *M* с ключом – последней цифрой числа и значением – суммой требуемых чисел (ср. с задачей STL5Assoc15). При формировании отображения *M* использовать, наряду с операциями индексирования и инкремента, функцию-член find.

STL5Assoc26. Дан вектор *V*, элементами которого являются английские слова, набранные заглавными буквами. В каждой группе его элементов, оканчивающихся на одну и ту же букву, получить строку, являющуюся суммой всех слов из этой группы, кроме последнего слова (предполагается, что элементы группы располагаются в том же порядке, что и в исходном векторе). При построении строки добавлять после каждого слова пробел. Если группа состоит из единственного элемента, то строка должна остаться пустой. Для каждой группы вывести соответствующую ей букву и найденную строку, упорядочивая выводимые пары в алфавитном порядке букв. Использовать вспомогательное отображение *M* и вариант группировки, описанный в задаче STL5Assoc20. Для нахождения сумм использовать алгоритм accumulate.

STL5Assoc27. Решить задачу STL5Assoc26, используя вспомогательное мультиотображение *M* и вариант группировки, описанный в задаче STL5Assoc22. Для нахождения сумм использовать алгоритм accumulate.

STL5Assoc28. Решить задачу STL5Assoc26, не прибегая к этапу предварительной группировки. Использовать отображение *M* с ключом – последней буквой слова и значением – суммой требуемых слов (ср. с задачей STL5Assoc17). При формировании отображения *M* организовать перебор исходного вектора *V* с конца и использовать, наряду с операциями индексирования и инкремента, функцию-член find.

STL5Assoc29. Даны векторы $V_1$ и $V_2$; все элементы в каждом векторе различны. Получить вектор $V$, содержащий пары чисел (типа pair<int, int>), удовлетворяющие следующим условиям: первый элемент пары принадлежит вектору $V_1$, второй принадлежит вектору $V_2$, и оба элемента оканчиваются одной и той же цифрой. Полученный набор пар называется *внутренним объединением* векторов $V_1$ и $V_2$ по ключу, определяемому последними цифрами исходных чисел. Порядок следования пар определяется исходным порядком элементов вектора $V_1$, а для равных первых элементов – исходным порядком элементов вектора $V_2$. Для построения вектора $V$ выполнить группировку элементов вектора $V_2$ по ключу – последней цифре, используя вариант группировки со вспомогательным отображением $M$, описанный в задаче STL5Assoc19, после чего в цикле по элементам вектора $V_1$ сформировать требуемое внутреннее объединение, перебирая для каждого элемента вектора $V_1$ соответствующие ему элементы отображения $M$. Вывести размер полученного вектора $V$ и все его элементы.

STL5Assoc30. Решить задачу STL5Assoc29, выполняя группировку элементов вектора $V_2$ способом, описанным в задаче STL5Assoc21 (с использованием вспомогательного мультиотображения $M$). Для поиска в мультиотображении $M$ элементов с требуемым ключом использовать функцию-член equal_range.

STL5Assoc31. Даны векторы $V_1$ и $V_2$, элементами которых являются английские слова, набранные заглавными буквами, причем все слова в каждом векторе различны. Получить вектор $V$, являющийся внутренним объединением векторов $V_1$ и $V_2$ (см. STL5Assoc29), каждая пара которого содержит слова одинаковой длины. Порядок следования пар определяется алфавитным порядком первых элементов пар, а для равных первых элементов – порядком вторых элементов, обратным алфавитному. Для построения вектора $V$ выполнить группировку элементов вектора $V_2$ по ключу – длине слова, используя вариант группировки со вспомогательным отображением $M$ типа map<int, set<string>> (ср. с задачей STL5Assoc20), после чего в цикле по отсортированным элементам вектора $V_1$ сформировать требуемое внутреннее объединение, перебирая для каждого элемента вектора $V_1$ соответствующие ему элементы отображения $M$. Вывести размер полученного вектора $V$ и все его элементы.

STL5Assoc32. Даны векторы $V_1$ и $V_2$, элементами которых являются английские слова, набранные заглавными буквами, причем все слова в каждом векторе различны. Получить вектор $V$, являющийся внутренним объединением векторов $V_1$ и $V_2$ (см. STL5Assoc29), в каждой паре которого первое слово начинается с буквы, которой оканчивается

второе слово. Порядок следования пар определяется алфавитным порядком первых элементов пар, а для равных первых элементов – порядком вторых элементов, обратным порядку их следования в векторе $V_2$. Для построения вектора $V$ выполнить группировку элементов вектора $V_2$ по ключу – последней букве слова, используя вариант группировки со вспомогательным мультиотображением $M$, описанный в задаче STL5Assoc22, после чего в цикле по отсортированным элементам вектора $V_1$ сформировать требуемое внутреннее объединение, перебирая для каждого элемента вектора $V_1$ соответствующие ему элементы отображения $M$. Вывести размер полученного вектора $V$ и все его элементы.

STL5Assoc33. Даны векторы $V_1$ и $V_2$; все элементы в каждом векторе различны и являются положительными числами. Получить вектор $V$ пар типа pair<int, vector<int>>, в которых первый элемент пары принадлежит вектору $V_1$, а второй представляет собой набор тех элементов вектора $V_2$, которые оканчиваются той же цифрой, что и первый элемент пары (если подходящие элементы в векторе $V_2$ отсутствуют, то второй элемент пары должен содержать единственный элемент, равный 0). Полученный набор пар называется *левым внешним объединением* векторов $V_1$ и $V_2$ по ключу, определяемому последними цифрами исходных чисел. Порядок следования пар определяется исходным порядком элементов вектора $V_1$; порядок чисел, входящих во вторые элементы пар, определяется исходным порядком элементов вектора $V_2$. Для построения вектора $V$ выполнить группировку элементов вектора $V_2$ по ключу – последней цифре, используя вариант группировки со вспомогательным отображением $M$, описанный в задаче STL5Assoc19, после чего в цикле по элементам вектора $V_1$ сформировать требуемое внешнее объединение, перебирая для каждого элемента вектора $V_1$ соответствующие ему элементы отображения $M$. Вывести полученный вектор $V$, указывая после первого элемента пары все числа, входящие во второй элемент данной пары.

STL5Assoc34. Даны векторы $V_1$ и $V_2$, элементами которых являются английские слова, набранные заглавными буквами, причем все слова в каждом векторе различны. Получить левое внешнее объединение векторов $V_1$ и $V_2$ (см. STL5Assoc33) по ключу – длине слова: каждому элементу вектора $V_1$ должен соответствовать набор всех элементов вектора $V_2$, имеющих ту же длину, или набор, содержащий только пустую строку, если требуемые элементы в векторе $V_2$ отсутствуют. Левое внешнее объединение должно быть упорядочено в алфавитном порядке элементов из вектора $V_1$; в каждом наборе элементов, соответствующих элементу из $V_1$, порядок должен быть обратным алфа-

витному порядку. Реализовать левое внешнее объединение в виде отображения $M$ с ключом – строкой и значением – строковым множеством. Для построения множества $M$ выполнить группировку элементов вектора $V_2$ по ключу – длине слова, используя вариант группировки со вспомогательным отображением $M_0$ типа map<int, set<string, greater<string>>> (ср. с задачей STL5Assoc20), после чего в цикле по элементам вектора $V_1$ сформировать требуемое внутреннее объединение, перебирая для каждого элемента вектора $V_1$ соответствующие ему элементы отображения $M_0$. Вывести полученное отображение $M$, указывая после каждого его ключа (т. е. элемента вектора $V_1$) все слова, входящие в связанное с этим ключом множество значений (т. е. все соответствующие ему элементы вектора $V_2$ или, при их отсутствии, пустую строку).

STL5Assoc35. Даны векторы $V_1$ и $V_2$; все элементы в каждом векторе различны и являются положительными числами. Получить вектор $V$ пар типа pair<int, int>, в которых первый элемент пары принадлежит вектору $V_1$, а второй представляет собой сумму значений тех элементов вектора $V_2$, которые оканчиваются той же цифрой, что и первый элемент пары (если подходящие элементы в векторе $V_2$ отсутствуют, то второй элемент пары должен быть равен 0). Порядок следования пар должен быть обратным порядку следования элементов вектора $V_1$. При построении вектора $V$ использовать вспомогательное отображение $M$, построенное на основе вектора $V_2$ и сопоставляющее каждой цифре сумму элементов вектора $V_2$, оканчивающихся этой цифрой (по поводу вариантов построения отображения $M$ см. задачи STL5Assoc23–STL5Assoc25; допускается использовать любой вариант).

STL5Assoc36. Даны векторы $V_1$ и $V_2$, элементами которых являются английские слова, набранные заглавными буквами, причем все слова в каждом векторе различны. Получить отображение $M$, в котором ключом является элемент вектора $V_1$, а значением – строка, полученная суммированием следующего набора элементов вектора $V_2$: в набор включаются все слова, оканчивающиеся на ту же букву, что и ключ, за исключением последнего подходящего слова. Порядок слов в наборе должен соответствовать их порядку в векторе $V_2$. При построении строки добавлять после каждого слова пробел. Если требуемый набор является пустым, то строка также должна остаться пустой. При построении отображения $M$ использовать вспомогательное отображение $M_0$, построенное на основе вектора $V_2$ и сопоставляющее каждой букве описанную выше сумму элементов вектора $V_2$, оканчивающихся этой буквой (по поводу вариантов построения отображения $M_0$ см. за-

дачи STL5Assoc26–STL5Assoc28; допускается использовать любой вариант).

## 3.6. Функциональные объекты: дополнительные возможности

Задания группы STL6Func позволяют изучить дополнительные возможности, связанные с функциональными объектами. Эти возможности могут оказаться полезными, если в используемых в программе объектах уже реализованы соответствующие функции-члены и/или необходимые операции.

В основном тексте каждого задания описываются средства, предусмотренные в стандарте C++11. Если эти средства отличаются от тех, которые были включены в предыдущий стандарт, то в примечании указываются их прежние варианты. Следует иметь в виду, что большинство прежних средств, связанных с функциональными объектами, в стандарте C++11 объявлены устаревшими и в дальнейшем будут исключены из библиотеки.

---

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи STL6Func9, приведенным в п. 2.4.2.

---

STL6Func1. Определить *функциональный объект* (*объект-функцию*, *функтор*) – структуру less_abs с операцией () – константной функцией-членом, имеющей два целочисленных параметра $a$ и $b$ и возвращающей результат сравнения $|a| < |b|$. Структура less_abs должна быть порождена от стандартной обобщенной структуры function<bool(int, int)>. Используя функциональный объект less_abs и алгоритм sort, отсортировать исходный вектор $V$ целых чисел по возрастанию их абсолютных значений.

Примечание. Если компилятор не поддерживает стандарт C++11, то в качестве предка структуры less_abs используйте обобщенную структуру binary_function<int, int, bool>.

STL6Func2. Дан вектор $V$ целых чисел. Используя вызов алгоритма adjacent_find, найти начальную пару элементов ($a$, $b$) вектора $V$, для которых выполняется неравенство $|a| \geq |b|$, и вывести найденную пару элементов в порядке возрастания их индексов. Если подходящих пар нет, то вывести единственное число 0. Для поиска требуемой пары использовать *функциональный адаптер* not2 (*инвертор*), применив его к функциональному объекту less_abs, описанному в STL6Func1.

Примечание. Для возможности применения функционального адаптера к функциональному объекту необходимо, чтобы, во-первых, данный объект был порожден от соответствующего базового класса (в данном случае класса function или binary_function) и, во-вторых,

имел реализацию операции () в виде *константной* функции-члена. Для непосредственного применения функционального объекта (без функциональных адаптеров) перечисленные условия не являются обязательными.

STL6Func3. Дано целое число $K$ (> 0) и вектор $V$, содержащий целые числа. Используя функциональный объект less_abs, описанный в STL6Func1, совместно с функциональным адаптером bind (*связывателем*) в алгоритме remove_if, а также метод erase, удалить из вектора $V$ все элементы, абсолютная величина которых меньше $K$. Вывести размер преобразованного вектора $V$ и его элементы. Для возможности использования параметра _1 адаптера bind в программу необходимо добавить директиву using namespace std::placeholders, указав ее после директивы #include <functional>.

Примечание. Если компилятор не поддерживает стандарт C++11, то вместо универсального связывателя bind следует использовать специализированный адаптер-связыватель для второго параметра bind2nd.

STL6Func4. Дано целое число $K$ (> 0) и вектор $V$, содержащий целые числа. Используя функциональный объект less_abs, описанный в STL6Func1, совместно с функциональным адаптером bind в алгоритме find_if, найти и вывести последний элемент вектора, абсолютная величина которого больше $K$. Если вектор не содержит требуемых элементов, то вывести 0.

Примечание. Если компилятор не поддерживает стандарт C++11, то вместо универсального связывателя bind следует использовать специализированный адаптер-связыватель для первого параметра bind1st.

STL6Func5. Дано целое число $K$ (> 0) и вектор $V$, содержащий целые числа. Используя функциональный объект less_abs, описанный в задаче STL6Func1, к которому последовательно применяются инвертор not2 и связыватель bind, построить унарный предикат для проверки условия $K \geq |x|$ и использовать его в качестве последнего параметра алгоритма count_if для нахождения количества элементов вектора, абсолютная величина которых меньше или равна $K$.

Примечания. (1) Если компилятор не поддерживает стандарт C++11, то вместо связывателя bind следует использовать bind1st.

(2) При использовании связывателя bind важен порядок вызова адаптеров: вначале not2, затем bind. Заметим, что старые варианты связывателей (bind1st и bind2nd) можно комбинировать с инверторами в любом порядке.

(3) В данном задании можно обойтись без применения инвертора (обеспечивающего преобразование строгого неравенства в нестрогое), так как для правильного решения достаточно использовать предикат со строгим неравенством вида $K + 1 > |x|$. Однако описанный прием может оказаться полезным в аналогичных ситуациях с более сложными функциональными объектами.

STL6Func6. Даны векторы $V_1$ и $V_2$ с одинаковым количеством элементов – целых чисел. Используя алгоритм transform и стандартный функциональный объект multiplies, преобразовать вектор $V_2$, умножив его элементы на соответствующие элементы вектора $V_1$.

STL6Func7. Дано целое число $K$ и вектор $V$, содержащий целые числа. Используя алгоритм transform и стандартный объект-функцию minus со связывателем bind, преобразовать исходный вектор, уменьшив значения всех его элементов на величину $K$.

Примечание. Если компилятор не поддерживает стандарт C++11, то вместо связывателя bind следует использовать bind2nd.

STL6Func8. Дано целое число $K$ и два вектора $V_1$ и $V_2$ одинакового размера, содержащие целые числа. Используя алгоритм transform и стандартные функциональные объекты plus и multiplies с вложенными связывателями bind, преобразовать вектор $V_1$, умножив каждый его элемент на число $K$ и прибавив к результату значение соответствующего элемента вектора $V_2$.

Примечание. Специализированные связыватели bind1st и bind2nd не позволяют сконструировать требуемый функциональный объект на основе стандартных. Если компилятор не поддерживает стандарт C++11, то решить задачу, не создавая новые функциональные объекты, можно с помощью *двукратного* применения алгоритма transform.

STL6Func9. Определить структуру point с целочисленными членами $x$, $y$ и строковым членом $s$. Предполагается, что строковый член $s$ не содержит пробелов. Отношение порядка для данной структуры определяется следующим образом: $A < B$, если $A.x < B.x$ или ($A.x == B.x$ и $A.y < B.y$). Реализовать это отношение порядка в виде константной функции-члена bool operator<(const point& b) const. Кроме того, реализовать операцию istream& operator>>(istream& is, point& p), которая последовательно считывает из входного потока *is* члены $x$, $y$ и $s$ структуры $p$, и операцию ostream& operator<<(ostream& os, const point& p), которая записывает все члены структуры $p$ в выходной поток *os* (члены записываются в том же порядке, в котором считываются; между ними вставляется пробел). Дан текстовый файл с именем *name*, содержащий текстовые представления элементов описанной выше структуры. Используя итератор

чтения istream_iterator, заполнить этими данными вектор *V* с элементами типа point. Используя алгоритм stable_sort, отсортировать полученные данные с учетом заданного отношения порядка и с помощью алгоритма copy и итератора ostream_iterator записать отсортированный вектор в исходный файл, заменив его прежнее содержимое (при этом точки с одинаковыми координатами будут располагаться в том же порядке, что и в исходном файле, поскольку алгоритм stable_sort обеспечивает устойчивую сортировку). Каждая точка должна отображаться на новой строке.

Примечания. (1) Решение данной задачи приведено в п. 2.4.2.

(2) Операции >> и << позволяют использовать для ввода и вывода объектов типа point стандартные потоковые итераторы чтения и записи.

(3) Благодаря явно определенной для структуры point операции <, в алгоритме сортировки не требуется указывать функциональный объект для сравнения элементов вектора: по умолчанию используется функциональный объект less<point>(), являющийся объектной оберткой для операции <.

(4) Следует заметить, что для сортировки *по убыванию* нельзя использовать (синтаксически допустимый) функциональный объект not2(less<point>()), так как в результате будет получен объект для *нестрогого* сравнения (a >= b), который не может использоваться в алгоритмах, связанных с сортировкой (для него нарушается *условие строгого сравнения*: если a «меньше» b, то b не должно быть одновременно «меньше» a). В подобной ситуации надо использовать объект greater<point>(), предварительно определив для класса point операцию >.

STL6Func10. Дан текстовый файл с именем *name*, содержащий текстовые представления элементов структуры point, реализованной в STL6Func9. Дополнить структуру point, включив в нее оператор преобразования operator string(), возвращающий строку с текстовыми представлениями членов *x*, *y*, *s*, разделенными пробелами. Кроме того, описать логическую функцию-член is_positive(), возвращающую значение true, если числа *x* и *y* являются положительными. Используя алгоритм replace_copy_if совместно с итератором чтения istream_iterator<point> и итератором записи ptout_iterator<string>, заменить те элементы из исходного текстового файла, которые имеют положительные члены *x* и *y*, на объект point (0, 0, "A") и вывести текстовые представления всех элементов преобразованного набора. Вспомогательные контейнеры не применять. В качестве параметра-предиката алгоритма replace_copy_if использовать функцию-член is_positive, преобразовав

ее в функциональный объект с помощью функционального адаптера mem_fn: mem_fn(&point::is_positive).

Примечания. (1) Если компилятор не поддерживает стандарт C++11, то вместо адаптера mem_fn следует использовать его старый вариант mem_fun_ref. Заметим, что адаптер mem_fn может применяться и в том случае, когда алгоритм обрабатывает контейнер, содержащий *указатели* на объекты (в то время как вместо mem_fun_ref в этой ситуации надо использовать другой функциональный адаптер: mem_fun).

(2) При реализации оператора преобразования в строку удобно использовать *строковый поток* ostringstream из заголовочного файла <sstream>.

(3) Если в классе или структуре определен оператор преобразования в строку, то объекты этого класса (структуры) можно указывать в качестве параметров функций Show и ShowLine, обеспечивающих вывод данных в *раздел отладки* окна задачника.

STL6Func11. Дан текстовый файл с именем *name*, содержащий текстовые представления элементов структуры point, реализованной в STL6Func9. Используя вспомогательный вектор *V* с элементами типа point и алгоритм stable_partition, перегруппировать элементы в исходном наборе, переместив в начало набора все элементы, которые меньше точки с координатами (0, 0). Вывести текстовые представления всех элементов преобразованного набора (см. STL6Func10). В качестве параметра – функционального объекта в алгоритме stable_partition использовать объект less<point>(), применив к нему связыватель bind.

Примечание. Если компилятор не поддерживает стандарт C++11, то вместо связывателя bind следует использовать bind2nd.

STL6Func12. Даны текстовые файлы с именами *name1* и *name2*, содержащие текстовые представления элементов структуры point, реализованной в STL6Func9. Файлы содержат одинаковое количество элементов. Определить операцию сложения для объектов point, которая складывает значения соответствующих полей (как числовых, так и строковых), в виде перегруженной функции point operator+(const point& a, const point& b). Прочесть данные из файлов *name1* и *name2* в векторы $V_1$ и $V_2$ соответственно. Используя алгоритм transform с параметром plus<point>(), преобразовать элементы вектора $V_1$, прибавив к ним соответствующие элементы вектора $V_2$. Записать преобразованный вектор $V_1$ в файл *name1*, заменив его прежнее содержимое.

STL6Func13. Дан текстовый файл с именем *name*, содержащий текстовые представления элементов структуры point, реализованной в STL6Func9. Используя вспомогательный вектор *V* с элементами типа point и алгоритм transtorm с функциональным объектом plus (см. STL6Func12) и связывателем bind, преобразовать исходный набор, прибавив к каждому его элементу объект point (10, 20, «Z»). Вывести текстовые представления всех элементов преобразованного набора (см. STL6Func11).

Примечание. Если компилятор не поддерживает стандарт C++11, то вместо связывателя bind следует использовать bind2nd.

STL6Func14. Дано целое число *K* и текстовые файлы с именами *name1* и *name2*, содержащие текстовые представления элементов структуры point, реализованной в STL6Func9. Файлы содержат одинаковое количество элементов. Дополнить структуру point, добавив к ней функцию-член point mult(int k), выполняющую умножение объекта point на число: вызов p.mult(k) возвращает объект типа point, у которого члены *x* и *y* равны соответственно *k**p.x* и *k**p.y*, а член *s* совпадает со строкой *p.s*. Прочесть данные из файлов *name1* и *name2* в векторы $V_1$ и $V_2$. Используя алгоритм transform с подходящим функциональным объектом, содержащим два связывателя bind, объект plus (см. STL6Func12) и функцию-член mult, преобразовать элементы вектора $V_1$, умножив каждый из них на число *K* и прибавив к результату умножения соответствующий элемент вектора $V_2$. Вывести текстовые представления всех элементов преобразованного вектора $V_1$ (см. STL6Func11).

Примечания. (1) Для связывания функции-члена mult с помощью bind достаточно указать в качестве параметра связывателя ссылку &point::mult; адаптер mem_fn (см. STL6Func10) в этом случае применять к ссылке не требуется (хотя и не запрещается).

(2) Специализированные связыватели bind1st и bind2nd не позволяют сконструировать требуемый функциональный объект на основе стандартных (ср. с STL6Func8). Если компилятор не поддерживает стандарт C++11, то решить задачу, не создавая новые функциональные объекты, можно с помощью *двукратного* применения алгоритма transform. При этом для выполнения умножения элементов первого вектора на *K* необходимо использовать связыватель bind2nd для объекта mem_fun_ref(&point::mult) (использование адаптера mem_fun_ref в данном случае является обязательным).

## 3.7. Применение различных средств стандартной библиотеки C++

В каждом задании даются имена одного или нескольких текстовых файлов, содержащих исходные последовательности, а также имя текстового файла, в который требуется записать результаты обработки исходных последовательностей (имя результирующего файла указывается последним). Каждая исходная последовательность содержится в отдельном файле. Все исходные файлы содержат текст в кодировке «windows-1251»; эта же кодировка должна использоваться при записи полученных данных в результирующий файл.

Каждый элемент последовательности размещается в отдельной строке файла, в начале и конце строки пробелы отсутствуют, поля элемента не содержат пробелов и разделяются ровно одним пробелом. Все исходные числовые данные являются положительными. В качестве десятичного разделителя используется *точка*.

Если в задание входят дополнительные числовые или строковые исходные данные, то они указываются в начале набора исходных данных (перед именами файлов).

Во всех заданиях данной группы заготовки решения уже содержат определения исходных структур данных (в заданиях STL7Mix1–STL7Mix70 структура имеет имя Data, в заданиях STL7Mix71–STL7Mix100 имя дополнительно содержит буквенный идентификатор соответствующего набора данных, например, DataA или DataC). Кроме того, заготовки содержат операторы, обеспечивающие чтение данных из исходных файлов в векторы типа vector<Data>, где Data – ранее определенная структура данных, и операторы, создающие результирующий файл.

Например, заготовка для задания STL7Mix1 включает определение структуры Data с целочисленными полями code, year, month, len, а функция Solve содержит следующие операторы:

```
string name1, name2;
pt >> name1 >> name2;
ifstream f1(name1);
vector<Data> V((istream_iterator<Data>(f1)),
  istream_iterator<Data>());
f1.close();
ShowLine(V.begin(), V.end(), "V: ");

ofstream f2(name2);

f2.close();
```

Оператор отладочной печати отображает в разделе отладки все введенные исходные данные. Выводить полученные данные следует в файловый поток f2.

Если компилятор поддерживает стандарт C++11, то для вывода результатов рекомендуется использовать цикл for(auto& e : C), где C – контейнер с результирующими данными; в противном случае можно использовать алгоритм for_each для этого контейнера.

При решении большинства задач требуется выполнить *группировку* исходных данных, используя вспомогательное отображение подходящего типа. В задачах STL7Mix71–STL7Mix100 может оказаться удобным предварительно выполнить внутреннее или внешнее *объединение* данных по общему ключу. Группировка и объединение данных с использованием ассоциативных контейнеров рассматриваются в группе заданий STL5Assoc.

Если порядок вывода сгруппированных данных отличается от порядка следования их ключей, то необходимо скопировать полученные данные в последовательный контейнер (обычно вектор) и выполнить для него требуемую сортировку (используя алгоритмы sort или stable_sort).

---

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи STL7Mix4, приведенным в п. 2.5.

---

### 3.7.1. Избранные задачи из группы STL7Mix

По причинам, связанным с охраной авторских прав, в настоящем издании не приводятся полные тексты всех 100 задач группы STL7Mix, поскольку, как отмечалось выше (см. преамбулу к разделу 3), они ранее были включены в книгу [1]. В данном пункте содержатся формулировки 7 задач, позволяющие получить представление об особенностях заданий, входящих в эту группу. В полном объеме задания группы STL7Mix доступны на сайте задачника http://ptaskbook.com/ (раздел «PT for STL»). Указания к ряду задач группы STL7Mix приведены в п. 3.7.2.

STL7Mix1. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Код клиента> <Год> <Номер месяца> <Продолжительность занятий (в часах)>

Найти элемент последовательности с минимальной продолжительностью занятий. Вывести эту продолжительность, а также соответствующие ей год и номер месяца (в указанном порядке на той же строке). Если имеется несколько элементов с минимальной продолжительностью, то вывести данные того из них, который является последним в исходной последовательности.

STL7Mix4. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Год> <Номер месяца> <Продолжительность занятий (в часах)> <Код клиента>

Для каждого клиента, присутствующего в исходных данных, определить суммарную продолжительность занятий в течение всех лет (вначале выводить суммарную продолжительность, затем код клиента). Сведения о каждом клиенте выводить на новой строке и упорядочивать по убыванию суммарной продолжительности, а при их равенстве – по возрастанию кода клиента.

Примечание. Решение данной задачи приведено в п. 2.5.

STL7Mix13. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Номер школы> <Год поступления> <Фамилия>

Для каждого года, присутствующего в исходных данных, найти школу с наибольшим номером среди школ, которые окончили абитуриенты, поступившие в этом году, и вывести год и найденный номер школы. Сведения о каждом годе выводить на новой строке и упорядочивать по возрастанию номера года.

STL7Mix25. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Фамилия> <Задолженность> <Номер квартиры>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом этаже располагаются по 4 квартиры. Найти номер подъезда, жильцы которого имеют наибольшую суммарную задолженность, и вывести этот номер вместе с размером суммарной задолженности (выводится с двумя дробными знаками). Считать, что суммарные задолженности для всех подъездов имеют различные значения.

STL7Mix37. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

<Компания> <Марка бензина> <Цена 1 литра (в копейках)> <Улица>

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет

не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой марки бензина, присутствующей в исходных данных, определить минимальную и максимальную цену литра бензина этой марки (вначале выводить марку, затем цены в указанном порядке). Сведения о каждой марке выводить на новой строке и упорядочивать по убыванию значения марки.

STL7Mix49. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Фамилия> <Инициалы> <Номер школы> <Баллы ЕГЭ>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Определить наименьший суммарный балл и вывести его. Вывести также сведения обо всех учащихся, получивших наименьший суммарный балл (для каждого учащегося указывать фамилию, инициалы и номер школы). Сведения о каждом учащемся выводить на отдельной строке и располагать в порядке их следования в исходном наборе.

STL7Mix61. Исходная последовательность содержит сведения об оценках учащихся по трем предметам: алгебре, геометрии и информатике. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

<Фамилия> <Инициалы> <Класс> <Название предмета> <Оценка>

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Для каждого учащегося определить среднюю оценку по каждому предмету и вывести ее с двумя дробными знаками (если по какому-либо предмету учащийся не получил ни одной оценки, то вывести для этого предмета 0.00). Сведения о каждом учащемся выводить на отдельной строке, указывая фамилию, инициалы и средние оценки по алгебре, геометрии и информатике. Данные располагать в алфавитном порядке фамилий и инициалов.

### 3.7.2. Указания к задачам группы STL7Mix

STL7Mix1. Используйте алгоритм min_element с предикатом.

STL7Mix2. Ср. с STL7Mix1. Для нахождения требуемого элемента используйте предикат, анализирующий поля «длительность», «год», «месяц».

STL7Mix3. Ср. с STL7Mix1. Для группировки по полю «год» используйте вспомогательное отображение.

STL7Mix4. Для группировки по полю «код» используйте вспомогательное отображение.

STL7Mix5. Ср. с STL7Mix4.

STL7Mix6. Ср. с STL7Mix4. В данном случае вместо вспомогательного отображения можно использовать вектор с 12 элементами, соответствующими номерам месяцев (при этом элементами вектора должны быть пары «длительность, номер месяца»).

STL7Mix7. Для отбора данных, связанных с клиентом *K*, и их одновременной группировки по полю «год» используйте вспомогательное отображение *M* типа map<int, Data*, greater<int>> (при этом функциональный объект greater обеспечит требуемую сортировку даных по убыванию номера года). Используя алгоритм for_each или цикл for по элементам контейнера, перебирайте элементы исходного набора данных, добавляя (или изменяя) элементы отображения *M* в соответствии с условиями задачи.

STL7Mix8. Ср. с STL7Mix7. В данном случае после построения отображения *M* следует скопировать его во вспомогательный вектор и отсортировать этот вектор в требуемом порядке. Если в отображении *M* использовать порядок по умолчанию (т. е. по возрастанию ключа «год»), то для сортировки вспомогательного вектора достаточно применить алгоритм устойчивой сортировки по возрастанию поля «длительность».

STL7Mix9. Ср. с STL7Mix8. В данном случае достаточно использовать отображение *M* типа map<int, int>.

STL7Mix10. Для группировки по составному ключу «год, номер месяца» используйте вспомогательное отображение *M* типа map<pair<int, int>, int, greater_less>, предварительно описав функциональный объект greater_less для реализации варианта сравнения пары ключей, описанного в задаче. Благодаря этому объекту полученное отображение будет содержать данные в требуемом порядке.

STL7Mix11. Ср. с STL7Mix10. В данном случае дополнительно потребуется сортировка полученных данных по возрастанию общей продолжительности занятий. Необходимо использовать алгоритм устойчивой сортировки, что обеспечит требуемый порядок пар с одинаковой общей продолжительностью (при условии использования отображения с функциональным объектом greater_less, описанным в указании к STL7Mix10).

STL7Mix12. Для группировки данных по полю «год» используйте отображение *M* типа map<int, vector<int>>, включая в вектор значений (длины 13) не только данные о каждом месяце (сохраняя их в элементах вектора с индексами от 1 до 12), но и суммарные данные по всему году (сохраняя их в элементе вектора с индексом 0). После формирования отображения *M* постройте на основе полученных данных вспомогательный вектор типа vector<pair<int, int>> и отсортируйте элементы вектора требуемым образом, используя алгоритм stable_sort.

STL7Mix13. Используйте группировку по полю «год».

STL7Mix14. Выполните группировку по полю «год», сохранив полученную последовательность во вспомогательном отображении. Используя алгоритм max_element, найдите наибольшее число абитуриентов, после чего воспользуйтесь алгоритмом count_if, чтобы определить количество лет, для которых число абитуриентов было максимальным.

STL7Mix15. Ср. с STL7Mix14.

STL7Mix16. Ср. с STL7Mix14. Для упорядочивания полученных сгруппированных данных по убыванию числа поступивших используйте вспомогательный вектор и алгоритм stable_sort.

STL7Mix17. В данном случае для группировки по полю «год» удобно использовать отображение map<int, set<int>>.

STL7Mix18. Ср. с STL7Mix16.

STL7Mix19. В данном случае для группировки по полю «номер школы» можно использовать отображение map<int, pair<int, string>>.

STL7Mix20. Ср. с STL7Mix14. После определения максимального количества абитуриентов с помощью первого вспомогательного отображения (типа map<int, int>) сформируйте второе отображение (типа map<int, vector<string>>), содержащее данные только о тех школах, количество абитуриентов для которых было максимальным.

STL7Mix21. Ср. с STL7Mix20. В данном случае можно использовать второе вспомогательное отображение типа map<int, string>, а также (для сортировки в нужном порядке) вспомогательный вектор типа vector<pair<int, string>>.

STL7Mix22. Здесь, как и в задании STL7Mix17, удобно использовать вспомогательное отображение map<int, set<int>>, позволяющее сразу получить результирующие данные в требумом порядке.

STL7Mix23. Ср. с STL7Mix10.

STL7Mix24. Ср. с STL7Mix10. В данном случае вспомогательный функциональный объект целесообразно назвать less_greater, а в качестве типа вспомогательного отображения использовать map<pair<int, int>, vector<string>, less_greater>.

STL7Mix25. Выполните группировку по номеру подъезда, используя следующую формулу нахождения номера подъезда entrance по номеру квартиры num: entrance = (num – 1) / 36 + 1 (операция «/» обозначает деление нацело). К полученному в результате группировки отображению примените алгоритм max_element. Вещественные результаты выводите с двумя дробными знаками.

STL7Mix26. Ср. с STL7Mix25. При выполнении группировки используйте отображение типа map<int, pair<int, double>>.

STL7Mix27. Ср. с STL7Mix26. Используйте формулу нахождения номера этажа floor по номеру квартиры num: floor = (num – 1) % 36 / 4 + 1 (операция «/» обозначает деление нацело, операция «%» – взятие остатка от деления нацело).

STL7Mix28. Ср. с STL7Mix27. В данном случае отображение, полученное в результате группировки по этажам, должно включать элементы, соответствующие всем этажам дома (при этом некоторые элементы могут содержать нулевые значения).

STL7Mix29. Ср. с STL7Mix25. При группировке можно использовать отображение типа map<int, vector<Data*>>, где Data – тип элемента из исходного набора данных.

STL7Mix30. Ср. с STL7Mix27 и STL7Mix29.

STL7Mix31. Ср. с STL7Mix29. Для отбора трех задолжников используйте алгоритм partial_sort.

STL7Mix32. Ср. с STL7Mix28. В данном случае достаточно использовать вспомогательный массив вещественных чисел из 9 элементов (соответствующих всем этажам дома).

STL7Mix33. Используйте алгоритм remove_copy_if для копирования требуемых элементов во вспомогательный массив. Для того чтобы при этом избежать многократного вычисления средней задолженности по дому, предварительно сохраните ее значение (найденное, например, с помощью алгоритма accumulate) во вспомогательной переменной.

STL7Mix34. Ср. с STL7Mix33.

STL7Mix35. См. указания к STL7Mix25 и STL7Mix33. В данном случае для хранения средних значений удобно использовать вспомогательный вектор размера 4.

STL7Mix36. Ср. с STL7Mix35; см. также указание к STL7Mix27.

STL7Mix39. При выполнении группировки по полю «улица» используйте отображение типа map<string, int>.

STL7Mix40. При выполнении группировки по полю «улица» используйте отображение типа map<string, vector<int>> (вектор должен иметь длину 3 и хранить информацию о количестве АЗС, предлагавших определенную марку бензина). Заметим, что индекс элемента вектора можно определить по значению марки m следующим образом: (m – 92)/3.

STL7Mix42. Ср. с STL7Mix40.

STL7Mix43. Ср. с STL7Mix41. Отличие от STL7Mix41 состоит в том, что в данном случае полученный набор должен содержать сведения обо всех компаниях (а не только тех, которые имеют АЗС, предлагающие бензин марки *M*). Можно, например, использовать для группировки по полю «компания» отображение типа map<string, vector<int>>, в котором для каждой компании хранятся все ее цены на бензин марки *M*, а затем на основе данного отображения сформировать вектор пар pair<string, int>, содержащий для каждой компании разброс цен, и отсортировать его в требуемом порядке. Для нахождения разброса цен можно использовать алгоритмы max_element и min_element.

STL7Mix44. Ср. с STL7Mix43. В данном случае при группировке по полю «компания» можно использовать отображение типа map<string, multimap<int, int>>, в котором для каждой компании хранятся все ее цены, сгруппированные по маркам бензина.

STL7Mix45. При группировке по полю «улица» используйте отображение типа map<string, set<string>>, в котором для каждой улицы хранится множество компаний, имеющих АЗС на этой улице.

STL7Mix46. Ср. с STL7Mix45. В данном случае при группировке по полю «компания» можно использовать отображение типа map<string, map<string, int>>, в котором с каждой компанией связывается вспомогательное отображение, содержащее в качестве ключей названия улиц, а в качестве значений – количество марок бензина, предлагавшихся на АЗС данной компании, расположенной на данной улице.

STL7Mix47. Выполните группировку по ключу, полученному в результате сцепления полей «компания» и «улица» (с пробелом между ними).

STL7Mix48. Предварительно сформируйте вспомогательное отображение *M* типа map<string, int>, содержащее в качестве ключей названия всех компаний, входящих в исходный набор данных (значения оставьте нулевыми). После этого выполните группировку по полю «улица»,

используя отображение типа map<string, map<string, int>>. При добавлении в отображение новой улицы сразу связывайте с ней построенное ранее вспомогательное отображение *M* и используйте его значения в качестве счетчика марок бензина, предлагавшихся на АЗС данной компании, расположенной на данной улице.

STL7Mix49. Для нахождения минимального суммарного балла используйте алгоритм min_element.

STL7Mix50. Ср. с STL7Mix49. Для нахождения двух наибольших суммарных баллов достаточно сформировать вспомогательное множество суммарных баллов типа set<int, greater<int>> и извлечь из него два первых элемента.

STL7Mix51. Используйте вспомогательное отображение типа map<int, Data*>.

STL7Mix52. Ср. с STL7Mix51.

STL7Mix53. Используйте вспомогательное отображение типа map<int, int> для группировки исходных данных и вектор типа vector<pair<int, int>> совместно с алгоритмом stable_sort для упорядочивания полученных результатов.

STL7Mix54. Ср. с STL7Mix53. В данном случае в качестве типа отображения для группировки можно использовать map<int, pair<int, int>>.

STL7Mix55. Используйте алгоритм remove_if.

STL7Mix56. Ср. с STL7Mix55.

STL7Mix57. Используйте вспомогательное отображение типа map<int, multiset<string>> для группировки исходных данных.

STL7Mix58. Ср. с STL7Mix57.

STL7Mix59. Для группировки исходных данных используйте вспомогательное отображение типа map<int, vector<int>>.

STL7Mix60. Ср. с STL7Mix59.

STL7Mix61. Для группировки исходных данных используйте вспомогательное отображение типа map<string, map<string, pair<double, int>>>. В паре pair<double, int> накапливайте суммарную оценку (в первом элементе) и количество оценок (во втором элементе); после обработки всех исходных данных достаточно поделить первый элемент на второй и сохранить полученное среднее значение в первом элементе пары (если второй элемент равен 0, то деление выполнять не следует).

STL7Mix62. Для группировки исходных данных используйте вспомогательное отображение типа map<int, map<string, map<string, int>>>.

STL7Mix63. Для группировки исходных данных используйте вспомогательное отображение типа map<pair<string, int>, pair<double, int>>. По поводу пары pair<double, int> см. указание к STL7Mix61.

STL7Mix64. Ср. с STL7Mix63. В данном случае для группировки исходных данных можно использовать вспомогательное отображение типа map<pair<int, string>, pair<double, int>>.

STL7Mix65. Для группировки исходных данных используйте вспомогательное отображение типа map<int, map<string, pair<double, int>>, greater<int>>, после чего постройте на его основе вектор типа vector<pair<int, int>> и отсортируйте его требуемым образом. По поводу пары pair<double, int> см. указание к STL7Mix61.

STL7Mix66. Ср. с STL7Mix65. Если вспомогательное отображение описать как map<int, map<string, pair<double, int>>>, то для построенного на его основе вспомогательного вектора не потребуется выполнять дополнительной сортировки.

STL7Mix67. Организуйте сохранение данных о двоечниках во вспомогательном отображении типа map<pair<int, string>, int, greater_less>, предварительно описав функциональный объект greater_less для реализации варианта сравнения пары ключей, описанного в задаче (номера классов должны упорядочиваться по убыванию, а фамилии в пределах класса – в алфавитном порядке).

STL7Mix68. Организуйте сохранение данных о двоечниках во вспомогательном отображении типа map<pair<string, int>, pair<int, int>>, сохраняя в первом компоненте пары ключей фамилию, а во втором – номер класса, и накапливая в первом компоненте пары значений количество четверок, полученных учеником, а во втором – количество двоек и троек. После завершения обработки исходных данных останется преобразовать полученное отображение в вектор типа vector<pair<pair<string, int>, pair<int, int>>>, удалить в нем сведения об учениках, имеющих ненулевое число двоек и троек, и отсортировать его алгоритмом stable_sort по возрастанию числа полученных четверок.

STL7Mix69. Для группировки данных используйте отображение типа map<int, map<string, int>> (данные группируются по классам, в каждый класс добавляются только двоечники, причем для каждого двоечника сохраняется количество его двоек). Затем, используя алгоритм max_element для каждого класса, постройте отображение, в котором для каждого класса хранится максимальное число двоек, полученных его учениками, и с помощью этого отображения постройте итоговое отображение, содержащее данные обо всех учащихся, получивших в каждом классе максимальное число двоек (если ключами в этом

отображении будут фамилии, то данные будут автоматически отсортированы требуемым в задаче способом).

STL7Mix70. Ср. с STL7Mix69. В данном случае в ходе начальной группировки необходимо сохранять не только количество пятерок, но также количество полученных двоек и троек. Затем надо удалить учащихся, получивших двойки и тройки, после чего действовать как при решении STL7Mix69 с учетом того, что сортировать итоговые данные требуется в другом порядке.

# Раздел 4. Приложение. Средства ввода-вывода, реализованные в задачнике Programming Taskbook

## *4.1. Поток ввода-вывода pt и связанные с ним итераторы*

Основным средством ввода-вывода данных при выполнении заданий на языке C++ с применением задачника Programming Taskbook является *поток ввода-вывода* pt, определенный в заголовочном файле pt4.h. С его помощью можно выполнять ввод и вывод данных всех стандартных типов, используемых в заданиях: bool, int, double, char, string. В заголовочном файле pt4.h определены также функции, предназначенные для ввода и вывода отдельных элементов, однако использование потока pt является более удобным способом организации ввода-вывода.

Для ввода или вывода *последовательностей* с элементами типа T предусмотрены итераторы ptin_iterator<T> и ptout_iterator<T>, связанные с потоком pt. Эти итераторы обладают теми же свойствами, что и стандартные потоковые итераторы (см. п. 1.1.2).

У итератора ptin_iterator<T> имеются два конструктора. Конструктор без параметров создает итератор конца последовательности. Конструктор с параметром count типа unsigned int создает итератор чтения, который позволяет прочесть count элементов типа T из потока pt, после чего переходит в состояние конца последовательности (т. е. становится равным итератору ptin_iterator<T>()).

В случае особого значения параметра count, равного 0, размер последовательности считывается из самого потока pt непосредственно перед считыванием первого элемента последовательности. Если при считывании размера оказывается, что прочитанный элемент данных не является целым числом или это число не является положительным, то итератор сразу переходит в состояние конца последовательности.

Подобная организация итераторов чтения ptin_iterator<T> позволяет легко реализовать считывание из потока ввода pt нескольких последовательностей, если заранее известен их размер или (как во всех заданиях, входящих в задачник Programming Taskbook for STL) если размер указывается непосредственно перед началом последовательности.

Итератор ptout_iterator<T> создается с помощью конструктора без параметров и позволяет записать в поток pt произвольное количество данных типа T.

## 4.2. Вывод отладочной информации: функции Show и ShowLine

Для вывода информации в раздел отладки окна задачника в заголовочном файле pt4.h предусмотрен набор перегруженных вариантов функций Show и ShowLine.

Простейшим вариантом является функция ShowLine() без параметров, позволяющая перейти в разделе отладки на новую строку.

Для вывода элемента a одного из базовых типов bool, int, double, char, string, а также типа pair<T1, T2> предусмотрены варианты функций Show и ShowLine с параметром a и с двумя параметрами (s, a), где дополнительный параметр s определяет строковый комментарий, который выводится перед элементом a.

При выводе числовых элементов можно также указать третий параметр w целого типа, который определяет *ширину поля вывода* (т. е. количество экранных позиций, отводимое для вывода числа). Если указанной ширины w поля вывода недостаточно, то значение параметра w игнорируется; в этом случае (а также в случае, если параметр w отсутствует) используется ширина поля вывода, минимально необходимая для отображения данного числа. Если число не занимает всего поля вывода, то оно дополняется *слева* пробелами (т. е. выравнивается по правой границе поля вывода). В качестве десятичного разделителя для чисел с дробной частью используется точка.

Вещественные числа по умолчанию выводятся в формате с фиксированной точкой и двумя дробными знаками. Изменить формат вывода вещественных чисел можно с помощью вспомогательной функции SetPrecision, описываемой в п. 4.3.

Элементы типа pair<T1, T2> при выводе заключаются в круглые скобки, а их члены first и second разделяются запятой. В качестве типов T1 и T2 могут использоваться типы int, double, char, string, а также типы pair (глубина вложения типов pair может быть произвольной) и контейнеры с данными указанных типов. Например, элемент данных типа pair<int, pair<int, int>> с членами 5, 2 и 9 и комментарием «p=» будет выведен следующим образом:

```
p=( 5 , ( 2 , 9 ) )
```

Для вывода элементов последовательностей предусмотрены шаблонные варианты функций Show и ShowLine с параметрами-итераторами:

```
template<typename InIter>
    void Show(InIter first, InIter last[, string s])
template<typename InIter>
    void ShowLine(InIter first, InIter last[, string s])
```

Итераторы first и last, как обычно, определяют начальный выводимый элемент и позицию за конечным выводимым элементом. Элементы последовательности могут иметь тип int, double, char и string, а также pair<T1, T2> (глубина вложения типов pair может быть произвольной); кроме того, сами элементы (или компоненты пары pair), в свою очередь, могут быть последовательностями. Необязательный последний параметр s позволяет задать строковый комментарий, который указывается перед выводимой последовательностью.

Шаблонный вариант Show выводит элементы последовательности на одной строке, разделяя их пробелами; после завершения вывода автоматически выполняется переход на новую строку. Шаблонный вариант ShowLine выводит каждый элемент последовательности на новой строке; строковый комментарий выводится в той же строке, что и первый элемент.

**Примечание 1.** Шаблонные варианты функций Show и ShowLine могут использоваться, в частности, для вывода текстовых строк, однако при этом символы выводимой строки будут разделяться пробелом (при использовании функции Show) или выводиться на отдельных строках (при использовании функции ShowLine). Приведем варианты вывода строки s, содержащей текст «Пример»:

```
string s("Пример");
ShowLine("s1=", s);
Show(s.begin(), s.end(), "s2=");
ShowLine(s.begin(), s.end(), "s3=");
```

В результате выполнения этих операторов в разделе отладки будет выведен следующий текст:

```
1>  s1=Пример
2>  s2=П р и м е р
3>  s3=П
4>  р
5>  и
6>  м
7>  е
8>  р
```

**Примечание 2.** Вызов функций Show и ShowLine с двумя параметрами типа char* обеспечивает вывод строки с указанным начальным и конечным итератором, поэтому в программе *не следует вызывать эти функции с двумя параметрами – литеральными строками*:

```
Show("s=", "Пример"); // Приведет к неверной работе программы!
```

При подобном вызове в разделе отладки отображаются посторонние данные, а программа обычно завершается ошибкой Access Violation.

Для правильного вывода данных следует либо объединить эти параметры в одну литеральную строку, либо преобразовать хотя бы один из них к типу string:

```
Show("s=Пример");
Show("s=", string("Пример"));
```

## 4.3. Вывод отладочной информации: вспомогательные функции

```
void HideTask();
```

Вызов функции HideTask обеспечивает автоматическое скрытие всех разделов окна задачника, кроме раздела отладки. Если раздел отладки в окне задачника не отображается (в частности, если программа запущена в демонстрационном режиме), то вызов функции HideTask игнорируется. Игнорируются также все повторные вызовы данной функции.

Скрыть/восстановить основные разделы окна задачника после его отображения на экране можно также с помощью клавиши пробела или соответствующей команды контекстного меню.

```
void SetPrecision(int n);
```

Функция SetPrecision предназначена для настройки формата вывода *вещественных* отладочных данных. Если параметр n положителен, то он определяет количество выводимых дробных разрядов; при этом число выводится в формате с фиксированной точкой. Если параметр n равен нулю или является отрицательным, то число выводится в формате с плавающей точкой (экспоненциальном формате); при этом число дробных знаков полагается равным *модулю* ненулевого параметра n или, если параметр n равен нулю, устанавливается количество дробных знаков, равное 6.

Действие текущей настройки числового формата, определенной функцией SetPrecision, продолжается до очередного вызова этой функции. До первого вызова функции SetPrecision вещественные числа выводятся в формате с фиксированной точкой и двумя дробными знаками.

# Литература


1. *Абрамян М. Э.* Технология LINQ на примерах. Практикум с использованием электронного задачника Programming Taskbook for LINQ. – М.: ДМК Пресс, 2014. – 326 с.
2. *Аммерааль Л.* STL для программистов на C++. – М.: ДМК Пресс, 1999. – 240 с.
3. *Вандервурд Д., Джосаттис Н. М.* Шаблоны C++: справочник разработчика. – М.: Издательский дом «Вильямс», 2003. – 544 с.
4. *Джосаттис Н. М.* Стандартная библиотека C++: справочное руководство. 2-е изд. – М.: Издательский дом «Вильямс», 2014. – 1136 с.
5. *Липпман С. Б., Лажойе Ж., Му Б. Э.* Язык программирования C++. Базовый курс. 5-е изд. – М.: Издательский дом «Вильямс», 2014. – 1120 с.
6. *Мейерс С.* Эффективное использование STL. – СПб.: Питер, 2002. – 226 с.
7. *Мюссер Д. Р., Дердж Ж. Дж., Сейни А.* C++ и STL: справочное руководство. 2-е изд. – М.: Издательский дом «Вильямс», 2010. – 432 с.
8. *Прата С.* Язык программирования C++. Лекции и упражнения. 6-е изд. – М.: Издательский дом «Вильямс», 2012. – 1248 с.