Федеральное агентство по образованию

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

М. Э. Абрамян, В. Н. Брагилевский

Язык программирования C++

Часть 2

Массивы, строки, файлы

Для студентов 1 курса
факультета математики, механики и компьютерных наук

Ростов-на-Дону 2008

Публикуется в электронном виде
в централизованной библиотеке учебно-методических ресурсов
Южного федерального университета
по решению кафедры алгебры и дискретной математики
факультета математики, механики и компьютерных наук ЮФУ
от 17 мая 2010 г. (протокол № 10)

Рецензенты:
к. ф.-м. н., доцент Михалкович С. С.,
к. ф-м. н., доцент Чечин Г. М.

Аннотация

Учебное пособие является вторым в серии из трех пособий, посвященных языку программирования C++. Оно разбито на 4 модуля, включающих как основное содержание, так и контрольные разделы (проектное задание и тесты рубежного контроля), позволяющие определить уровень усвоения материала. Каждый из модулей посвящен определенной теме, связанной с обработкой сложных структур данных: массивов (модуль № 1), символьных строк (модуль № 2), двоичных файлов (модуль № 3) и текстовых файлов (модуль № 4).

Пособие предназначено для студентов факультета математики, механики и компьютерных наук (направления «Прикладная математика», «Информационные технологии»).

Авторы: М. Э. Абрамян, В. Н. Брагилевский.

Введение

Учебное пособие является вторым в серии из трех пособий, посвященных языку программирования C++ [1–2]. В нем рассматриваются вопросы, связанные с обработкой сложных структур данных: массивов, строк и файлов.

В соответствии с *модульной технологией* организации учебного материала (см. [7]) пособие разбито на 4 модуля, включающих как основное содержание, так и контрольные разделы (проектное задание и тесты рубежного контроля), позволяющие определить уровень усвоения материала. Перечислим темы, рассмотренные в пособии: массивы (модуль № 1), символьные строки (модуль № 2), двоичные файлы (модуль № 3), текстовые файлы (модуль № 4). Помимо базовых сведений о различных возможностях языка, настоящее пособие содержит решения большого числа типовых задач. Кроме того, в пособии приводятся указания более чем к 300 задачам, выполнение которых позволит закрепить изученный материал (формулировки задач в пособие не включены, так как они ранее были опубликованы в методических указаниях [3–4]). Наличие большого количества учебных задач позволило представить проектные задания к каждому модулю в виде набора из 24 вариантов, что дает возможность преподавателю снабдить каждого студента отдельным вариантом проектного задания.

В приложении А приводится дополнительный справочный материал по языку C++, посвященный работе с массивами, символьными строками и файловыми потоками.

Все задачи, рассмотренные в настоящем учебном пособии, включены в электронный задачник по программированию Programming Taskbook. Использование электронного задачника существенно ускоряет процесс выполнения заданий, так как избавляет учащегося от дополнительных усилий по организации

ввода/вывода, а также позволяет отобразить исходные и результирующие данные в наглядном виде, что является особенно удобным при работе со сложными структурами данных. Предоставляя учащемуся готовые исходные данные, задачник акцентирует его внимание на разработке и программной реализации *алгоритма* решения задания, причем разнообразие исходных данных обеспечивает надежное *тестирование* предложенного алгоритма.

При описании решений типовых задач используются возможности, предоставляемые электронным задачником Programming Taskbook (в частности, применяется специальный поток *pt* для ввода/вывода данных). Благодаря этим возможностям тексты программ с решениями удалось сделать более компактными и наглядными. Средства задачника Programming Taskbook, используемые в настоящем пособии, описаны в приложении В пособия [1].

Модули № 1 и 2 разработаны М. Э. Абрамяном, модули № 3 и 4 — В. Н. Брагилевским.

1. Модуль № 1. Массивы

1.1. Комплексная цель

Ознакомиться с массивами и особенностями их использования в C++. Изучить основные алгоритмы обработки массивов, в том числе алгоритмы, связанные с перебором элементов, их добавлением, удалением, сортировкой, а также слиянием отсортированных массивов.

1.2. Содержание

1.2.1. Ввод и вывод элементов массива: Array7

Array7. Дан массив размера *N*. Вывести его элементы в обратном порядке.

Данное задание является очень простым. Его решение приводится лишь для того, чтобы показать, как описываются массивы, и организуется ввод и вывод их элементов. Как и в примерах решений первого пособия [1] настоящей серии, будем приводить только операторы, входящие в функцию *solve*.

```
task("Array7");
int n;
pt >> n;
double a[10];
for (int i = 0; i < n; ++i)
    pt >> a[i];
for (int i = 0; i < n; ++i)
    pt << a[n - i - 1];
```

В C++ при описании массива указывается его *размер*, то есть число его элементов, причем элементы индексируются от 0 (таким образом, если размер

массива равен 10, то его элементы имеют индексы от 0 до 9). Для доступа к элементу массива *a* с индексом *i* используется операция индексирования []: *a*[*i*].

При таком способе описания массива в качестве его размера можно указывать только *константу*. Так как в задачнике Programming Taskbook предполагается, что размер исходных массивов не превосходит 10, мы указали при описании массива этот максимально возможный размер; при этом переменная *n* определяет фактическую «заполненность» массива, то есть количество элементов, которые содержат данные (содержимое прочих элементов массива является неопределенным).

Подробности, связанные с описанием и инициализацией массивов, см. в п. 5.1.

При выводе элементов массива в обратном порядке можно было бы также воспользоваться циклом с *убывающим* параметром:

```
for (int i = n - 1; i >= 0; --i)
    pt << a[i];
```

Следует, однако, иметь в виду, что в программах нередко приходится организовывать перебор элементов в обратном порядке в цикле с *возрастающим* параметром (см., например, второй вариант решения задания String10 в п. 2.2.1).

☑ В C++ можно также использовать массивы, размер которых неизвестен до начала выполнения программы (такие массивы называются *динамическими*). Для создания динамического массива следует описать *указатель* на тип, являющийся типом элементов массива, после чего выделить для этого указателя память требуемого размера, используя операцию выделения памяти *new*. В нашем случае массив *a* размера *n* можно было бы создать следующим образом:

```
double* a = new double[n];
```

Символ * показывает, что переменная *a* является *указателем* на данные типа *double*, а квадратные скобки [] означают, что должна быть выделена память не для одной, а для *n* переменных типа *double* (то есть для массива). Разумеется, к моменту выполнения этого оператора значение переменной *n* должно быть определено.

Все прочие операторы приведенного выше решения изменять не требуется. Таким образом, данный вариант позволяет реализовать универсальный алгоритм решения задания для массива *произвольного* размера.

Если память для некоторого элемента данных (в частности, массива) выделена с помощью операции *new*, то после завершения работы с этим элементом данных необходимо освободить выделенную память, используя операцию *delete*. Причем, если при выделении памяти использовался вариант операции *new* с квадратными скобками, то для освобождения памяти надо использовать аналогичный вариант операции *delete*: *delete*[]. Так, в нашем случае в конец функции *solve* надо добавить оператор:

```
delete[] a;
```

Более подробно операции над указателями описываются в [2] (см. модули № 2–3, а также п. 5). См. также п. 5.1–5.2 настоящего пособия.

1.2.2. Анализ элементов массива: Array47

Array47. Дан целочисленный массив размера *N*. Найти количество различных элементов в данном массиве.

```
task("Array47");
int n, k = 1;
pt >> n;
int a[10];
for (int i = 0; i < n; ++i)
    pt >> a[i];
for (int i = 1; i < n; ++i)
{
    int k0 = 1;
    for (int j = 0; j < i; ++j)
        if (a[j] == a[i])
        {
            k0 = 0;
            break;
```

```
            }
        k += k0;
    }
    pt << k;
```

Количество различных элементов массива *a* подсчитывается в переменной *k*. При анализе элемента *a*[*i*] организуется просмотр всех предыдущих элементов (с индексами от 0 до $i - 1$), и если обнаруживается элемент, совпадающий с *a*[*i*], то просмотр немедленно прекращается, и значение вспомогательной переменной *k0* сбрасывается в 0. Если элементы, предшествующие элементу *a*[*i*], не совпадают с ним, то переменная *k0* сохраняет начальное значение, равное 1, которое добавляется к переменной *k*. Поскольку переменная *k0* используется только внутри цикла по *i*, она описана и инициализирована именно в этом цикле.

1.2.3. Преобразование массива: Array79

Array79. Дан массив размера *N*. Осуществить *сдвиг* элементов массива вправо на одну позицию (при этом A_1 перейдет в A_2, A_2 — в A_3, …, A_{N-1} — в A_N, а исходное значение последнего элемента будет потеряно). Первый элемент полученного массива положить равным 0.

```
task("Array79");
int n;
pt >> n;
double a[10];
for (int i = 0; i < n; ++i)
    pt >> a[i];
for (int i = n - 2; i >= 0; --i)
    a[i + 1] = a[i];
a[0] = 0;
```

```
for (int i = 0; i < n; ++i)
    pt << a[i];
```

Обработка элементов при правом сдвиге выполняется *с конца массива*, чтобы преждевременно не стереть исходные значения изменяемых элементов.

1.2.4. Серии целых чисел: Array116

Array116. Дан целочисленный массив *A* размера *N*. Назовем *серией* группу подряд идущих одинаковых элементов, а *длиной серии* — количество этих элементов (длина серии может быть равна 1). Сформировать два новых целочисленных массива *B* и *C* одинакового размера, записав в массив *B* длины всех серий исходного массива, а в массив *C* — значения элементов, образующих эти серии.

Задание Array116 является первым из заданий группы Array, посвященных обработке *серий*, то есть последовательностей одинаковых элементов. Для выделения серий в массиве может быть использовано простое наблюдение: *элемент массива является началом серии, если его значение не равно значению предыдущего элемента* (кроме того, началом серии является *первый* элемент массива).

```
task("Array116");
int n, k = 1;
pt >> n;
int a[10], b[10], c[10];
for (int i = 0; i < n; ++i)
    pt >> a[i];
b[0] = 1;
c[0] = a[0];
for (int i = 1; i < n; ++i)
    if (a[i - 1] == a[i])
```

```
            ++b[k - 1];
        else
        {
            b[k] = 1;
            c[k] = a[i];
            ++k;
        }
    for (int i = 0; i < k; ++i)
        pt << b[i];
    for (int i = 0; i < k; ++i)
        pt << c[i];
```

В переменной *k* хранится информация о количестве *заполненных* элементов массивов *b* и *c*. Поскольку число серий массива *a* может находиться в диапазоне от 1 до 10 (так как размер массива *a* по условию не превосходит 10 элементов), размер массивов *b* и *c* также задается «по максимуму», равному 10.

1.3. Проектное задание

1.3.1. Варианты

Выполните задачи группы Array, указанные в вашем варианте проектного задания (формулировки задач приводятся в [3]). Если вы не получили вариант проектного задания, то выполните задачи из какого-либо варианта.

ВАРИАНТ 1	**ВАРИАНТ 2**
(1) Формирование и вывод: 3, 13	(1) Формирование и вывод: 5, 9
(2) Анализ элементов: 24, 37, 40	(2) Анализ элементов: 22, 32, 49
(3) Изменение: 59, 76, 80, 91, 110, 115	(3) Изменение: 58, 74, 87, 100, 104, 114
(4) Серии целых чисел: 125	(4) Серии целых чисел: 130

ВАРИАНТ 3 (1) Формирование и вывод: 5, 15 (2) Анализ элементов: 19, 32, 48 (3) Изменение: 51, 70, 88, 98, 107, 113 (4) Серии целых чисел: 124	**ВАРИАНТ 4** (1) Формирование и вывод: 3, 16 (2) Анализ элементов: 21, 39, 45 (3) Изменение: 62, 78, 82, 90, 108, 113 (4) Серии целых чисел: 127
ВАРИАНТ 5 (1) Формирование и вывод: 2, 8 (2) Анализ элементов: 26, 30, 42 (3) Изменение: 55, 69, 82, 96, 101, 112 (4) Серии целых чисел: 122	**ВАРИАНТ 6** (1) Формирование и вывод: 1, 12 (2) Анализ элементов: 29, 33, 43 (3) Изменение: 52, 67, 86, 97, 108, 114 (4) Серии целых чисел: 120
ВАРИАНТ 7 (1) Формирование и вывод: 2, 9 (2) Анализ элементов: 23, 36, 41 (3) Изменение: 54, 71, 81, 93, 103, 112 (4) Серии целых чисел: 128	**ВАРИАНТ 8** (1) Формирование и вывод: 4, 11 (2) Анализ элементов: 18, 34, 46 (3) Изменение: 53, 77, 85, 99, 102, 112 (4) Серии целых чисел: 129
ВАРИАНТ 9 (1) Формирование и вывод: 1, 8 (2) Анализ элементов: 20, 31, 50 (3) Изменение: 60, 68, 81, 92, 109, 113 (4) Серии целых чисел: 123	**ВАРИАНТ 10** (1) Формирование и вывод: 6, 17 (2) Анализ элементов: 27, 35, 44 (3) Изменение: 61, 75, 89, 92, 106, 115 (4) Серии целых чисел: 121
ВАРИАНТ 11 (1) Формирование и вывод: 6, 14 (2) Анализ элементов: 25, 38, 45 (3) Изменение: 57, 73, 84, 94, 111, 115	**ВАРИАНТ 12** (1) Формирование и вывод: 4, 10 (2) Анализ элементов: 28, 33, 44 (3) Изменение: 56, 72, 83, 95, 105, 114

(4) Серии целых чисел: 126	(4) Серии целых чисел: 119
ВАРИАНТ 13 (1) Формирование и вывод: 2, 10 (2) Анализ элементов: 29, 34, 45 (3) Изменение: 59, 72, 82, 92, 101, 114 (4) Серии целых чисел: 122	**ВАРИАНТ 14** (1) Формирование и вывод: 5, 16 (2) Анализ элементов: 27, 32, 42 (3) Изменение: 58, 68, 84, 96, 110, 114 (4) Серии целых чисел: 128
ВАРИАНТ 15 (1) Формирование и вывод: 3, 13 (2) Анализ элементов: 18, 37, 49 (3) Изменение: 62, 78, 88, 90, 103, 115 (4) Серии целых чисел: 126	**ВАРИАНТ 16** (1) Формирование и вывод: 3, 8 (2) Анализ элементов: 28, 31, 43 (3) Изменение: 57, 70, 86, 95, 102, 115 (4) Серии целых чисел: 127
ВАРИАНТ 17 (1) Формирование и вывод: 2, 9 (2) Анализ элементов: 23, 39, 40 (3) Изменение: 52, 73, 87, 97, 108, 115 (4) Серии целых чисел: 120	**ВАРИАНТ 18** (1) Формирование и вывод: 5, 8 (2) Анализ элементов: 21, 36, 44 (3) Изменение: 51, 69, 83, 99, 106, 114 (4) Серии целых чисел: 124
ВАРИАНТ 19 (1) Формирование и вывод: 4, 12 (2) Анализ элементов: 26, 32, 41 (3) Изменение: 60, 76, 80, 94, 105, 113 (4) Серии целых чисел: 119	**ВАРИАНТ 20** (1) Формирование и вывод: 6, 14 (2) Анализ элементов: 24, 35, 45 (3) Изменение: 61, 74, 85, 98, 107, 113 (4) Серии целых чисел: 130
ВАРИАНТ 21 (1) Формирование и вывод: 4, 15 (2) Анализ элементов: 19, 38, 48	**ВАРИАНТ 22** (1) Формирование и вывод: 6, 17 (2) Анализ элементов: 20, 33, 50

(3) Изменение: 55, 75, 89, 100, 104, 112 (4) Серии целых чисел: 121	(3) Изменение: 56, 77, 81, 92, 108, 113 (4) Серии целых чисел: 123
ВАРИАНТ 23 (1) Формирование и вывод: 1, 9 (2) Анализ элементов: 22, 30, 44 (3) Изменение: 53, 67, 81, 91, 109, 112 (4) Серии целых чисел: 125	**ВАРИАНТ 24** (1) Формирование и вывод: 1, 11 (2) Анализ элементов: 25, 33, 46 (3) Изменение: 54, 71, 82, 93, 111, 112 (4) Серии целых чисел: 129

1.3.2. Указания

Array1–6. При вычислении очередного элемента массива удобно использовать значение предыдущего элемента (в Array5 — значения двух предыдущих элементов). Следует заметить, что в Array6 для вычисления очередного элемента (начиная с четвертого) также достаточно использовать значение только *одного* предыдущего элемента.

Array7. Решение этого задания приводится в п. 1.2.1.

Array11–15. Организуйте цикл таким образом, чтобы в нем перебирались только элементы с требуемыми индексами, учитывая, что индекс элемента на 1 меньше его порядкового номера. Например, для перебора всех нечетных индексов можно использовать выражение $2K - 1$, $K = 1, 2, \ldots$ (нечетные индексы соответствуют *четным порядковым номерам*). При организации цикла важно правильно определить *конечное значение* параметра цикла. В Array14–15 следует учитывать, что размер массива может быть как четным, так и нечетным.

Array16–17. На каждой итерации цикла удобно выводить не один, а *несколько* элементов массива (два в Array16, четыре в Array17). При этом может возникнуть необходимость в особой обработке некоторых элементов массива

(например, если массив имеет нечетный размер, то при выполнении Array16 средний элемент этого массива надо вывести после завершения цикла).

Array18–19. В Array18 после обнаружения первого подходящего элемента следует прервать цикл, используя оператор *break*. В Array19 ищется *последний* подходящий элемент, поэтому необходимо просмотреть *все* элементы, которые могут оказаться подходящими (в данном случае это элементы с индексами от 1 до 8). Для решения Array19 можно также просматривать исходный массив *с конца* (см. Array7); в этом случае, как и при решении Array18, достаточно обнаружить первый подходящий элемент, после чего выйти из цикла.

Array24–25. Используя два первых элемента массива, найдите возможное значение разности (знаменателя) прогрессии, после чего в цикле проверьте остальные элементы массива (например, в Array25 каждый элемент массива должен быть равен предыдущему, умноженному на найденный знаменатель).

Array26–27. В Array26 достаточно проверить, что сумма двух соседних элементов массива является нечетной, в Array27 — что произведение двух соседних элементов является отрицательным.

Array28–29. Ср. с решением Minmax1 в [1], п. 4.2.2. См. также указание к Array11–15.

Array30–31. При переборе элементов массива следует учесть, что первый из элементов не имеет левого соседа, а последний не имеет правого.

Array32–35. При анализе элементов необходимо учитывать, что первый и последний элементы массива (*граничные* элементы) имеют по одному соседу, тогда как все остальные (*внутренние* элементы) — по два. Можно анализировать первый элемент до цикла, в котором анализируются внутренние

элементы, а последний элемент — после цикла. Более короткий алгоритм можно получить, включив в массив *фиктивные* элементы с индексами 0 и $N + 1$, где N — исходный размер массива («настоящие» элементы будут иметь индексы от 1 до N). Тем самым граничные элементы превратятся во внутренние, и их можно будет обрабатывать в общем цикле. При этом необходимо так определить значения фиктивных элементов, чтобы они обеспечили правильный анализ граничных элементов. Можно, например, положить фиктивный элемент равным «настоящему» соседу граничного элемента, то есть присвоить элементу с индексом 0 значение элемента с индексом 2, а элементу с индексом $N + 1$ — значение элемента с индексом $N - 1$.

Array36. Первый и последний элементы массива можно исключить из рассмотрения, так как любой граничный элемент вещественного массива *обязательно* является либо локальным минимумом, либо локальным максимумом (поскольку предполагается, что все элементы вещественного массива являются *различными*). Анализ остальных (внутренних) элементов можно упростить, если учесть, что внутренний элемент A_K является *локальным экстремумом* (то есть минимумом или максимумом) в том случае, когда

$$(A_K - A_{K-1}) \cdot (A_K - A_{K+1}) > 0.$$

Array37–39. Поскольку любой участок возрастания оканчивается локальным максимумом, а любой участок убывания — локальным минимумом, эти задания сводятся к заданиям о нахождении локальных минимумов/максимумов (Array32–36). Для Array37–38 воспользуйтесь указанием к Array32–35, для Array39 — указанием к Array36.

Array44–46. Для перебора пар различных элементов массива надо использовать *двойной цикл*. Перебор следует организовать таким образом, чтобы в пару входили *различные* элементы, и чтобы каждая пара анализировалась *ровно один раз*. Для этого достаточно в двойном цикле сделать один из

пределов изменения параметра *внутреннего* цикла зависящим от текущего значения параметра *внешнего* цикла, например:

```
for (i = 0; i < n - 1; ++i)
    for (j = i + 1; j < n; ++j)
    ...
```

Array47. Решение этого задания приводится в п. 1.2.2.

Array48. Ср. с Array47.

Array49. Элемент будет недопустимым, если его значение либо не принадлежит диапазону чисел от 1 до *N*, либо уже встречалось ранее в массиве. Таким образом, для решения задания достаточно выполнить соответствующую проверку для каждого элемента массива. Для проверки того, что значение элемента уже встречалось в массиве, можно просмотреть все предыдущие элементы во вложенном цикле.

Array50. См. указание к Array44–46.

Array54. Поскольку размер результирующего массива *B* заранее не известен, следует завести вспомогательную переменную *K* целого типа, в которой хранить количество элементов, *уже размещенных* в массиве *B* (вначале переменная *K* равна нулю; при обнаружении в исходном массиве элемента, который следует записать в массив *B*, этот элемент присваивается элементу массива *B* с индексом *K*, после чего значение *K* увеличивается на 1). Когда просмотр исходного массива *A* завершится, в переменной *K* будет храниться *размер* сформированного массива *B*.

Array55–57. См. указание к Array11–15.

Array58–61. В этих заданиях, как и в Array6, можно обойтись без вложенных циклов.

Array62. См. указание к Array54.

Array63–64. Задания на *слияние* упорядоченных массивов. Используйте для каждого исходного массива вспомогательную переменную, содержащую индекс его *текущего элемента*, то есть первого из элементов, еще не помещенных в результирующий массив. Очередной элемент, добавляемый в результирующий массив, выбирается из текущих элементов исходных массивов, после чего происходит переход к следующему элементу в том массиве, элемент которого был добавлен в результирующий массив. При реализации этого алгоритма возникает дополнительная проблема: что делать в ситуации, когда исчерпываются элементы одного из массивов. Для ее решения удобно использовать прием *барьера*, добавив в конец каждого массива *фиктивный* элемент. Если массивы упорядочены по возрастанию (как в Array63), то эти фиктивные элементы должны быть заведомо *больше* «настоящих» элементов всех исходных массивов; если массивы упорядочены по убыванию (как в Array64), то фиктивные элементы должны быть заведомо *меньше* всех «настоящих» элементов. В качестве фиктивных элементов можно использовать определения из заголовочного файла *limit*s, возвращающие минимальные или максимальные значения требуемых числовых типов (см. [1], п. 5.1), например, *numeric_limits<int>::min*() или *numeric_limits<double>::max*().

Array71–73. Для изменения порядка в наборе из K элементов на обратный достаточно поменять местами первый элемент с последним, второй с предпоследним и т. д.; при этом число обменов должно быть равно $K/2$ (распространенной ошибкой является повторение операций обмена K раз; в этом случае каждая пара элементов меняется местами *дважды*, и в результате порядок элементов не изменяется).

Array74–75. Для решения этих заданий необходимо предварительно найти *индексы* минимального и максимального элементов массива (ср. с Minmax4 в [1], п. 4.3.3) и выяснить, в каком порядке эти элементы располагаются в

массиве (то есть какой индекс является меньшим, а какой — большим). Для Array75 см. также указание к Array71–73.

Array76–78. Перед обработкой элемента массива необходимо сохранять его прежнее значение во *временной переменной*, которая на следующей итерации цикла будет использована при анализе следующего элемента. Подчеркнем, что в этих заданиях, как и в прочих заданиях на преобразование массива, достаточно использовать несколько *простых* временных переменных, не заводя вспомогательного массива. Для Array76–77 см. также указание к Array32–35.

Array79–82. Задания на *простой сдвиг* элементов (с потерей части исходных данных). Решение Array79 приводится в п. 1.2.3; прочие задания решаются аналогично. При их выполнении не требуются временные переменные для хранения элементов массива. Сдвиг влево следует проводить в направлении от начала массива к его концу, а сдвиг вправо — в направлении от конца к началу.

Array83–86. Задания на *циклический сдвиг* элементов. Направление циклического сдвига совпадает с направлением простого сдвига (см. указание к Array79–82), однако, в отличие от заданий на простой сдвиг, здесь необходимо использовать *временные переменные* для хранения элементов. В случае сдвига на одну позицию достаточно одной переменной (при сдвиге влево в ней сохраняется первоначальное значение первого элемента, при сдвиге вправо — первоначальное значение последнего элемента). При сдвиге на K позиций можно использовать временный массив размера 4 (так как по условию задания $K \le 4$). Другой вариант решения позволяет для *любого* K обойтись *единственной* временной переменной (за счет потери в скорости выполнения): он состоит в последовательном *повторении* сдвига на *одну* позицию в требуемом направлении; число повторений равно остатку от деления K на N, где N — размер массива.

Array87–88. Для повышения эффективности алгоритма следует совместить поиск места вставки и сдвиг элементов массива для выделения пустой позиции в месте вставки. Так, для Array87 следует сохранить первый элемент массива во временной переменной *X* и выполнять сдвиг остальных элементов массива на одну позицию влево до тех пор, пока не будет найден элемент, больший *X*, после чего записать в массив значение *X перед* этим элементом (для Array88 следует сохранить *последний* элемент массива и выполнять сдвиг *вправо*). Необходимо также предусмотреть ситуацию, когда для достижения упорядоченности первый элемент придется переместить в конец массива (или, соответственно, последний элемент в начало). Для обработки этой особой ситуации удобно использовать прием *барьера*, устанавливаемого в конце массива (или, соответственно, в его начале). Данный прием описан в указании к Array63–64.

Array89. Решение следует разбить на два этапа: вначале ищется элемент, нарушающий упорядоченность, затем этот элемент перемещается в новую позицию. На первом этапе надо просматривать элементы исходного массива *A* в поисках *возрастающей пары* соседних элементов: $A_I < A_{I+1}$. Обнаружение такой пары означает, что элементом, нарушающим упорядоченность, является *либо* A_I, *либо* A_{I+1}. Например, для набора 6, 1, 4, 3 нарушает упорядоченность элемент 1 (то есть *первый* элемент возрастающей пары), для набора 6, 3, 7, 1 нарушает упорядоченность элемент 7 (то есть *второй* элемент возрастающей пары), наконец, для набора 6, 3, 4, 1 в качестве элемента, нарушающего упорядоченность, можно выбрать как 3, так и 4. После нахождения элемента, нарушающего упорядоченность, наступает второй этап: перемещение найденного элемента на новую позицию. На этом этапе вначале надо определить *направление* перемещения (влево или вправо), после чего действовать по схеме, описанной в указании к Array87–88.

Array91–95. В заданиях на удаление нескольких элементов желательно *сразу* перемещать оставляемый в массиве элемент на его окончательное место, избегая многократного сдвига одного и того же элемента массива. Для этого надо завести вспомогательную переменную *K*, в которой хранить количество элементов, *уже размещенных* на новых позициях в ходе просмотра массива (таким образом, очередной элемент, оставляемый в массиве, должен размещаться в позиции с индексом *K*). Просмотр элементов должен осуществляться в направлении от начала к концу массива. Для Array93–94 см. также указание к Array11–15.

Array96–97. В этих заданиях для принятия решения об удалении текущего элемента следует просматривать все элементы массива, *предшествующие* текущему (в случае Array96) или *следующие* за текущим (в случае Array97). В остальном алгоритм не будет отличаться от схемы, описанной в указании к Array91–95. Заметим, что в случае Array96 изменение начальной части массива в ходе перемещения в нее элементов, оставляемых в массиве, *не повлияет* на правильность алгоритма.

Array98–100. В этих заданиях для принятия решения об удалении текущего элемента следует просматривать *все* элементы массива, причем в данном случае (в отличие от предыдущих заданий) изменение начальной части массива при перемещении в нее элементов, оставляемых в массиве, *может привести к неправильной работе алгоритма*. Для того чтобы этого не произошло, можно применить следующую модификацию алгоритма, описанного в указании к Array91–95: не просто перемещать текущий элемент, оставляемый в массиве, на позицию с индексом *K* (стирая тем самым прежнее содержимое элемента с данным индексом), а *менять местами* текущий элемент и элемент с индексом *K*. Благодаря этому исходный состав элементов массива будет сохраняться неизменным, что позволит на любом

этапе выполнения алгоритма правильно определить, надо ли удалять из массива текущий элемент.

Array101–107. При выполнении заданий на вставку элементов в массив необходимо, прежде всего, предусмотреть *место* в памяти для размещения новых элементов, указав в описании массива больший размер. Далее, в алгоритме вставки элементов, как и в алгоритме удаления, желательно *сразу* перемещать элемент массива на его новое место (см. указание к Array91–95). Отличие от алгоритма удаления состоит в том, что конечные элементы массива при вставке новых элементов должны перемещаться не влево (как при удалении), а *вправо*, поэтому в данном случае перебор элементов следует проводить *от конца массива к его началу*. В Array101–107 размер результирующего массива можно определить, не анализируя элементы исходного массива. Для Array106–107 см. также указание к Array11–15.

Array108–111. При выполнении этих заданий необходимо предварительно определить *размер* результирующего массива, подсчитав количество его положительных (Array108), отрицательных (Array109), четных (Array110) или нечетных (Array111) элементов. В остальном алгоритм вставки элементов не будет отличаться от схемы, описанной в указании к Array101–107.

Array112–115. Более подробные сведения об алгоритмах сортировки содержатся, например, в [5].

Array116–130. Решение Array116 приводится в п. 1.2.4. Наиболее простой способ решения остальных заданий на обработку серий состоит в использовании вспомогательных массивов *B* и *C*, содержащих, соответственно, *длины* всех серий исходного массива *A* и *значения* элементов, образующих эти серии. После формирования массивов *B* и *C* (см. решение Array116) следует обработать эти вспомогательные массивы требуемым образом (например, в Array119 достаточно увеличить значение каждого элемента массива *B* на 1) и затем *сформировать заново* массив *A* по данным из массивов *B* и *C*, учи-

тывая условия задания (например, в Array117 перед записью очередной серии надо добавлять в массив *A* элемент с нулевым значением). Желательно, однако, при выполнении заданий Array117–130 *обойтись без использования вспомогательных массивов*.

1.4. Тест рубежного контроля

Тест содержит 6 заданий, на выполнение которых отводится 3 минуты. Выберите правильный, по вашему мнению, вариант ответа и отметьте его любым значком в бланке ответов.

<table>
<tr><td colspan="4">1. Укажите правильный вариант описания целочисленного массива a размера 6.</td></tr>
<tr><td>(1)</td><td><code>int a[6];</code></td><td>(2)</td><td><code>int[] a =
new int [6];</code></td></tr>
<tr><td>(3)</td><td><code>int [] a[6];</code></td><td>(4)</td><td><code>int* a = int[6];</code></td></tr>
<tr><td colspan="4">2. Укажите неверное утверждение.</td></tr>
<tr><td>(1)</td><td>Массивы индексируются с 0.</td><td>(2)</td><td>При создании массива все элементы инициализируются нулевыми значениями.</td></tr>
<tr><td>(3)</td><td>Выход за границы изменения индекса может привести к неопределенным последствиям.</td><td>(4)</td><td>Индекс последнего элемента массива равен количеству его элементов минус 1.</td></tr>
<tr><td colspan="4">3. Память для массива была выделена следующим образом:
<code>int* a = new int[10];</code>
Для освобождения памяти, занятой массивом a, следует использовать операцию:</td></tr>
<tr><td>(1)</td><td><code>delete a[];</code></td><td>(2)</td><td><code>delete a[10];</code></td></tr>
<tr><td>(3)</td><td><code>delete[] a;</code></td><td>(4)</td><td><code>delete[10] a;</code></td></tr>
</table>

4. Укажите содержимое массива *a* после выполнения следующего фрагмента:

```
int a[5];
for (int i = 0; i < 5; ++i)
    a[i] = 5 - i;
```

(1)	1, 2, 3, 4, 5	(2)	0, 1, 2, 3, 4
(3)	5, 4, 3, 2, 1	(4)	4, 3, 2, 1, 0

5. Что произойдет при выполнении следующего кода:

```
int a[5];
a[5] = 10;
```

(1)	Последнему элементу массива присвоится значение 10.	(2)	Этот код не скомпилируется.
(3)	Программа остановится с ошибкой в момент выполнения этого фрагмента.	(4)	Неизвестно, что произойдет (неопределенное поведение).

6. Укажите содержимое массива *a* после выполнения следующего фрагмента:

```
const int n = 5;
int a[n];
for (int i = n-1; i >= 0; --i)
    a[i] = n – i - 1;
```

(1)	4, 3, 2, 1, 0	(2)	0, 1, 2, 3, 4
(3)	5, 4, 3, 2, 1	(4)	1, 2, 3, 4, 5

2. Модуль № 2. Символьные строки

2.1. Комплексная цель

Изучить возможности языка C++, связанные с обработкой символов и строк. Рассмотреть способы преобразования чисел в их строковые представления, а также способы выполнения обратного преобразования. Освоить основные приемы обработки строк, в том числе способы анализа слов в строке.

2.2. Содержание

2.2.1. Формирование строк: String10

String10. Дана строка. Вывести строку, содержащую те же символы, но расположенные в обратном порядке.

Данное задание можно выполнить с помощью операции *сцепления* (*конкатенации*) +, если учесть, что для доступа к *i*-му символу строки *s* в C++ следует использовать конструкцию *s*[*i*].

```
task("String10");
string s, s1;
pt >> s;
for (string::size_type i = 0; i < s.length(); ++i)
    s1 = s[i] + s1;
pt << s1;
```

Символы исходной строки *s* добавляются в начало результирующей строки *s1*; это обеспечивает требуемый порядок их размещения. Индексы отдельных символов строки принадлежат к типу *string::size_type*, который на большинстве платформ является аналогом *unsigned int*. Заметим, что запись *string::size_type* означает, что тип *size_type* определен *внутри* класса *string*, поэтому при обращении к нему используется *квалификатор* «`::`». Для определения длины строки *s* используется функция *length* (являющаяся функцией-

членом класса *string* и поэтому вызываемая с помощью точечной нотации: *s.length*()). Функция *length* возвращает значение типа *string::size_type*.

Следует заметить, что при создании объектов типа *string* они автоматически инициализируются пустыми строками, поэтому в начальном присваивании вида *s1* = "" нет необходимости (напомним, что строковые константы в C++ заключаются в двойные кавычки).

Приведем другой способ решения задания String10, в котором результирующая строка заполняется *посимвольно*, как обычный массив (ср. с решением задания Array7 из п. 1.2.1).

```
task("String10");
string s;
pt >> s;
string::size_type n = s.length();
string s1(n, ' ');
for (string::size_type i = 0; i < n; ++i)
    s1[i] = s[n - i - 1];
pt << s1;
```

Поскольку длина исходной строки многократно используется в тексте программы, она сохраняется во вспомогательной переменной *n*. Для возможности доступа к символам строки *s1* необходимо, чтобы данные символы *существовали*, поэтому переменная *s1* инициализируется строкой длины *n*, состоящей из пробелов (для этого используется конструктор *string::string*(*n*, *c*), создающий строку длины *n*, состоящую из одинаковых символов *c*). Заметим, что символы-константы, в отличие от строк-констант, заключаются не в двойные, а в одинарные кавычки.

Возможен еще один вариант решения, в котором преобразуется нужным образом сама исходная строка:

```
task("String10");
```

```
string s;
pt >> s;
string::size_type n = s.length();
for (string::size_type i = 0; i < n / 2; ++i)
{
    char c = s[i];
    s[i] = s[n - i - 1];
    s[n - i - 1] = c;
}
pt << s;
```

Количество итераций цикла равно *n*/2, где *n* — длина исходной строки (выполняется *целочисленное* деление); благодаря этому каждая пара символов меняется местами *ровно один раз*.

☑ Класс *string*, реализованный в стандартной библиотеке C++, предоставляет надежные, удобные и эффективные средства для обработки строковых данных. Однако язык C++ предполагает также возможность обработки строк традиционным для языка C способом — с применением символьных массивов с элементами типа *char* и связанных с ними стандартных функций. Приведем в качестве примера соответствующий вариант решения задачи String10:

```
task("String10");
char s[81], s1[81];
pt >> s;
int n = strlen(s);
for (int i = 0; i < n; ++i)
    s1[i] = s[n - i - 1];
s1[n] = '\0';
pt << s1;
```

В данном варианте решения предполагается, что длина исходной строки не превышает 80 символов (при описании символьного массива в нем необходимо также зарезервировать дополнительный элемент для хранения '\0' — *нулевого*, или *терминального* символа строки с кодом 0). При выполнении оператора *pt* >> *s* в символьный массив *s* копи-

руются все символы исходной строки, после чего в него добавляется нулевой символ '\0'. Благодаря этому нулевому символу стандартная функция *strlen* правильно определяет длину прочитанной строки. Обратите также внимание на оператор *s1*[*n*] = '\0', обеспечивающий запись нулевого символа в соответствующую позицию результирующего строкового массива *s1*. Если бы этот оператор отсутствовал, то при передаче массива *s1* в поток вывода *pt* не удалось бы правильно определить длину полученной строки.

Как видно из приведенного примера, обработка строковых данных с помощью массивов с символьными элементами является не слишком удобной и может приводить к появлению трудно выявляемых ошибок. Поэтому в дальнейшем при обработке строк мы будем ограничиваться классом *string*.

2.2.2. Преобразование строк в числа: String19

String19. Дана строка. Если она представляет собой запись целого числа, то вывести 1, если вещественного (с дробной частью) — вывести 2; если строку нельзя преобразовать в число, то вывести 0. Считать, что дробная часть вещественного числа отделяется от его целой части десятичной *точкой* «.».

Приводимое ниже решение этого задания демонстрирует использование стандартной функции *atof*, предназначенной для преобразования строки в вещественное число (типа *double*). Данный вариант решения правильно распознает не только числа в формате с фиксированной точкой (например, "1.2345"), но также и числа в формате с плавающей точкой (например, "1.2345e+02").

```
task("String19");
string s;
pt >> s;
int k = 0;
double a = atof(s.c_str());
if (a != atof((s + '1').c_str()))
    k = (a == ceil(a) ? 1 : 2);
```

```
pt << k;
```

В качестве параметра функции *atof* необходимо указать символьный массив, содержащий исходную строку. Для получения такого массива из исходной строки *s* типа *string* достаточно использовать функцию-член *c_str* класса *string*: *s.c_str*().

Функция *atof* преобразует в число не всю строку, а только ту ее начальную часть, которую *можно* преобразовать в число (при этом предварительно пропускаются начальные пробельные символы строки). Если уже первый символ, расположенный после начальных пробельных символов, является недопустимым (или строка состоит только из пробельных символов), то функция возвращает 0. Поэтому для проверки того, что *вся* исходная строка представляет собой число, в программе использован следующий прием: делается попытка преобразовать в число строку *s*, *дополненную* справа символом «1». Если при подобном преобразовании возвращается то же значение, что и при преобразовании исходной строки *s*, значит, добавленная единица не была использована при преобразовании строки в число, а это возможно только при наличии в исходной строке *s* недопустимого символа.

Для того чтобы проверить, содержит ли число *a* дробную часть, достаточно сравнить это число с его округленным значением *ceil*(*a*) (напомним, что при использовании в программе математических функций к ней необходимо подключить заголовочный файл *cmath*). Здесь следует обратить внимание на использование *тернарной операции* ?:, с помощью которой можно избежать применения лишнего условного оператора.

☑ Для преобразования строкового представления *целого* числа в само это число предназначена стандартная функция: *atoi*(*s*) (с параметром — символьным массивом), возвращающая значение типа *int*.

Другой вариант решения этой задачи предполагает использование *строковых потоков*, возможности которых аналогичны стандартным потокам ввода/вывода *cin* и *cout*.

```
task("String19");
string s;
pt >> s;
istringstream is(s);
int k = 0;
double a;
is >> a;
if (is && is.eof())
    k = (a == ceil(a) ? 1 : 2);
pt << k;
```

Строковый поток ввода реализуется классом *istringstream* из заголовочного файла *sstream*. При объявлении соответствующего объекта в конструкторе указывается строка, содержащая исходные данные для потока. Чтение данных из такого потока происходит обычным образом с помощью операции >>.

Опишем алгоритм решения. Сначала делается попытка чтения из строкового потока вещественной переменной, если это не удается, поток приходит в ошибочное состояние и первая часть условия в операторе *if* не выполняется (см. также п. 5.9), таким образом, значение переменной *k* остается нулевым. Если в потоке осталось что-то еще, кроме вещественного числа, то условие оператора *if*, также окажется ложным (не сработает проверка завершения потока *is.eof*()). Если же все проверки прошли успешно, то останется проверить, содержит ли число дробную часть, это делается описанным ранее способом.

Более подробно строковые потоки описываются в п. 5.15.

2.2.3. Анализ слов в строке: String41

При выполнении заданий на анализ и преобразование слов главная проблема заключается в выделении каждого слова из исходной строки. Будем предполагать, что строка не является пустой, не содержит начальных и конечных пробелов, и слова в ней разделяются одним или несколькими пробелами

(именно такие строки предлагаются в качестве исходных данных во всех заданиях соответствующего раздела группы String). Опишем для таких строк алгоритм выделения слов, использующий стандартные функции для работы со строками.

Ищется первый пробел в строке; начало строки, вплоть до найденного пробела, представляет собой *первое слово*. После обработки этого слова оно удаляется из исходной строки вместе со следующими за ним пробелами. Если после этого строка не становится пустой, то описанная процедура повторяется (в результате выделяется второе слово, и т. д.).

Воспользуемся описанным алгоритмом для решения задания String41 — первого задания из раздела, посвященного анализу слов в строке.

String41. Дана строка, состоящая из русских слов, разделенных пробелами (одним или несколькими). Найти количество слов в строке.

```
task("String41");
string s;
pt >> s;
int n = 0;
while (!s.empty())
{
    string::size_type p = s.find(" ");
    ++n;
    s.erase(0, p);
    s.erase(0, s.find_first_not_of(" "));
}
pt << n;
```

Функция *s.empty*() возвращает значение *true*, если строка является пустой, поэтому данный цикл будет выполняться, пока строка непустая.

Для поиска первого пробела используется функция *find*: *s*.*find*(*sub*) возвращает начальную позицию первой найденной подстроки *sub* в строке *s* или специальное значение *string::npos*, если подстрока *sub* в строке *s* отсутствует.

Для удаления из строки обработанного слова используется функция *erase*. Ее параметрами являются позиция символа, с которого начинается удаление, и количество удаляемых символов. За обработанным словом могут располагаться один или несколько пробелов, для их удаления следует найти первый непробельный символ, для этого используется функция *find_first_not_of*: *s*.*find_first_not_of(sub)* возвращает позицию первого символа строки *s*, не содержащегося в строке *sub*. После такого поиска можно удалять все символы, предшествующие непробельному.

Заметим, что при обработке последнего слова строки возникает особая ситуация: функции *s*.*find*() и *s*.*find_first_not_of*() вернут значение *string::npos*. Тем не менее, оно будет корректно использовано: первый вызов функции *s*.*erase*() удалит все *до конца* строки, а второй не будет иметь никакого эффекта, поскольку применится уже к пустой строке.

Так как в задании String41 требуется лишь подсчитать содержащиеся в строке слова, обработка слов сводится к выполнению единственного оператора, увеличивающего на 1 переменную-счетчик *n*. Если в задании требуется проанализировать очередное слово, то его можно получить в данном месте программы с помощью выражения *s*.*substr*(0, *p*).

Заметим, что описанный алгоритм легко модифицировать так, чтобы он правильно обрабатывал строки, имеющие начальные и конечные пробелы (а также пустые строки и строки, состоящие из одних пробелов). Для этого достаточно перед циклом *while* организовать удаление из строки начальных пробелов:

```
s.erase(0, s.find_first_not_of(" "));
```

Альтернативой для данного варианта решения является *посимвольный анализ строки*, который, как правило, оказывается более эффективным. В случае задания String41 можно воспользоваться тем, что *каждому слову в строке, кроме первого, предшествует пробел*:

```
task("String41");
string s;
pt >> s;
s = ' ' + s
int n = 0;
for (string::size_type i = 1; i < s.length(); ++i)
    if (s[i] != ' ' && s[i - 1] == ' ')
        n++;
pt << n;
```

Для того чтобы перед каждым словом в строке присутствовал пробел, после считывания исходной строки к ней добавляется пробел слева. Обратите внимание на то, что в цикле анализируются все символы исходной строки, кроме первого (с индексом 0).

В заключение заметим, что на основе посимвольного анализа строки *s* можно реализовать вариант алгоритма выделения слов, отличный от приведенного в начале пункта:

```
s = ' ' + s + ' ';
for (string::size_type i = 1; i < s.length() - 1; ++i)
{
    if (s[i] != ' ' && s[i - 1] == ' ')
        n1 = i;
    if (s[i] != ' ' && s[i + 1] == ' ')
    {
        n2 = i;
```

```
            // обработка очередного слова,
            // равного s.substr(n1, n2 – n1 + 1)
        }
    }
```

Первый условный оператор позволяет найти позицию *n1 начала* каждого слова, а второй — позицию *n2* его *конца*. Для того чтобы начало первого слова и конец последнего слова также можно было обнаружить в цикле, в начало и конец исходной строки *s* добавляются *дополнительные пробелы*. Приведенный алгоритм правильно обрабатывает строки, содержащие начальные и конечные пробелы, а также пустые строки и строки, состоящие из одних пробелов.

2.3. Проектное задание

2.3.1. Варианты

Выполните задачи группы String, указанные в вашем варианте проектного задания (формулировки задач приводятся в [3]). Если вы не получили вариант проектного задания, то выполните задачи из какого-либо варианта.

ВАРИАНТ 1 (1) Формирование: 2, 11 (2) Обработка: 18, 25, 27, 38 (3) Анализ слов: 47, 52 (4) Разные задачи: 61, 65, 68	**ВАРИАНТ 2** (1) Формирование: 1, 7 (2) Обработка: 17, 20, 28, 37 (3) Анализ слов: 44, 55 (4) Разные задачи: 58, 63, 70
ВАРИАНТ 3 (1) Формирование: 4, 8 (2) Обработка: 15, 22, 28, 39 (3) Анализ слов: 50, 54 (4) Разные задачи: 60, 64, 70	**ВАРИАНТ 4** (1) Формирование: 6, 9 (2) Обработка: 16, 20, 28, 33 (3) Анализ слов: 43, 57 (4) Разные задачи: 60, 62, 69

ВАРИАНТ 5 (1) Формирование: 2, 7 (2) Обработка: 15, 24, 29, 31 (3) Анализ слов: 49, 57 (4) Разные задачи: 59, 66, 69	**ВАРИАНТ 6** (1) Формирование: 6, 11 (2) Обработка: 13, 23, 27, 40 (3) Анализ слов: 46, 52 (4) Разные задачи: 59, 66, 70
ВАРИАНТ 7 (1) Формирование: 5, 9 (2) Обработка: 16, 21, 30, 36 (3) Анализ слов: 45, 56 (4) Разные задачи: 60, 62, 70	**ВАРИАНТ 8** (1) Формирование: 3, 12 (2) Обработка: 13, 23, 28, 39 (3) Анализ слов: 42, 55 (4) Разные задачи: 58, 63, 68
ВАРИАНТ 9 (1) Формирование: 5, 9 (2) Обработка: 14, 22, 29, 32 (3) Анализ слов: 51, 54 (4) Разные задачи: 61, 64, 68	**ВАРИАНТ 10** (1) Формирование: 4, 8 (2) Обработка: 18, 24, 30, 34 (3) Анализ слов: 49, 56 (4) Разные задачи: 61, 65, 69
ВАРИАНТ 11 (1) Формирование: 1, 12 (2) Обработка: 14, 25, 26, 40 (3) Анализ слов: 48, 53 (4) Разные задачи: 58, 67, 68	**ВАРИАНТ 12** (1) Формирование: 3, 9 (2) Обработка: 17, 21, 26, 35 (3) Анализ слов: 48, 53 (4) Разные задачи: 59, 67, 69
ВАРИАНТ 13 (1) Формирование: 2, 8 (2) Обработка: 16, 24, 26, 31 (3) Анализ слов: 49, 55	**ВАРИАНТ 14** (1) Формирование: 3, 7 (2) Обработка: 18, 24, 27, 32 (3) Анализ слов: 44, 57

(4) Разные задачи: 59, 62, 69	(4) Разные задачи: 60, 62, 68
ВАРИАНТ 15 (1) Формирование: 1, 9 (2) Обработка: 15, 20, 28, 35 (3) Анализ слов: 48, 53 (4) Разные задачи: 60, 66, 68	**ВАРИАНТ 16** (1) Формирование: 6, 9 (2) Обработка: 13, 21, 28, 34 (3) Анализ слов: 42, 52 (4) Разные задачи: 58, 64, 68
ВАРИАНТ 17 (1) Формирование: 4, 7 (2) Обработка: 15, 23, 28, 38 (3) Анализ слов: 50, 52 (4) Разные задачи: 60, 65, 70	**ВАРИАНТ 18** (1) Формирование: 3, 9 (2) Обработка: 18, 22, 28, 40 (3) Анализ слов: 45, 57 (4) Разные задачи: 58, 63, 69
ВАРИАНТ 19 (1) Формирование: 1, 11 (2) Обработка: 13, 22, 29, 37 (3) Анализ слов: 46, 55 (4) Разные задачи: 61, 67, 68	**ВАРИАНТ 20** (1) Формирование: 5, 11 (2) Обработка: 14, 20, 30, 36 (3) Анализ слов: 51, 54 (4) Разные задачи: 61, 64, 70
ВАРИАНТ 21 (1) Формирование: 4, 9 (2) Обработка: 17, 21, 29, 39 (3) Анализ слов: 43, 54 (4) Разные задачи: 58, 63, 70	**ВАРИАНТ 22** (1) Формирование: 5, 12 (2) Обработка: 17, 23, 30, 40 (3) Анализ слов: 47, 56 (4) Разные задачи: 59, 65, 70
ВАРИАНТ 23 (1) Формирование: 2, 8 (2) Обработка: 14, 25, 26, 33	**ВАРИАНТ 24** (1) Формирование: 6, 12 (2) Обработка: 16, 25, 27, 39

(3) Анализ слов: 49, 53 (4) Разные задачи: 59, 66, 69	(3) Анализ слов: 48, 56 (4) Разные задачи: 61, 67, 69

2.3.2. Указания

String1–2. Для определения кодового номера символа в используйте преобразование символа к типу *int*. Для определения символа с данным кодовым номером используйте преобразование номера к типу *char*. Подробнее о преобразовании типов см. п. 5.1 пособия [1].

String3–5. Для получения символа, следующего в кодовой таблице за данным символом *c*, можно использовать выражение *static_cast<char>(c+1)* (для получения предшествующего символа достаточно заменить в указанном выражении знак «+» на «–»).

String6. Чтобы проверить, является ли цифрой символ *c*, можно использовать операции сравнения: *c >= '0' && c <= '9'*. Для проверки цифр можно также использовать функцию *isdigit* из заголовочного файла *cctype*. Буквы анализируются с помощью операций сравнения; следует учитывать, что латинские и русские буквы, имеющие одинаковые изображения (например, «а»), считаются *различными* символами, поскольку имеют разные кодовые номера (см. п. 5.4).

String7. Для доступа к *i*-му символу строки *s* достаточно применить к строке операцию *индексирования* (как к обычному массиву): *s*[*i*]. Для определения длины строки используйте функцию-член *string::length*.

String8–9. Используйте операцию сцепления (конкатенации) строк. Операндами этой операции могут быть как строки, так и символы. Задание String8 легко выполнить с помощью подходящего конструктора класса *string*.

String10. Решение этого задания приводится в п. 2.2.1.

String11–12. Ср. с первым вариантом решения String10 в п. 2.2.1. В String12 следует вначале создать вспомогательную строку, содержащую *N* символов «*», а затем использовать ее при формировании требуемой строки.

String13–15. См. указания к String6 и String7.

String16–18. См. указания к String3–5, String6 и String7. При преобразовании символов надо учитывать, что для латинских букв и русских букв кодовое смещение между одинаковыми прописными и строчными буквами равно 32 (см. п. 5.4).

String19. Решение этого задания приводится в п. 2.2.2.

String20–21. Для получения строкового представления числа примените *строковый поток вывода* (см. п. 5.15). Обратите внимание на то, что в заданиях надо вывести не строку, а *последовательность символов*. Для String21 см. также решение String10.

String22–23. Для того чтобы по символу *c*, изображающему цифру, определить соответствующее число, можно воспользоваться либо строковым потоком ввода (см. решение String19), либо следующим выражением: *static_cast<int>(c) – static_cast<int>('0')*. В String23 удобно вначале обработать первый символ строки (то есть первую цифру), а затем в цикле обрабатывать *по два символа*: знак операции и очередную цифру.

String24–25. В String24 надо на основе анализа исходной строки сформировать требуемое целое число, которое затем преобразовать в строку с помощью строкового потока вывода. В String25, наоборот, удобнее вначале преобразовать исходную строку в целое число (с помощью строкового потока ввода), а затем на основе анализа этого числа сформировать его двоичное представление в виде строки.

String26. Используйте функцию *string::erase* и операцию сцепления.

String27. Используйте функцию *string::substr* и операцию сцепления.

String28–30. Просмотр символов исходной строки следует проводить *с конца*, применяя операцию индексирования (см. указание к String7); благодаря просмотру с конца, внесенные в строку изменения *не повлияют* на расположение тех символов, которые еще не проанализированы. Для вставки символов и строк используйте функцию *string::insert*.

String31. Используйте функцию *string::find*.

String32. Используйте в цикле *while* вариант функции *string::find с двумя параметрами*: на каждой последующей итерации цикла начинайте поиск с позиции, следующей за ранее найденным вхождением подстроки S_0. Цикл завершится, когда при очередном вызове функция *string::find* вернет *string::npos*.

String33. Используйте функции *string::find* и *string::erase*.

String34. Используйте функции *string::rfind* и *string::erase*.

String35. Используйте в цикле *while* функции *string::find* и *string::erase*.

String36–38. Ср. String36 с String33, String37 с String34, String38 с String35. Для вставки подстроки S_2 в строку S используйте функцию *string::insert*.

String39–40. Определите позицию первого пробела (с помощью функции *string::find*) и удалите все символы строки до первого пробела включительно (с помощью функции *string::erase*). Теперь для поиска следующего пробела (в String39) можно использовать функцию *string::find* (так как после удаления начального фрагмента строки этот пробел станет *первым* пробелом), а для поиска последнего пробела (в String40) — функцию *string::rfind*. Следует также учитывать, что исходная строка может содержать *единственный* пробел.

String41–46. Решение String41 приводится в п. 2.2.3. Вариант этого решения, использующий выделение слова из исходной строки с его последующим удалением, легко модифицировать для остальных заданий этой группы.

Более эффективными являются решения, основанные на *посимвольном анализе* исходной строки (как во втором варианте решения String41).

String47. Ср. с String41. Действуя, как в первом варианте решения String41, можно выделять слова из исходной строки и сразу добавлять их к новой строке, разделяя символами «.». Опишем другой способ решения, основанный на посимвольном анализе (как во втором варианте решения String41). Просматриваем исходную строку *с конца* в поисках пробела; найдя пробел, действуем следующим образом: если перед найденным пробелом также находится пробел, то удаляем найденный пробел, в противном случае заменяем найденный пробел на символ «.».

String48–49. Используйте посимвольный анализ строки (как во втором варианте решения String41). В String48 просмотр строки выполняйте от начала к концу, обрабатывая нужным образом следующие ситуации: 1) найдено начало очередного слова; 2) найдена буква, совпадающая с начальной буквой текущего слова. В String49 просмотр строки выполняйте от конца к началу, обрабатывая следующие ситуации: 1) найден конец очередного слова; 2) найдена буква, совпадающая с конечной буквой текущего слова.

String50–51. Используйте выделение слов из исходной строки (как в первом варианте решения String41). В String50 выделенное слово можно сразу добавлять *в начало* результирующей строки; в String51 необходимо сформировать вспомогательный *массив слов*, упорядочить его по алфавиту (используя любой из вариантов сортировки, описанных в Array112–114) и после этого объединить отсортированные слова в результирующую строку.

String52. Для нахождения начала очередного слова используйте посимвольный анализ строки (как во втором варианте решения String41). О преобразовании строчных букв в прописные (заглавные) см. указание к String16–18.

String53–54. Поскольку ни знаки препинания, ни гласные буквы не располагаются в кодовой таблице символов подряд (см. п. 5.4), можно сформировать

вспомогательную строку, содержащую все требуемые символы, и при просмотре символов строки проверять, принадлежат ли они этой строке, используя функцию *string::find*. Проверять, является ли символ знаком пунктуации можно также с помощью функции *ispunct* из заголовочного файла *cctype*.

String55–56. Эти задания, подобно String41–46, можно решить, либо выделяя слова из исходной строки, либо организуя ее посимвольный просмотр (см. варианты решения String41). В отличие от String41–46, разделителями слов здесь являются не только пробелы, но и *знаки препинания*, для их учета см. указание к String53–54.

String57. Ср. с String47.

String58–61. Имя диска отделяется от пути двоеточием и обратной косой чертой «\», путь отделяется от имени файла обратной косой чертой, имя файла отделяется от расширения точкой, например: «C:\Dir1\File1.txt». Следует иметь в виду, что косая черта также разделяет каталоги, перечисленные в пути («C:\Dir1\Dir2\Dir3\File1.txt»), что файл может находиться непосредственно в корневом каталоге («C:\File1.txt») и что расширение файла может отсутствовать («C:\Dir1\File1») или быть пустым («C:\Dir1\File1.»).

String62. См. указание к String3–5. Для обработки букв из разных диапазонов следует использовать вложенные условные операторы (см. указание к String6).

String63–65. Ср. с String62. В данном случае диапазоны букв, которые должны перемещаться по алфавиту «вперед» и «назад», зависят от значения числа *K*. При выполнении задания можно также воспользоваться «универсальной» формулой, применимой как для букв, перемещаемых «вперед», так и для букв, перемещаемых «назад» (в этой формуле будет присутствовать операция взятия остатка от деления).

String66. См. указание к Array11–15.

String67. На каждой итерации цикла удобно обрабатывать по *два* символа исходной строки: последний и первый, предпоследний и второй и т. д. (именно в таком порядке символы должны добавляться в конец результирующей строки). Для строк нечетной длины необходимо предусмотреть особую обработку их *среднего* символа, который должен стать *последним* символом результирующей строки.

String68. Поскольку при проверке упорядоченности необходимо учитывать *только буквы*, удобно завести вспомогательную символьную переменную, в которой хранить последний просмотренный символ строки, являющийся буквой, и использовать эту переменную при анализе следующей обнаруженной буквы.

String69. Заведите вспомогательную переменную-счетчик с нулевым начальным значением; в ходе просмотра строки увеличивайте эту переменную на 1 при обнаружении открывающей скобки и уменьшайте на 1 при появлении скобки закрывающей. При правильной расстановке скобок счетчик в ходе просмотра строки не должен принимать отрицательные значения, а после завершения просмотра должен равняться нулю.

String70. В данном случае нельзя обойтись вспомогательными переменными-счетчиками (как в String69), поскольку с их помощью невозможно распознать ошибочную ситуацию вида «[a(b]c)». Можно использовать вспомогательную *строку*, в которую записывать *все* открывающие скобки, а при появлении закрывающей скобки сравнивать ее с последней скобкой из вспомогательной строки: если скобки имеют одинаковый тип, то из вспомогательной строки удаляется последняя скобка и проверка продолжается; если же скобки имеют различный тип, значит, текущая закрывающая скобка является ошибочной. Требуется также обеспечить контроль за ситуацией, когда число закрывающих скобок не равно числу открывающих.

2.4. Тест рубежного контроля

Тест содержит 6 заданий, на выполнение которых отводится 3 минуты. Выберите правильный, по вашему мнению, вариант ответа и отметьте его любым значком в бланке ответов.

<table>
<tr><td colspan="4">1. Укажите функцию, которая не объявляется в заголовочном файле cctype.</td></tr>
<tr><td>(1)</td><td>isdigit</td><td>(2)</td><td>isspace</td></tr>
<tr><td>(3)</td><td>isalpha</td><td>(4)</td><td>ispunkt</td></tr>
<tr><td colspan="4">2. Укажите неверное утверждение.</td></tr>
<tr><td>(1)</td><td>Строки индексируются с 0.</td><td>(2)</td><td>Последний символ строки s имеет индекс s.length().</td></tr>
<tr><td>(3)</td><td>С помощью индексного доступа можно изменять символы строки.</td><td>(4)</td><td>Строки можно изменять.</td></tr>
<tr><td colspan="4">3. Укажите, каким будет значение строки s после выполнения следующего фрагмента кода:
<code>string s(" ABC ");</code>
<code>s.erase(0,1);</code>
<code>s.erase(4,1);</code></td></tr>
<tr><td>(1)</td><td><code>"ABC "</code></td><td>(2)</td><td><code>" ABC "</code></td></tr>
<tr><td>(3)</td><td><code>"ABC"</code></td><td>(4)</td><td>Возникнет ошибка времени выполнения</td></tr>
<tr><td colspan="4">4. Какое значение возвращает функция string::find при неудачном поиске подстроки в строке?</td></tr>
<tr><td>(1)</td><td>string::eof</td><td>(2)</td><td>string::npos</td></tr>
<tr><td>(3)</td><td>string::end</td><td>(4)</td><td>число –1</td></tr>
<tr><td colspan="4">5. Укажите неверное утверждение.</td></tr>
</table>

(1)	Строковый поток ввода можно использовать для преобразования целого числа в строку.	(2)	Строковый поток вывода можно использовать для преобразования целого числа в строку.
(3)	Строковый поток ввода можно использовать для преобразования строки в целое число.	(4)	Строковый поток вывода можно использовать для преобразования вещественного числа в строку.

6. Дан фрагмент программы:

```
string s(10, '*');
s.erase(0, s.find('!'));
```

Укажите, что произойдет при обработке этого фрагмента.

(1)	Произойдет ошибка компиляции.	(2)	Произойдет ошибка времени выполнения.
(3)	Из строки будет удален первый символ.	(4)	Из строки будут удалены все символы.

3. Модуль № 3. Двоичные файлы

3.1. Комплексная цель

Изучить средства стандартной библиотеки C++, предназначенные для обработки двоичных файлов, в частности, их создания и преобразования. Рассмотреть особенности нетипизированных двоичных файлов и двоичных файлов, содержащих символьные данные.

3.2. Содержание

При выполнении заданий, связанных с обработкой файлов, будем использовать файловые потоки ввода-вывода, реализованные в стандартном заголо-

вочном файле *fstream*, не применяя стандартные функции ввода-вывода языка С (такие, например, как *fopen* и *fprintf*).

Для подключению заголовочного файла *fstream* к программе, выполняющей задание на обработку файлов, в начало программы следует добавить директиву

```
#include <fstream>
```

Во всех примерах решения заданий на обработку файлов предполагается, что в тексте файла, содержащего функцию *solve*, указана данная директива.

3.2.1. Создание файла, ввод и вывод его элементов: File48

File48. Даны три файла целых чисел одинакового размера с именами S_A, S_B, S_C и строка S_D. Создать новый файл с именем S_D, в котором чередовались бы элементы исходных файлов с одним и тем же номером:

$$A_1, \quad B_1, \quad C_1, \quad A_2, \quad B_2, \quad C_2, \quad \ldots .$$

Напомним, что программу-заготовку для решения данного задания можно создать с помощью ярлыка Load.lnk. В состав данной заготовки будет входить файл File48.cpp. Функция *solve*, описанная в этом файле, будет содержать вызов функции *task* с параметром "File48", инициализирующий данное задание (см. текст аналогичного файла-заготовки для задания Begin3, приведенный в п. 1.2.1 пособия [1]).

После запуска этой программы-заготовки на экране появится окно, подобное приведенному на рис. 1.

В первой строке раздела исходных данных указаны имена трех исходных файлов (S_A, S_B и S_C) и одного результирующего (S_D). В трех последних строках раздела исходных данных показано содержимое исходных файлов. Для отображения содержимого каждого файла отводится по одной строке. Элементы файлов изображаются бирюзовым цветом, чтобы подчеркнуть их отличие от обычных исходных данных (желтого цвета) и комментариев (светло-серого цвета).

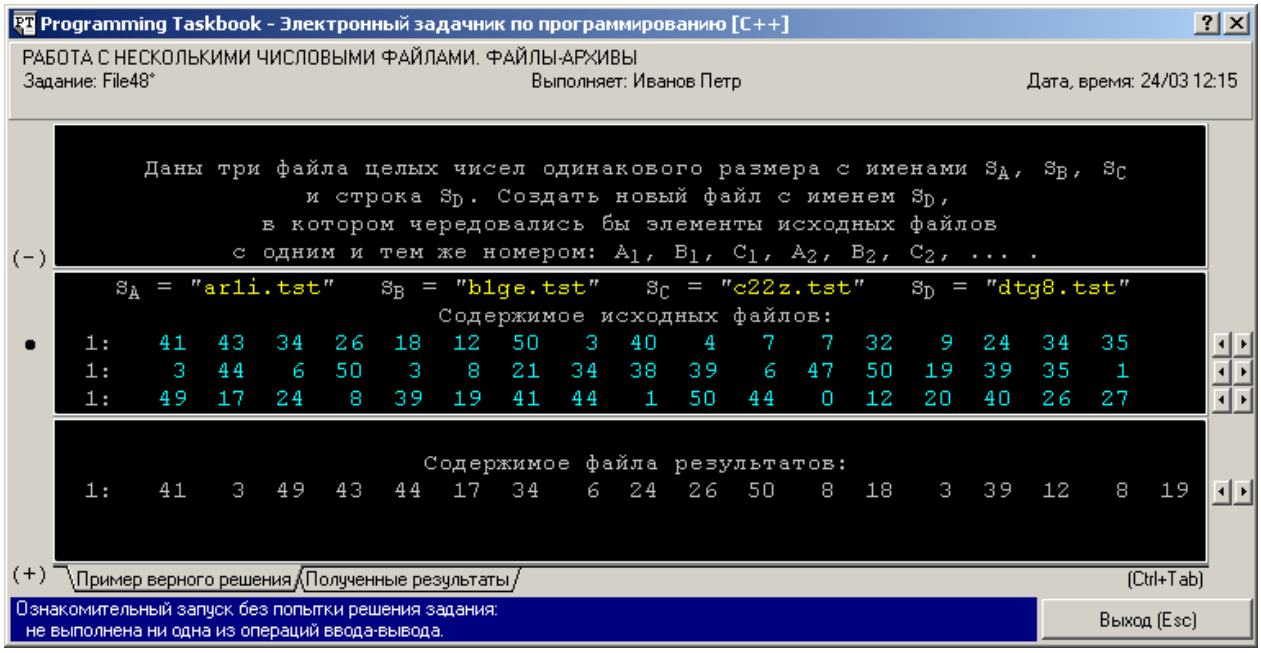

Рис. 1.

Поскольку размер файлов, как правило, превышает количество элементов, которое может уместиться на одной экранной строке, предусмотрена возможность *прокрутки* (листания) элементов файла с помощью мыши или клавиатуры. Для листания содержимого файла с помощью мыши достаточно щелкнуть на одной из двух кнопок ◂▸, расположенных справа от строки с содержимым нужного файла. С помощью клавиатуры можно выполнять поэлементную или «постраничную» прокрутку. Для поэлементной прокрутки используются клавиши [→] или [↓] (прокрутка вперед) и [←] или [↑] (прокрутка назад); для «постраничной» прокрутки, обеспечивающей переход к новой «порции» элементов, умещающейся на экранной строке, используются клавиши [Ctrl]+[→] или [PgDn] (прокрутка вперед) и [Ctrl]+[←] или [PgUp] (прокрутка назад). Кроме того, клавиша [Home] обеспечивает перемещение на начало, а клавиша [End] — на конец файла[1].

[1] Режим листания предусмотрен также для ситуации, когда текст *формулировки задания* содержит более 5 строк. В этом случае кнопки, аналогичные кнопкам листания для файлов, появляются рядом с разделом формулировки задания. Для прокрутки текста задания, помимо этих кнопок, можно использовать все описанные выше клавиатурные комбинации.

Если окно программы содержит несколько строк с файловыми данными, то прокрутка с помощью клавиатуры выполняется для *выделенного* файла (выделенный файл помечается в окне символом •, расположенным слева от файловой строки). Для смены выделенного файла предусмотрены клавиши [+] и [–]: первая из них обеспечивает циклический перебор всех отображенных на экране файлов в направлении сверху вниз, а вторая — в направлении снизу вверх.

Для того чтобы определить, какие элементы файла отображаются в данный момент на экране, предусмотрен *указатель текущей позиции*, значение которого равно номеру первого из элементов, отображенных на экране. Данный указатель располагается в начале строки с элементами файла и отделяется от них двоеточием. Нумерация элементов файла начинается с единицы.

Следует подчеркнуть, что ни выделение файла, ни прокрутка его элементов никак не влияют на сам файл, находящийся на диске: все эти действия проводятся над *копиями* файловых данных. Отметим также, что при каждом запуске программы с учебным заданием исходные файлы создаются под *новыми* именами и заполняются *новыми* данными, а при завершении программы (после анализа решения) как исходные, так и результирующие файлы *уничтожаются*.

Вернемся к нашей программе, только что запущенной на выполнение. Так как в функции *solve* не выполняются действия по вводу-выводу данных, в разделе результатов приводится пример верного решения (в нашем случае в этом разделе указываются числа, которые должны содержаться в результирующем файле при правильном решении задания). Чтобы подчеркнуть, что отображаемые в разделе результатов данные являются лишь *примером* верного решения и не связаны с каким-либо дисковым файлом, они изображаются светло-серым цветом, как обычные комментарии. Для этих данных также доступна прокрутка.

Запуск нашей программы был признан *ознакомительным* (и поэтому правильность решения не анализировалась), так как в ходе ее выполнения не было

введено ни одного элемента исходных данных. Добавим в функцию *solve* фрагмент, позволяющий ввести имена исходных файлов и связать с этими файлами соответствующие *файловые потоки*. Поскольку мы собираемся работать с четырьмя файлами, удобно предусмотреть *массив* потоков (как и в предыдущих примерах решений, будем приводить только содержимое функции *solve*):

```
task("File48");
fstream* f = new fstream[4];
int i;
for (i = 0; i < 3; ++i)
{
    string s;
    pt >> s;
    f[i].open(s.c_str(), ios::binary | ios::in);
}
delete[] f;
```

Мы намеренно ограничились *тремя* итерациями цикла, оставив непрочитанным имя результирующего файла. Считывание имен файлов производится в одну и ту же переменную *s*, поскольку после связывания файла, имеющего имя *s*, с файловым потоком *f*[*i*] все остальные действия с данным файлом в нашей программе будут осуществляться с использованием потока *f*[*i*], без обращения к имени файла.

Для ввода имени файла использовалась переменная типа *string*, в то время как в функции *open* первый параметр (имя файла) должен представлять собой *массив символов* (типа *char**). Для получения такого массива из исходной строки типа *string* предназначена функция *c_str* класса *string* (ранее эту функцию мы использовали в п. 2.2.2).

Поскольку массив файловых потоков был создан с помощью операции *new*, в конце функции *solve* необходимо освободить выделенную память с помощью операции *delete*[].

☑ Массив файловых потоков можно было бы описать следующим образом:

```
fstream f[4];
```

В этом случае в конце программы не нужно указывать оператор *delete*[] *f*. Однако компиляция указанного варианта описания массива *f* в среде C++Builder приводит к сообщению об ошибке (в средах Visual C++ и Visual Studio ошибки не возникает).

☑ Имена файлов можно было бы вводить в переменную типа *char** или *char*[*n*], если размер *n* строки заранее известен, например:

```
char s[13];
```

В этом случае в качестве первого параметра функции *open* можно указать саму переменную *s*. Размер массива символов *s* выбран таким образом, чтобы в массиве можно было сохранить имя файла в стандартном DOS-формате «8.3» (собственно имя файла длины не более 8 символов, точка и расширение длины не более 3 символов; при этом учитывается также, что любая строка в массиве символов должна быть дополнена нулевым символом). Такого размера достаточно для выполнения учебных заданий, так как имена всех исходных и результирующих файлов, предлагаемых в заданиях, имеют формат «8.3». В случае, если длина имени файла заранее неизвестна, можно использовать макрос *FILENAME_MAX* из стандартного заголовочного файла *cstdio*, возвращающий максимально возможную длину имени файла.

Запуск нового варианта программы уже не будет считаться ознакомительным, поскольку в программе выполняется ввод исходных данных. Так как имя результирующего файла осталось непрочитанным, этот вариант решения будет признан неверным и приведет к сообщению «*Введены не все требуемые исходные данные. Количество прочитанных данных: 3 (из 4)*».

Изменим в программе заголовок цикла так, чтобы в цикле выполнялись не 3, а 4 итерации:

```
for (int i = 0; i < 4; ++i)
```

Теперь все данные, необходимые для выполнения задания, в программу введены. Однако при запуске программы результирующий файл создан не бу-

дет. Поэтому решение опять будет признано ошибочным с диагностикой «*Результирующий файл не найден*».

☑ Отметим, что при выполнении последней, четвертой, итерации цикла попытка открыть файл *приведет к ошибке времени выполнения*, поскольку для этого файла, как и для всех предыдущих, указан режим открытия *для чтения* (*ios::in*), в то время как файл с указанным именем не существует. Однако подобные ошибки в языке C++ не приводят к аварийному завершению программы.

Для того чтобы избежать ошибки, возникшей на предыдущем шаге выполнения задания, отсутствующий файл результатов следует открыть не для чтения, а для записи. Далее, после завершения работы с файлами, открытыми в программе, их необходимо закрыть функцией *close*. Добавим в функцию *solve* соответствующие операторы; в результате тело этой функции примет следующий вид:

```
task("File48");
fstream *f = new fstream[4];
for (int i = 0; i < 4; ++i)
{
    string s;
    pt >> s;
    if (i < 3)
        f[i].open(s.c_str(), ios::binary | ios::in);
    else
        f[i].open(s.c_str(), ios::binary | ios::out);
}
// Обработка файловых данных
for (int i = 0; i < 4; ++i)
    f[i].close();
delete[] f;
```

Комментарий «*Обработка файловых данных*» расположен в том месте программы, в котором можно выполнять операции ввода-вывода для всех четырех файлов: они уже открыты функцией *open* и еще не закрыты функцией *close*.

При выполнении данного варианта программы результирующий файл будет создан, однако он останется *пустым*, то есть не содержащим ни одного элемента. Поэтому в окне задачника на информационной панели появится сообщение «*Ошибочное решение*», а строка, которая должна содержать элементы результирующего файла, примет вид

```
EOF:
```

Особое значение EOF (*End Of File* — «конец файла») для указателя текущей файловой позиции означает, что данный файл существует, но не содержит ни одного элемента. Таким образом, нам осталось реализовать фрагмент алгоритма, обеспечивающий ввод и вывод файловых данных.

Во всех ранее рассмотренных вариантах программы мы не использовали файловые операции ввода-вывода, поэтому тип файловых элементов нас не интересовал. Однако при чтении данных из файла (и при их записи в файл) *крайне важно* правильно указывать тип файловых элементов. Чтобы продемонстрировать это на примере нашей программы, попытаемся прочесть из исходных файлов по одному элементу *вещественного* типа и запишем прочитанные элементы в файл результатов. Для этого заменим строку с комментарием «*Обработка файловых данных*» на следующий фрагмент:

```
for (int i = 0; i < 3; ++i)
{
    double x;
    f[i].read((char *)&x, sizeof(x));
    f[3].write((char *)&x, sizeof(x));
}
```

Данный фрагмент обеспечивает считывание *одного вещественного* элемента для каждого из трех исходных файлов и запись этих элементов в результирующий файл (в требуемом порядке). Подчеркнем, что мы *неправильно* указали тип файловых элементов; тем не менее, компиляция программы пройдет успешно, а после ее запуска не возникнет ошибок времени выполнения.

☑ Для чтения и записи двоичных данных в файловых потоках предусмотрены функции *read* и *write* соответственно. Они могут использоваться для обработки данных *любого* типа, однако для этого приходится преобразовывать эти данные к типу *char**, а также указывать размер этих данных (в качестве второго параметра). Для преобразования переменной *x* любого типа к типу *char** достаточно использовать следующую конструкцию: (*char* *)*&x* (обратите внимание на символ &). Более предпочтительной, но достаточно громоздкой, является операция *static_cast*: *static_cast<char*>(&x)*. Для определения размера переменной *x* достаточно воспользоваться операцией *sizeof*. Заметим, что эту операцию можно применять и к именам типов, например, *sizeof*(*double*) (в рассматриваемых реализациях C++ это значение равно 8).

Результат работы программы будет неожиданным: судя по экранной строке с содержимым результирующего файла, в него будут записаны не три, а *шесть элементов*, по два начальных элемента из каждого исходного файла. Объясняется это тем, что считывание из файла и последующая запись в файл одного «вещественного элемента» фактически приводит к считыванию и записи блока данных размером 8 байтов, содержащего *два* последовательных целочисленных элемента исходного файла.

Итак, мы выяснили, что ошибки, связанные с несоответствием типов файловых элементов, не выявляются при компиляции и не всегда приводят к ошибкам времени выполнения. Это следует иметь в виду, и при появлении «странных» результирующих данных начинать поиск ошибки с проверки типов файловых элементов.

Для исправления данной ошибки достаточно изменить описание переменной *x*:

```
int x;
```

После запуска исправленной программы мы получим все еще неверный, но вполне естественный результат: созданный файл будет содержать три элемента, совпадающих с начальными элементами исходных файлов.

При выполнении этого варианта программы следует обратить внимание на одну полезную возможность: при переключении между вкладками «Полученные результаты» и «Пример верного решения» (см. рис. 1) указатель текущей позиции для результирующего файла не меняется. Это облегчает поиск несоответствий между полученными и контрольными файловыми элементами.

Приведем, наконец, верное решение задания File48:

```
task("File48");
fstream *f = new fstream[4];
for (int i = 0; i < 4; ++i)
{
    string s;
    pt >> s;
    if (i < 3)
        f[i].open(s.c_str(), ios::binary | ios::in);
    else
        f[i].open(s.c_str(), ios::binary | ios::out);
}
while (f[0].peek() != -1)
    for (int i = 0; i < 3; ++i)
    {
        int x;
        f[i].read((char *)&x, sizeof(x));
        f[3].write((char *)&x, sizeof(x));
    }
for (int i = 0; i < 4; ++i)
    f[i].close();
```

```
delete[] f;
```

От предыдущего варианта данное решение отличается добавлением цикла *while*, который обеспечивает считывание всех элементов из исходных файлов (напомним, что по условию задания все исходные файлы имеют *одинаковый размер*) и запись их в результирующий файл в нужном порядке. В условии цикла *while* использована функция *peek*, которая возвращает код очередного символа из файла, не считывая данный символ (и, следовательно, не перемещая файловый указатель), причем если достигнут конец файла, то данная функция возвращает –1.

☑ В классе *fstream* имеется функция *eof*, возвращающая значение *true*, если достигнут конец файла, и *false* в противном случае. Данную функцию применять менее удобно, чем функцию *peek* из-за следующей особенности ее реализации: функция *eof* возвращает *true* только *после попытки выполнить операцию чтения за концом файла*, таким образом, сразу после чтения последнего элемента данная функция будет по-прежнему возвращать значение *false*.

После запуска этого варианта мы получим сообщение «*Верное решение. Тест номер 1 (из 5)*», а после пяти запусков — сообщение «*Задание выполнено!*».

Заметим, что при решении задания File48 мы не пользовались потоком *pt* для вывода результатов. Это объясняется тем, что ввод и вывод *файловых элементов* при решении любого учебного задания осуществляется с использованием *стандартных* функций библиотеки C++.

После завершения выполнения задания можно убедиться, что в рабочем каталоге не осталось ни одного из тех файлов (как исходных, так и результирующих), которые создавались при запусках программы. Таким образом, «засорения» диска ненужными в дальнейшем файлами не происходит.

3.2.2. Преобразование файла: File25

File25. Дан файл вещественных чисел. Заменить в нем все элементы на их квадраты.

В задании File25, в отличие от рассмотренного ранее задания File48, требуется не *создать* новый файл, а *преобразовать* уже имеющийся.

Мы встречаемся здесь с ситуацией, когда один и тот же файл является и исходным, и (после преобразования) результирующим. В простых случаях, когда требуемое преобразование не связано с удалением, вставкой или перемещением элементов файла, это преобразование можно провести непосредственно в исходном файле, открыв его одновременно для чтения и для записи. Приведем один из вариантов подобного решения:

```
task("File25");
string s;
pt >> s;
fstream f(s.c_str(), ios::binary |
    ios::in | ios::out);
f.seekg(0, ios::end);
int len = f.tellg(),
    size = sizeof(double);
f.seekg(0);
for (int i = 0; i < len / size; ++i)
{
    f.seekg(f.tellg());
    double x;
    f.read((char *)&x, size);
    x = x * x;
    f.seekg(size * i);
    f.write((char *)&x, size);
```

```
}
f.close();
```

Здесь мы воспользовались другим вариантом открытия файлового потока, основанным не на функции *open*, а на использовании *конструктора* класса *fstream*: достаточно при создании объекта *f* указать в скобках те же параметры, которые требуются и для функции *open*. Данный вариант открытия файлового потока удобно использовать, если к моменту описания файлового потока уже известно имя связываемого с ним файла.

В этом решении используется возможность *прямого доступа* к байтам двоичных файлов, то есть доступа к байту с нужным порядковым номером. Эта возможность реализуется с помощью двух пар функций: *seekg–seekp* и *tellg–tellp*. Функции *seekg*(*num*) и *seekp*(*num*) перемещают файловый указатель на байт файла с номером *num* (нумерация начинается от нуля), а функции *tellg* и *tellp* возвращают номер текущего байта (то есть байта, на котором расположен файловый указатель). Имеется также перегруженный вариант функций *seekg* и *seekp* с *двумя* параметрами; в этом варианте первый параметр *num* задает относительное смещение от базовой позиции (и может быть как положительным, так и отрицательным), а второй параметр определяет базовую позицию и может принимать три значения: *ios*::*beg* (базовая позиция совпадает с началом файла), *ios*::*end* (базовая позиция совпадает с концом файла) и *ios*::*cur* (базовая позиция совпадает с текущей позицией файлового указателя).

☑ Функции с суффиксом *g* реализованы для потоков, открытых на чтение (**g**et), а функции с суффиксом *p* — для потоков, открытых на запись (**p**ut). Однако если файл открыт одновременно и на запись, и на чтение, то соответствующие потоки всегда синхронизированы, поэтому для управления файловым указателем можно использовать любой из двух вариантов данных функций.

В программе использован цикл *for*, число итераций которого равно количеству вещественных чисел, содержащихся в файле. Для нахождения этого количества достаточно определить размер файла *f в байтах*, после чего поделить его

на размер в байтах вещественного числа (типа *double*). Поскольку в классе *fstream* не предусмотрено функции для определения размера файла, этот размер находится с помощью перемещения файлового указателя на конец файла (функцией *f.seekg*(0, *ios*::*end*)) и последующего определения текущей файловой позиции (функцией *f.tellg*()). Размер файла в байтах сохраняется в переменной *len*, размер переменной типа *double* — в переменной *size*. После этого файловый указатель возвращается на начало файла функцией *f.seekg*(0).

Прямой доступ к элементам файла используется и в самом цикле *for*. Это связано с тем, что после считывания каждого элемента файловый указатель автоматически перемещается к следующему элементу файла, и для того чтобы записать преобразованный элемент на прежнее место, приходится «возвращать» файловый указатель назад на позицию, которую занимал файловый указатель до считывания элемента (на *i*-й итерации цикла данная позиция равна *size* * *i*).

☑ При анализе программы возникает вопрос, зачем в начале цикла используется оператор *f.seekg*(*f.tellg*()), который, казалось бы, ничего не делает (так как он «перемещает» файловый указатель на ту позицию, в которой он и так находится). Ответ состоит в том, что в среде Visual Studio без данного оператора программа будет работать неверно. Следует заметить, что в среде Borland C++Builder программа правильно работает и без этого «лишнего» оператора.

☑ Возможен вариант решения данной задачи, в котором не требуется предварительного определения количества элементов в файле. В самом деле, для возврата файлового указателя на предыдущий элемент достаточно выполнить смещение на величину *–size* относительно *текущей* позиции файлового указателя. Таким образом, переменная цикла *i* оказывается ненужной, и вместо цикла *for* можно использовать цикл *while*, который должен выполняться, пока не будет достигнут конец файла. Мы получаем следующий вариант решения:

```
task("File25");
string s;
pt >> s;
fstream f(s.c_str(), ios::binary | ios::in | ios::out);
```

```
while (f.peek() != -1)
{
    double x;
    f.seekg(f.tellg());
    f.read((char *)&x, sizeof(x));
    x = x * x;
    f.seekg(-8, ios::cur);
    f.write((char *)&x, sizeof(x));
}
f.close();
```

К сожалению, следует отметить, что данный вариант будет неверно работать в среде Borland C++Builder.

☑ Проблемы, связанные с функцией *seekg*, лежат глубже, чем это может показаться. Так, необходимость в «фиктивном» вызове *seek* возникает даже в ситуации, когда вместо классов-потоков используются файловые функции из стандартного заголовочного файла *cstdio*:

```
task("File25");
string s;
pt >> s;
FILE* f = fopen(s.c_str(),"rb+");
while (true)
{
    double x;
    fseek(f, ftell(f), SEEK_SET);
    fread(&x, 8, 1, f);
    if (feof(f))
        break;
    x = x * x;
    fseek(f, -8, SEEK_CUR);
    fwrite(&x, 8,1,f);
}
fclose(f);
```

Данная программа правильно выполняет задание, однако только при наличии выделенный полужирным шрифтом «фиктивного» оператора (причем как в среде Visual Studio, так и в среде Borland C++Builder).

Итак, мы убедились, что «прямой» способ преобразования исходного файла может создавать проблемы даже при решении достаточно простых задач. Еще сложнее реализовать его в ситуации, когда преобразования *приходится изменять порядок следования элементов*, в частности, удалять некоторые элементы или добавлять новые.

Поэтому для преобразования файлов обычно используют другой подход, который заключается в использовании *вспомогательного файла*. Приведем соответствующий вариант решения задания File25:

```
task("File25");
string s1, s2 = "$F25$.tmp";
pt >> s1;
ifstream f1(s1.c_str(), ios::binary);
ofstream f2(s2.c_str(), ios::binary);
while (f1.peek() != -1)
{
    double x;
    f1.read((char *)&x, sizeof(x));
    x = x * x;
    f2.write((char *)&x, sizeof(x));
}
f1.close();
f2.close();
remove(s1.c_str());
rename(s2.c_str(), s1.c_str());
```

Данный вариант решения использует *последовательный* доступ к элементам файлов, причем для исходного файлового потока *f1* — только для чтения, а

для вспомогательного файлового потока *f2* — только для записи. Имя вспомогательного файла следует подобрать таким образом, чтобы обеспечить его *уникальность*. В нем можно использовать символы *# $ % & ( )*, редко встречающиеся в именах обычных файлов, а также стандартное расширение .tmp для временных файлов.

☑ Поскольку в программе использованы потоки *istream* и *ostream*, при открытии файлов не требуется уточнять способ доступа (поток *ifstream* открывается на чтение, а поток *ofstream* — на запись), однако указывать двоичный режим *ios*::*binary* по-прежнему необходимо.

После заполнения вспомогательного файла *и закрытия обоих файлов* исходный файл уничтожается, а имя вспомогательного файла заменяется на имя исходного (для выполнения этих действий используются функции *remove* и *rename* из стандартного заголовочного файла *cstdio*). В результате на диске остается один преобразованный файл, имя которого совпадает с именем исходного файла. Заметим, что без предварительного уничтожения исходного файла вызов функции *rename* приведет к ошибке, так как при переименовании файла ему нельзя присваивать имя существующего файла (в результате требуемое переименование файла выполнено не будет).

Следует иметь в виду, что преобразовать *текстовый* файл можно *только* с помощью вспомогательного текстового файла. «Прямой» способ преобразования текстовых файлов невозможен, поскольку для них предусмотрен только *последовательный* способ доступа к элементам-строкам.

3.2.3. Нетипизированные двоичные файлы: File43

File43. Дан файл произвольного типа. Создать его копию с новым именем.

Задание File43 относится к группе заданий на обработку *нетипизированных двоичных файлов*. В этих заданиях тип элементов исходных файлов не известен, и требуется выполнить действие над файлами *в целом*: скопировать содержимое файла в новый файл, добавить это содержимое в конец другого фай-

ла и т. д. В подобной ситуации файл следует рассматривать как *последовательность байтов* и осуществлять действия по вводу-выводу *блоков данных* достаточно большого размера.

Приводимое ниже решение задания File43 является хорошей иллюстрацией особенностей работы с нетипизированными двоичными файлами:

```
task("File43");
string s1,s2;
pt >> s1 >> s2;
ifstream f1(s1.c_str(), ios::binary);
ofstream f2(s2.c_str(), ios::binary);
char buf[1024];
while (f1.peek() != -1)
    f2.write(buf, f1.readsome(buf, 1024));
f1.close();
f2.close();
```

Использованная в программе функция *readsome* считывает из исходного файла 1024 байта (или менее, если в исходном файле недостаточно данных; максимальное количество считываемых данных указывается в качестве второго параметра функции) и помещает их в символьный массив *buf*. Возвращаемым значением функции *readsome* является количество фактически считанных байтов. Функция *write* записывает в результирующий файл байты, содержащиеся в массиве *buf*. Количество записываемых байтов указывается во втором параметре, в качестве которого мы поместили вызов функции *readsome*. Таким образом, в одном операторе мы выполнили и ввод, и вывод блока данных размером 1 килобайт.

Аналогичный подход, основанный на считывании блоков данных, оказывается полезным во многих других ситуациях, например, при объединении файлов (см. задания File46 и File47).

3.2.4. Символьные и строковые файлы: File63

Символьными файлами называют двоичные файлы, содержащие отдельные символы (то есть данные типа *char*).

Строковыми файлами называют двоичные файлы, содержащие строковые данные. При этом предполагается, что для хранения каждой строки выделяется память *одинакового размера* (начальная часть выделенной памяти отводится для хранения строки, а ее «остаток» не используется). Благодаря этому при обработке файла можно использовать прямой доступ к его элементам-строкам (так как определить начальную позицию *i*-й строки можно, не считывая предыдущие строки, а просто умножив размер памяти, выделенной для отдельной строки, на ее номер *i*). Кроме того, для строкового файла можно одновременно выполнять и операции чтения, и операции записи данных. Это отличает строковые файлы от текстовых (см. модуль № 4), при обработке которых возможен только последовательный доступ, причем либо только на чтение, либо только на запись. К числу недостатков строковых файлов следует отнести невозможность их просмотра и редактирования в обычных текстовых редакторах, а также их больший размер по сравнению с текстовыми файлами (увеличение размера обусловлено наличием неиспользуемых участков памяти, имеющихся после каждой строки, хранящейся в файле).

При выполнении заданий на строковые файлы с использованием электронного задачника Programming Taskbook следует учитывать, что для хранения каждой строки в строковом файле всегда отводится 80 байтов, независимо от фактической длины строки (фактическая длина определяется по положению завершающего ее нулевого символа '\0'). Поэтому для операций ввода-вывода, связанных со строковыми файлами, необходимо использовать переменные типа char[80]. При открытии строковых файлов, как и при открытии других двоичных файлов, необходимо указывать атрибут *ios*::*binary*. Для чтения строк из

строкового файла, как и для чтения данных из двоичных файлов других типов, следует использовать функцию *read*.

Подчеркнем, что условие, согласно которому длина элементов строковых файлов равна 80 символам, *накладывается задачником Programming Taskbook*. В обычных программах, не связанных с выполнением учебных заданий, можно создавать строковые файлы с элементами любой требуемой длины.

В качестве примера задания на обработку строковых файлов рассмотрим задание File63.

File63. Дано целое число K (> 0) и строковый файл. Создать два новых файла: строковый, содержащий первые K символов каждой строки исходного файла, и символьный, содержащий K-й символ каждой строки (если длина строки меньше K, то в строковый файл записывается вся строка, а в символьный файл записывается пробел).

```
task("File63");
int k;
string s1,s2,s3;
pt >> k >> s1 >> s2 >> s3;
ifstream f1(s1.c_str(), ios::binary);
ofstream f2(s2.c_str(), ios::binary),
    f3(s3.c_str(), ios::binary);
while (f1.peek() != -1)
{
    char s[80];
    char c = ' ';
    f1.read(s, 80);
    if (strlen(s) >= k)
    {
        s[k] = '\0';
```

```
            c = s[k - 1];
        }
        f2.write(s, 80);
        f3.write((char *)&c, 1);
    }
    f1.close();
    f2.close();
    f3.close();
```

Для уменьшения размера строки, содержащейся в символьном массиве, до *k* символов, достаточно записать нулевой символ в элемент массива с индексом *k* (подобное действие в программе выполняется только для строк, длина которых больше или равна *k*). Также обратите внимание на то, что при использовании в функции *write* в качестве первого параметра массива *s* типа *char*[] нет необходимости в его явном приведении к типу *char** (тогда как для других типов данных, в частности, для отдельного символа *c* типа *char*, такое преобразование является обязательным).

3.3. Проектное задание

3.3.1. Варианты

Выполните задачи группы File, указанные в вашем варианте проектного задания (формулировки задач приводятся в [3]). Если вы не получили вариант проектного задания, то выполните задачи из какого-либо варианта.

ВАРИАНТ 1	**ВАРИАНТ 2**
(1) Формирование, ввод-вывод: 9, 24	(1) Формирование, ввод-вывод: 10, 15
(2) Изменение: 28, 41, 50, 57	(2) Изменение: 27, 36, 50, 53
(3) Нетипизированные файлы: 42	(3) Нетипизированные файлы: 44
(4) Строковые файлы: 58, 69	(4) Строковые файлы: 59, 64

ВАРИАНТ 3 (1) Формирование, ввод-вывод: 2, 17 (2) Изменение: 32, 36, 51, 57 (3) Нетипизированные файлы: 45 (4) Строковые файлы: 60, 64	**ВАРИАНТ 4** (1) Формирование, ввод-вывод: 1, 18 (2) Изменение: 29, 40, 49, 55 (3) Нетипизированные файлы: 46 (4) Строковые файлы: 61, 72
ВАРИАНТ 5 (1) Формирование, ввод-вывод: 12, 22 (2) Изменение: 33, 40, 51, 53 (3) Нетипизированные файлы: 45 (4) Строковые файлы: 59, 70	**ВАРИАНТ 6** (1) Формирование, ввод-вывод: 7, 21 (2) Изменение: 35, 38, 49, 56 (3) Нетипизированные файлы: 47 (4) Строковые файлы: 59, 67
ВАРИАНТ 7 (1) Формирование, ввод-вывод: 11, 23 (2) Изменение: 26, 39, 49, 52 (3) Нетипизированные файлы: 46 (4) Строковые файлы: 58, 71	**ВАРИАНТ 8** (1) Формирование, ввод-вывод: 6, 16 (2) Изменение: 34, 38, 49, 55 (3) Нетипизированные файлы: 44 (4) Строковые файлы: 61, 65
ВАРИАНТ 9 (1) Формирование, ввод-вывод: 13, 14 (2) Изменение: 33, 37, 51, 54 (3) Нетипизированные файлы: 42 (4) Строковые файлы: 60, 73	**ВАРИАНТ 10** (1) Формирование, ввод-вывод: 3, 15 (2) Изменение: 31, 39, 51, 52 (3) Нетипизированные файлы: 44 (4) Строковые файлы: 61, 66
ВАРИАНТ 11 (1) Формирование, ввод-вывод: 8, 19 (2) Изменение: 34, 37, 50, 54 (3) Нетипизированные файлы: 47	**ВАРИАНТ 12** (1) Формирование, ввод-вывод: 4, 20 (2) Изменение: 30, 41, 50, 56 (3) Нетипизированные файлы: 44

(4) Строковые файлы: 60, 65	(4) Строковые файлы: 58, 68
ВАРИАНТ 13 (1) Формирование, ввод-вывод: 4, 14 (2) Изменение: 28, 38, 49, 56 (3) Нетипизированные файлы: 46 (4) Строковые файлы: 60, 67	**ВАРИАНТ 14** (1) Формирование, ввод-вывод: 6, 17 (2) Изменение: 26, 39, 51, 53 (3) Нетипизированные файлы: 44 (4) Строковые файлы: 61, 69
ВАРИАНТ 15 (1) Формирование, ввод-вывод: 8, 18 (2) Изменение: 33, 36, 49, 56 (3) Нетипизированные файлы: 47 (4) Строковые файлы: 60, 70	**ВАРИАНТ 16** (1) Формирование, ввод-вывод: 9, 19 (2) Изменение: 33, 37, 49, 52 (3) Нетипизированные файлы: 44 (4) Строковые файлы: 59, 68
ВАРИАНТ 17 (1) Формирование, ввод-вывод: 2, 16 (2) Изменение: 29, 41, 50, 55 (3) Нетипизированные файлы: 45 (4) Строковые файлы: 60, 64	**ВАРИАНТ 18** (1) Формирование, ввод-вывод: 11, 24 (2) Изменение: 30, 38, 51, 54 (3) Нетипизированные файлы: 44 (4) Строковые файлы: 61, 65
ВАРИАНТ 19 (1) Формирование, ввод-вывод: 3, 23 (2) Изменение: 34, 36, 51, 55 (3) Нетипизированные файлы: 46 (4) Строковые файлы: 59, 73	**ВАРИАНТ 20** (1) Формирование, ввод-вывод: 7, 15 (2) Изменение: 31, 40, 50, 54 (3) Нетипизированные файлы: 44 (4) Строковые файлы: 59, 71
ВАРИАНТ 21 (1) Формирование, ввод-вывод: 1, 15 (2) Изменение: 27, 40, 50, 53	**ВАРИАНТ 22** (1) Формирование, ввод-вывод: 10, 20 (2) Изменение: 32, 41, 50, 57

(3) Нетипизированные файлы: 42 (4) Строковые файлы: 58, 64	(3) Нетипизированные файлы: 45 (4) Строковые файлы: 58, 66
ВАРИАНТ 23 (1) Формирование, ввод-вывод: 12, 21 (2) Изменение: 35, 37, 49, 52 (3) Нетипизированные файлы: 42 (4) Строковые файлы: 58, 72	**ВАРИАНТ 24** (1) Формирование, ввод-вывод: 13, 22 (2) Изменение: 34, 39, 51, 57 (3) Нетипизированные файлы: 47 (4) Строковые файлы: 61, 65

3.3.2. Указания

File1. Проще всего «переложить» проверку допустимости имени файла на операционную систему, попытавшись открыть файл с этим именем в *режиме создания* (для этого достаточно создать объект класса *ofstream*, передав ему имя файла, после чего проверить состояние потока функцией *ofstream::fail*).

File4. Для того чтобы проверить, существует ли файл с указанным именем, достаточно создать поток для чтения данного файла и проверить состояние потока функцией *ofstream::fail*.

File5. По поводу проверки существования файла см. указание к File4. Для нахождения количества элементов файла следует поделить размер файла в байтах на размер элемента (4 байта в случае целочисленных элементов и 8 байтов в случае вещественных элементов). Простой способ определения размера файла в байтах приведен в первом варианте решения File25.

File6–7. В File6 для проверки существования элемента с требуемым номером сравните его с размером файла (см. указание к File5). Для перемещения к существующему элементу используйте функцию *seekg*.

File8. По поводу чтения элементов файла с требуемыми номерами см. указание к File6–7. Для записи элементов в новый файл следует создать файловый поток вывода *ofstream* и воспользоваться его функцией *write*.

File9. По поводу проверки существования файла см. указание к File4. Если связать существующий файл (независимо от его порядкового номера) с файловым потоком *f1*, а отсутствующий файл — с файловым потоком *f2*, то оставшаяся часть решения будет очень похожа на решение File8, в котором поток *f1* связан с первым файлом, а поток *f2* — со вторым.

File10. Считывайте элементы исходного файла *в обратном порядке* (используя функцию *seekg* со вторым параметром *ios::end*); это позволит заполнять результирующий файл *последовательно*, не выполняя действий по явному перемещению его файлового указателя.

File11. Удобно считывать по *два* элемента исходного файла, записывая по одному элементу в каждый из результирующих файлов. Повторять эти действия требуется $N/2$ раз, где N — количество элементов исходного файла. В случае, когда N — нечетное, следует отдельно обработать *последний* элемент исходного файла.

File13. См. указание к File10.

File15. См. указание к File11.

File16–17. Ср. с Array116. Поскольку в данном случае обрабатываются элементы не массива, а файла (и поэтому нельзя обеспечить *одновременный* доступ к элементам), прочитанный файловый элемент необходимо сохранять во вспомогательной переменной, которую в дальнейшем использовать при анализе следующего элемента.

File18–22. Организуйте последовательное считывание элементов файла с сохранением во вспомогательных переменных *трех* последних прочитанных элементов (при этом средний из этих элементов можно будет проверить на

локальный минимум/максимум/экстремум). При нахождении локальных минимумов/максимумов (File18–19 и File21) требуется также предусмотреть особую обработку первого и последнего элемента, которые имеют по одному соседу. При нахождении всех локальных *экстремумов* (File20 и File22) данные элементы анализировать не требуется, так как они *обязательно* являются локальными экстремумами (см. указание к Array36). Для File22 воспользуйтесь также указанием к File10.

File23–24. См. указания к Array37–39 и File18–22.

File25. Решение этого задания приводится в п. 3.2.2.

File26. Ср. с Array68.

File27. Порядок считывания элементов исходного файла задавайте с помощью функции *seekg*; прочитанные элементы записывайте *в конец* вспомогательного файла (см. последний вариант решения File25).

File28. Последовательно считывайте элементы исходного файла с сохранением во вспомогательных переменных трех последних прочитанных элементов, определяйте по этим трем элементам новое значение среднего из них и записывайте его во вспомогательный файл (см. последний вариант решения File25). Заметим, что задание можно решить и без вспомогательного файла, ограничившись несколькими вспомогательными переменными (ср. с Array78) и осуществляя прямой доступ к файловым элементам с помощью функции *seekg*.

File29–30. Воспользуйтесь *вспомогательным файлом* (как во втором варианте решения File25), переписывая в него те элементы, которые требуется сохранить.

File31–41. Воспользуйтесь *вспомогательным файлом* (как во втором варианте решения File25). Задания File36–37 несложно решить и без вспомогатель-

ного файла, используя прямой доступ к файловым элементам с помощью функции *seekg*.

File42. Открывать файлы для выполнения задания не требуется. Достаточно трижды выполнить переименование файлов с помощью функции *rename*, используя при переименовании вспомогательное имя файла (подобно тому, как при обмене содержимого двух переменных используется третья, вспомогательная переменная).

File43–45. Решение File43 приводится в п. 3.2.3. Задания File44–45 решаются аналогично; для определения размера файлов в этих заданиях используйте прием из первого варианта решения File25.

File46–47. Используйте функции r*ead* и w*rite* (как в решении File43). В File46 достаточно использовать *две* переменные класса *fstream*: одна связывается с создаваемым файлом результатов, а вторая (в цикле) последовательно связывается с каждым из исходных файлов для копирования данных из этого исходного файла в конец файла результатов. В File47 удобно перед дополнением одного из файлов создать его *копию* (см. решение File43), которую использовать в дальнейшем при дополнении другого файла.

File48–49. Решение File48 приводится в п. 3.2.1, задание File49 решается аналогично. При решении File49 удобно перед заполнением результирующего файла определить его размер (равный размеру самого короткого из исходных файлов — см. решение Minmax1 в [1], п. 4.2.2), после чего заполнить файл в цикле *for*. Для определения размера файлов в File49 используйте прием из первого варианта решения File25.

File50–51. Воспользуйтесь алгоритмом слияния, описанным в указании к заданиям Array63–64. В данном случае можно обойтись без специальной переменной для хранения информации о текущем элементе файла, так как текущий элемент определяется позицией файлового указателя. Далее, *барьер* в конце файла можно реализовать двумя способами: либо просто допол-

нить каждый из исходных файлов барьерным элементом, либо перед чтением элемента из файла в переменную проверять, не достигнут ли конец этого файла, и при достижении конца файла записывать в эту переменную барьерное значение. Второй способ предпочтительнее, так как он не требует дополнительных действий, связанных с добавлением в файл нового элемента.

File58–61. Воспользуйтесь *вспомогательным файлом* (как в последнем варианте решения File25). При поиске *последнего* пробела в исходном файле (в File59 и File61) удобно просматривать элементы файла *в обратном порядке*, используя функцию *seekg* с вторым параметром *ios::end*.

File62. Благодаря возможности прямого доступа к элементам двоичных файлов, для их сортировки можно использовать любой из алгоритмов, описанных в Array112–114 (модифицировав алгоритм с учетом того, что обрабатываются элементы не массива, а файла). Поскольку перемена местами двух элементов файла требует нескольких файловых операций, в данном случае целесообразно воспользоваться алгоритмом сортировки *простым выбором* (см. Array113), в котором количество перестановок элементов не превосходит $N - 1$, где N — количество элементов. Впрочем, все простейшие алгоритмы сортировки, описанные в Array112–114, весьма неэффективны, особенно для файлов большого размера. Для сортировки файловых данных (в предположении, что размер файла может оказаться настолько велик, что не позволит разместить все данные в оперативной памяти и тем самым свести задачу к сортировке *массива*) разработаны специальные методы. Эти методы *файловой*, или *дисковой*, *сортировки* описаны, например, в [5–6].

File63. Решение этого задания приводится в п. 3.2.4.

File64–65. Простейший способ решения состоит в *двукратном* просмотре исходного файла: при первом просмотре определяется минимальная (макси-

мальная) длина файловой строки (см. решение Minmax1 в [1], п. 4.2.2), при втором все строки данной длины записываются в результирующий файл. Можно, однако, реализовать и *однопроходный* алгоритм, совмещая нахождение строк минимальной (максимальной) длины с их записью в результирующий файл; в этом случае при обнаружении строки, длина которой меньше (или, соответственно, больше) длины ранее прочитанных строк, следует *очищать* результирующий файл, повторно открывая его со вторым параметром *ios::binary | ios:trunc*, и начинать его заполнение заново. Для File65 см. также указание к File10.

File66. См. указание к File62.

File67–70. Для выделения из элементов-строк требуемых подстрок используйте функцию *string::substr* (или операцию индексирования — см. указание к String7), для преобразования этих подстрок в целые числа используйте строковый поток ввода (см. решение String19). В File69–70 можно обойтись без строковых потоков, непосредственно анализируя *символы*, определяющие месяц (для File69 достаточно проверить, что *второй* из этих символов лежит в диапазоне «6»–«8», для File70 придется использовать более сложное условие отбора). Для File68 и File70 см. также указание к File10.

File71–72. Особенности алгоритмов нахождения *условного* минимума/максимума описаны в указании к Minmax12–15 (см. [1], п. 4.3.3). По поводу отбора дат, соответствующих определенному времени года, см. указание к File67–70.

File73. См. указания к File62 и File71–72.

3.4. Тест рубежного контроля

Тест содержит 6 заданий, на выполнение которых отводится 3 минуты. Выберите правильный, по вашему мнению, вариант ответа и отметьте его лю-

бым значком в бланке ответов.

1. Закончите фразу: «Файловый поток *fstream* по умолчанию открывается...			
(1)	для чтения»	(2)	для записи»
(3)	для чтения и записи»	(4)	для дополнения»
2. При организации вывода двоичной информации функциями *read* и *write* требуется указатель буфера данных типа			
(1)	*byte**	(2)	*char**
(3)	*double**	(4)	*long double**

3. Что произойдет при выполнении следующего фрагмента кода при условии, что открываемого файла не существует:
`ifstream is("unexisted");`
`is.get();`			

(1)	Возникнет ошибка компиляции.	(2)	Возникнет ошибка времени выполнения, которая приведет к немедленному завершению работы программы..
(3)	Будет возбуждена исключительная ситуация.	(4)	Установится флаг ошибочного состояния потока.
4. Какой функции у класса *ifstream* нет?			
(1)	*get*	(2)	*read*
(3)	*readsome*	(4)	*readend*
5. Закончите фразу: «Флаг *ios::binary* устанавливается…			
(1)	при создании файла»	(2)	при открытии и создании файла»
(3)	при записи в файл»	(4)	при чтении файла»

6. Как называется функция потока *fstream*, определяющая размер файла?			
(1)	*size*	(2)	*length*
(3)	*count*	(4)	такой функции нет

4. Модуль № 4. Текстовые файлы

4.1. Комплексная цель

Изучить средства стандартной библиотеки C++, связанные с обработкой текстовых файлов, в том числе с их созданием, анализом содержимого и преобразованием, в частности, форматированием.

4.2. Содержание

4.2.1. Создание текстового файла: Text1

Текстовые файлы содержат текстовые строки, разделенные специальными *маркерами конца строки* EOLN (к примеру, в операционной системе Windows маркер EOLN представляет собой два подряд идущих символа с кодами 13 и 10). Такие файлы можно просматривать и редактировать в обычных текстовых редакторах, например, в редакторе системы Visual Studio.

Главной особенностью текстовых файлов (по сравнению с двоичными) является *переменная длина их элементов* — текстовых строк. Такой формат хранения строковых данных является наиболее экономичным, однако он делает невозможным *прямой доступ* к файловой строке по ее номеру. Эти и другие особенности текстовых файлов приводят к тому, что способы их обработки отличаются от способов обработки двоичных файлов.

При выполнении учебных заданий с использованием электронного задачника Programming Taskbook следует учитывать отличия в *отображении* содержимого текстовых файлов. Для вывода строковых данных, составляющих со-

держание текстового файла, отводится не одна, а несколько (от двух до пяти) подряд идущих экранных строк, на каждой из которых размещается *одна строка* текстового файла. Указатель текущей позиции в данном случае содержит номер *первой* отображаемой на экране файловой строки (нумерация ведется от единицы) и размещается в начале первой из экранных строк, отведенных для отображения текстового файла. Прокрутка строк текстового файла обеспечивается теми же клавишами, что и прокрутка элементов двоичного файла.

Рассмотрим задание Text1, посвященное созданию текстового файла.

Text1. Дано имя файла и целые положительные числа *N* и *K*. Создать текстовый файл с указанным именем и записать в него *N* строк, каждая из которых состоит из *K* символов «*» (звездочка).

Вначале приведем вариант решения, в котором файл заполняется *построчно*:

```
task("Text1");
string name;
int n, k;
pt >> name >> n >> k;
ofstream f(name.c_str());
string s(k, '*');
for (int i = 0; i < n; ++i)
    f << s << endl;
f.close();
```

В данном решении вначале формируется вспомогательная строка из *k* символов «*» (для этого используется соответствующий конструктор класса *string*), а затем созданная строка *n* раз записывается в текстовый файл. Обратите внимание на то, что для записи в текстовый файл используется операция <<, ранее применявшаяся нами для вывода результатов в специальный поток *pt*, связанный с задачником. После пересылки каждой строки в текстовый файл в него

необходимо добавить *маркер конца строки*; для этого можно использовать особый объект-манипулятор *endl*, передав его в поток вывода.

При открытии текстового потока *f* указан всего один параметр — имя соответствующего файла. Необходимости в уточнении способа доступа к файлу нет, так как выбранный вариант потока — *ofstream* — означает, что файл надо открыть для вывода, а отсутствие атрибута *ios*::*binary* означает, что файл надо рассматривать как текстовый.

☑ Имеются два режима открытия текстового файлового потока *ofstream*: для *перезаписи* (прежнее содержимое файла пропадает) и для *дополнения* (прежнее содержимое файла дополняется новыми строками). Второй параметр в конструкторе текстового потока (или функции *open*) требуется указывать только для режима дополнения; этот параметр имеет вид *ios*::*app*. Если файл не существует, то он автоматически создается (в этом случае результат не зависит от выбранного режима открытия текстового файла).

При формировании текстового файла можно обойтись без использования строк, записывая данные в файл *посимвольно*. Приведем соответствующий вариант решения задания Text1.

```
task("Text1");
string name;
int n, k;
pt >> name >> n >> k;
ofstream f(name.c_str());
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < k; ++j)
        f << '*';
    f << endl;
}
f.close();
```

В данном варианте решения операция << используется для передачи в файловый поток *отдельного символа*.

4.2.2. Анализ текстового файла: Text4

Text4. Дан текстовый файл. Вывести количество содержащихся в нем символов и строк (маркеры концов строк EOLN и конца файла EOF при подсчете количества символов не учитывать).

В первом варианте решения будем считывать файловые данные *построчно*:

```
task("Text4");
string name;
pt >> name;
ifstream f(name.c_str());
int nc = 0, ns = 0;
while (f.peek() != -1)
{
    string s;
    getline(f, s);
    nc += s.length();
    ++ns;
}
f.close();
pt << nc << ns;
```

Обратите внимание на то, что в условии цикла *while* используется та же функция *peek*, которую мы применяли и при обработке двоичных файлов.

Поскольку в данном задании текстовый файл существует, и его надо открыть на чтение, в программе используется файловый поток типа *ifstream*; для его создания достаточно указать единственный параметр — имя существующего файла.

Для считывания из текстового файла отдельной строки можно использовать несколько способов. В программе применяется функция *getline*, описанная в стандартном заголовочном файле *string*. Ее особенностью является то, что в качестве строкового параметра используется объект типа *string*. Данная функция может иметь третий, дополнительный параметр, в котором указывается символ, являющийся признаком завершения строки (если третий параметр отсутствует, то этим символом считается маркер конца строки). Признак завершения строки не добавляется в результирующую строку, однако считывается из файла.

☑ Имеется функция *ifstream::getline*, отличающаяся от функции *getline* тем, что в качестве строкового параметра в ней надо указывать строковый массив типа *char**, а также его размер (как и в функции *getline*, можно дополнительно указывать символ завершения строки).

Обратите также внимание на вызов функции *length*, выполняемый для строки, которая возвращается функцией *getline*.

Задание Text4 можно решить и с помощью *посимвольного* считывания файловых данных:

```
task("Text4");
string name;
pt >> name;
ifstream f(name.c_str());
int nc = 0, ns = 0;
while (f.peek() != -1)
{
    char c = f.get();
    if (c == '\n')
        ++ns;
    else
        ++nc;
```

```
}
f.close();
pt << nc << ns;
```

Здесь для считывания из текстового файла очередного символа используется функция *get* класса *ifstream*, возвращающая очередной прочитанный символ (или –1, если достигнут конец файла; для обеспечения такой возможности в качестве возвращаемого значения данной функции используется тип *int*).

☑ В классе *ifstream* реализовано несколько перегруженных вариантов функции *get*, из которых мы использовали наиболее простой.

Следует отметить, что маркер конца строки в операционной системе Windows состоит из *двух* управляющих символов: '\r' (carriage return — *возврат каретки*, код 13) и '\n' (line feed — *переход на новую строку*, код 10). Тем не менее, при использовании функции *get* первая часть маркера конца строки пропускается, и в качестве очередного символа возвращается символ '\n', что позволяет реализовать очень простую проверку прочитанного символа (с помощью единственного условного оператора).

Второй вариант решения будет правильно анализировать текстовый файл только в том случае, если каждая строка этого файла оканчивается маркером конца строки. Для текстовых файлов, которые генерируются электронным задачником Programming Taskbook, это условие всегда выполняется, поэтому данный вариант решения будет признан правильным. Однако для произвольного текстового файла возможна ситуация, когда за *последней* строкой файла следует не маркер конца строки, а *сразу* маркер конца файла (это произойдет, если после записи последней строки в файл не будет добавлен маркер конца строки). В этом случае при использовании второго из рассмотренных алгоритмов найденное количество строк будет на 1 меньше фактического, так как строки в этом алгоритме подсчитываются по маркерам конца строки, а последняя строка такого маркера не имеет. Первый вариант решения лишен подобного недостат-

ка, поскольку функция *getline* правильно считывает последнюю файловую строку и в том случае, когда эта строка оканчивается не маркером конца строки, а сразу маркером конца файла.

☑ При анализе предложенных вариантов решения может возникнуть вопрос, почему для чтения данных из текстового файла не используется операция >> (применяемая, в частности, в наших программах для получения данных от задачника Programming Taskbook с использованием потока *pt*). Эта операция реализована в файловых потоках, однако ее неудобно применять в ситуациях, подобных рассмотренным выше, в силу одной ее особенности: при попытке считывания строки из файла с помощью операции >> признаком завершения считываемой строки считается не только маркер конца строки или файла, но и *пробел* (а также *символ табуляции*). При считывании с помощью операции >> символов автоматически пропускаются все «незначащие» символы: пробелы, символы табуляции, маркеры конца строки. Таким образом, если, к примеру, в последнем варианте решения вместо функции *get* (*char c = f.get*();) использовать операцию >> (*char c*; *f >> c*;), то после завершения цикла в переменной *ns* по-прежнему содержалось бы значение 0, а значение переменной *nc* соответствовало бы количеству символов в файле, *отличных от пробела*.

4.2.3. Преобразование текстового файла: Text21

Text21. Дан текстовый файл, содержащий более трех строк. Удалить из него последние три строки.

Поскольку исходный текстовый файл нельзя одновременно открыть и на чтение, и на запись, любое его преобразование необходимо выполнять с помощью *вспомогательного текстового файла* (ср. с последним вариантом решения задания File25 в п. 3.2.2). Для определения количества строк, содержащихся в исходном файле, проще всего выполнить их считывание. Учитывая эти замечания, получаем первый вариант решения:

```
task("Text21");
string name1, name2 = "$T21$.tmp", s;
pt >> name1;
```

```
ifstream f1(name1.c_str());
ofstream f2(name2.c_str());
int n = 0;
while (f1.peek() != -1)
{
    getline(f1, s);
    n++;
}
f1.clear();
f1.seekg(0);
for (int i = 0; i < n - 3; ++i)
{
    getline(f1, s);
    f2 << s << endl;
}
f1.close();
f2.close();
remove(name1.c_str());
rename(name2.c_str(), name1.c_str());
```

Обратите внимание на вызов функции *f1.clear*(), возвращающий поток ввода-вывода из состояния «достигнут конец файла» или из состояния ошибки в стандартное состояние: если не вызвать данную функцию, то все последующие попытки чтения из потока будут оканчиваться неудачей (и в результате полученный файл будет содержать лишь пустые строки).

Приведенный выше вариант решения является неэффективным, поскольку требует *двух* просмотров исходного файла *f1*: первый — для определения его размера, который записывается в переменную *n*, второй — для создания вспомогательного файла *f2*, содержащего все строки исходного файла, кроме трех последних.

Задание Text21 можно выполнить и за один просмотр исходного файла, если воспользоваться следующим наблюдением: строка должна быть записана во вспомогательный файл, *если после нее в исходном файле находятся по крайней мере три строки*. Таким образом, записывать очередную строку во вспомогательный файл следует только после считывания из исходного файла трех следующих за ней строк. Благодаря такому упреждающему считыванию необходимость в предварительном определении размера исходного файла отпадает. Для хранения строк, которые уже считаны из исходного файла, но еще не записаны во вспомогательный файл, удобно использовать массив из трех элементов типа *string*. Приведем этот вариант решения.

```
task("Text21");
string name1, name2 = "$T21$.tmp", s[3];
pt >> name1;
ifstream f1(name1.c_str());
ofstream f2(name2.c_str());
for (int i = 0; i < 3; ++i)
    getline(f1, s[i]);
int n = 0;
while (f1.peek() != -1)
{
    f2 << s[n] << endl;
    getline(f1, s[n]);
    n = (n + 1) % 3;
}
f1.close();
f2.close();
remove(name1.c_str());
rename(name2.c_str(), name1.c_str());
```

Вначале элементы вспомогательного массива *s* инициализируются первыми тремя строками из исходного файла (по условию файл содержит не менее трех строк), а переменная *n*, в которой будет храниться номер текущего элемента массива *s*, инициализируется нулевым значением.

На каждой итерации цикла *while* во вспомогательный файл записывается строка *s*[*n*], то есть текущий элемент массива *s* (в этот момент уже известно, что после данной строки в исходном файле содержатся, по крайней мере, три строки: две из этих строк уже прочитаны в массив, а на третьей строке располагается файловый указатель). Затем из исходного файла считывается очередная строка, которая записывается на место только что сохраненного во вспомогательном файле элемента массива *s*. Наконец, выполняется циклическое изменение номера *n*: 0 переходит в 1, 1 — в 2, 2 — в 0.

После завершения цикла *while* во вспомогательный файл будут записаны все строки исходного файла, кроме последних трех, а эти три последние строки будут содержаться в массиве *s*.

Итак, использовав вспомогательный массив из трех строк, мы получили эффективный *однопроходный* алгоритм решения задания Text21. Аналогичным способом можно решить задания Text22 и Text23, в которых требуется удалить или скопировать *k* последних строк текстового файла.

4.3. Проектное задание

4.3.1. Варианты

Выполните задачи группы Text, указанные в вашем варианте проектного задания (формулировки задач приводятся в [4]). Если вы не получили вариант проектного задания, то выполните задачи из какого-либо варианта.

ВАРИАНТ 1 (1) Основные операции: 2, 13, 17 (2) Анализ текста: 25, 31 (3) Форматирование: 36, 37 (4) Числовые данные: 43, 47 (5) Разные задачи: 56, 59	**ВАРИАНТ 2** (1) Основные операции: 2, 12, 23 (2) Анализ текста: 25, 33 (3) Форматирование: 35, 38 (4) Числовые данные: 42, 44 (5) Разные задачи: 54, 60
ВАРИАНТ 3 (1) Основные операции: 2, 8, 22 (2) Анализ текста: 24, 33 (3) Форматирование: 35, 39 (4) Числовые данные: 42, 46 (5) Разные задачи: 54, 58	**ВАРИАНТ 4** (1) Основные операции: 3, 11, 19 (2) Анализ текста: 26, 31 (3) Форматирование: 36, 39 (4) Числовые данные: 41, 49 (5) Разные задачи: 55, 58
ВАРИАНТ 5 (1) Основные операции: 3, 6, 18 (2) Анализ текста: 27, 31 (3) Форматирование: 36, 38 (4) Числовые данные: 40, 48 (5) Разные задачи: 56, 58	**ВАРИАНТ 6** (1) Основные операции: 3, 15, 22 (2) Анализ текста: 28, 30 (3) Форматирование: 34, 39 (4) Числовые данные: 41, 50 (5) Разные задачи: 53, 60
ВАРИАНТ 7 (1) Основные операции: 2, 5, 17 (2) Анализ текста: 26, 29 (3) Форматирование: 36, 37 (4) Числовые данные: 40, 51 (5) Разные задачи: 54, 57	**ВАРИАНТ 8** (1) Основные операции: 2, 7, 23 (2) Анализ текста: 27, 30 (3) Форматирование: 34, 37 (4) Числовые данные: 40, 52 (5) Разные задачи: 56, 59

ВАРИАНТ 9 (1) Основные операции: 2, 16, 19 (2) Анализ текста: 27, 31 (3) Форматирование: 35, 38 (4) Числовые данные: 43, 46 (5) Разные задачи: 53, 57	**ВАРИАНТ 10** (1) Основные операции: 3, 14, 20 (2) Анализ текста: 24, 32 (3) Форматирование: 35, 38 (4) Числовые данные: 43, 45 (5) Разные задачи: 55, 60
ВАРИАНТ 11 (1) Основные операции: 3, 10, 20 (2) Анализ текста: 28, 29 (3) Форматирование: 34, 37 (4) Числовые данные: 41, 45 (5) Разные задачи: 53, 59	**ВАРИАНТ 12** (1) Основные операции: 3, 9, 18 (2) Анализ текста: 27, 32 (3) Форматирование: 34, 39 (4) Числовые данные: 42, 47 (5) Разные задачи: 55, 57
ВАРИАНТ 13 (1) Основные операции: 2, 8, 18 (2) Анализ текста: 27, 29 (3) Форматирование: 34, 37 (4) Числовые данные: 40, 48 (5) Разные задачи: 54, 60	**ВАРИАНТ 14** (1) Основные операции: 3, 11, 20 (2) Анализ текста: 27, 32 (3) Форматирование: 35, 39 (4) Числовые данные: 41, 47 (5) Разные задачи: 56, 58
ВАРИАНТ 15 (1) Основные операции: 3, 13, 20 (2) Анализ текста: 27, 30 (3) Форматирование: 34, 37 (4) Числовые данные: 41, 46 (5) Разные задачи: 53, 57	**ВАРИАНТ 16** (1) Основные операции: 3, 15, 22 (2) Анализ текста: 26, 31 (3) Форматирование: 36, 38 (4) Числовые данные: 43, 46 (5) Разные задачи: 53, 59

ВАРИАНТ 17 (1) Основные операции: 2, 16, 18 (2) Анализ текста: 27, 32 (3) Форматирование: 35, 38 (4) Числовые данные: 43, 47 (5) Разные задачи: 55, 59	**ВАРИАНТ 18** (1) Основные операции: 2, 9, 17 (2) Анализ текста: 26, 30 (3) Форматирование: 34, 38 (4) Числовые данные: 42, 44 (5) Разные задачи: 54, 59
ВАРИАНТ 19 (1) Основные операции: 3, 10, 22 (2) Анализ текста: 25, 29 (3) Форматирование: 34, 39 (4) Числовые данные: 40, 50 (5) Разные задачи: 55, 58	**ВАРИАНТ 20** (1) Основные операции: 2, 12, 23 (2) Анализ текста: 24, 33 (3) Форматирование: 36, 39 (4) Числовые данные: 41, 51 (5) Разные задачи: 56, 57
ВАРИАНТ 21 (1) Основные операции: 2, 7, 19 (2) Анализ текста: 25, 33 (3) Форматирование: 35, 37 (4) Числовые данные: 42, 45 (5) Разные задачи: 53, 60	**ВАРИАНТ 22** (1) Основные операции: 3, 14, 19 (2) Анализ текста: 28, 31 (3) Форматирование: 36, 38 (4) Числовые данные: 43, 49 (5) Разные задачи: 54, 60
ВАРИАНТ 23 (1) Основные операции: 2, 5, 17 (2) Анализ текста: 24, 31 (3) Форматирование: 35, 37 (4) Числовые данные: 40, 52 (5) Разные задачи: 55, 57	**ВАРИАНТ 24** (1) Основные операции: 3, 6, 23 (2) Анализ текста: 28, 31 (3) Форматирование: 36, 39 (4) Числовые данные: 42, 45 (5) Разные задачи: 56, 58

4.3.2. Указания

Text1–3. Решение Text1 приводится в п. 4.2.1. Остальные задания решаются аналогично; для них, как и для Text1, можно реализовать два варианта решения: с построчным и посимвольным заполнением результирующего файла.

Text4. Решение этого задания приводится в п. 4.2.2.

Text5–20. Для добавления новых данных *в конец текстового файла* (Text5–6) достаточно открыть текстовый поток *ofstream* в режиме *дополнения*, указав в его конструкторе в качестве второго параметра значение *ios::app*. Все прочие действия по преобразованию строк файла, а именно, вставка строк в начало (Text7–8) или середину файла (Text9–11), замена строк (Text12), удаление строк (Text13–16), преобразование строк (Text17–20), выполняются с помощью *вспомогательного* текстового файла (ср. с последним вариантом решения File25). В Text14 для того чтобы проверить, является ли только что прочитанная (функцией *getline*) строка последней строкой файла, используйте функцию *peek*. В Text19–20 удобно использовать *посимвольное* чтение и запись файловых данных (как во вторых вариантах решений Text1 и Text4). По поводу преобразования прописных русских букв в строчные и обратно (в Text19) см. указание к String16–18.

Text21–23. Решение Text21 приводится в п. 4.2.3. Остальные задания решаются аналогично.

Text24–25. Непустая строка является первой строкой абзаца, если ей предшествует пустая строка (или если она является первой строкой файла). Для хранения предшествующей строки используйте вспомогательную переменную. В Text25 используйте вспомогательный текстовый файл.

Text26–28. Для определения начала абзаца достаточно выделить из текущей строки первые пять символов (функцией *string::substr*) и сравнить их со

строкой из пяти пробелов. В Text27–28 используйте вспомогательный текстовый файл.

Text29–30. Для выделения слов удобно использовать посимвольное считывание данных из исходного файла (как во втором варианте решения Text4), при котором анализируется текущий и предшествующий ему символ (ср. с последним вариантом решения String41). Предшествующий символ должен храниться во вспомогательной переменной. Для того чтобы избежать специальной обработки начального и конечного слова в строке, удобно считать разделителем не только пробел, но и символы с кодами 13 и 10 (составляющие маркер конца строки EOLN); проверять, является ли символ подобным разделителем, проще всего с помощью функции *isspace*. Алгоритм нахождения минимумов/максимумов приведен в решении Minmax1 (см. [1], п. 4.2.2); для определения длины слов используйте функцию *string::length*.

Text31–33. По поводу выделения слов из файла см. указание к Text29–30. В данном случае разделителями считаются не только пробельные символы, но и все *знаки препинания*, поэтому для проверки того, что символ является разделителем, надо использовать не только функцию *isspace*, но и функцию *ispunct*. По поводу преобразования прописных русских букв в строчные и строчных в прописные см. указание к String16–18.

Text34–37. Используйте вспомогательный текстовый файл. Для определения длины строки используйте функцию *string::length*. В Text34–35 для вставки в начало строки пробелов используйте функцию *string::insert* или операцию сцепления, в Text36 для удаления половины начальных пробелов используйте функцию *string::erase*, в Text37 для вставки пробелов в середину строки используйте функцию *string::insert*.

Text38–39. Можно создать вспомогательный файл, в котором каждый абзац из исходного файла будет располагаться в *одной* строке, после чего организо-

вать *посимвольное* считывание данных из вспомогательного файла (выделяя из него слова и определяя, можно ли дописать очередное слово в конец текущей строки результирующего файла или необходимо перейти на новую строку). Более эффективным является решение, не использующее вспомогательный файл.

Text40–41. При формировании текстовых файлов с числовой информацией удобно организовать непосредственную запись данных из числовых переменных в результирующий файл, используя *манипуляторы потоков*. Например, в Text40 запись в текстовый поток *os* типа *ofstream* отформатированной строки, содержащей целые числа *a* и *b* и снабженной начальным и конечным разделителем «|», будет осуществляться оператором

```
os << "|" << setw(30) << a
    << setw(30) << b << "|" << endl;
```

Манипулятор потока *setw* задает ширину поля вывода для следующего за ним *одного* элемента. Для выравнивания данных *по левому краю* (в Text41) достаточно один раз вывести в поток манипулятор *left*. Манипуляторы потоков с параметрами объявлены в заголовочном файле *iomanip*.

Text42–43. См. указание к Text40–41. Для задания длины дробной части выводимых чисел используйте манипулятор *setprecision*, передавая ему количество цифр после точки. Чтобы гарантировать вывод с фиксированной точкой, достаточно однократно вывести в поток манипулятор *fixed*.

Text44–45. Считывайте содержимое файла оператором >> непосредственно в целые или вещественные переменные. Все пробельные символы и переводы строки будут автоматически игнорироваться. Для проверки того, что вещественное число *x* имеет ненулевую дробную часть, достаточно сравнить это число с его округленным значением *ceil(x)*.

Text46–48. См. указание к Text44–45.

Text49. Используйте вспомогательный текстовый файл; см. также указание к Text40–41. В этом задании при записи числовых данных в текстовый файл не следует указывать ширину поля вывода.

Text50. Организуйте построчное считывание данных их исходного файла. Используя функцию *getline* с третьим параметром — максимальной длиной считываемой строки, считывайте первую часть строки. Остаток строки считывайте в вещественную переменную. Начальная часть (длиной 30 символов) записывается в строковый файл (по поводу записи данных в строковый файл см. п. 3.2.4). Прочитанное число записывается в файл вещественных чисел.

Text51–52. Ср. с Text46. В Text52 пропускайте символы-разделители (при их чтении в целые переменные поток переходит в ошибочное состояние, которое можно проверять функцией потока *fail* и сбрасывать функцией потока *clear*).

Text53. Организуйте посимвольное считывание данных из исходного файла (как во втором варианте решения задания Text4). Для того чтобы проверить, является ли символ знаком препинания, проще всего воспользоваться функцией *ispunct,* объявленной в заголовочном файле *cctype*.

Text54–56. Организуйте посимвольное считывание данных из исходного файла (как во втором варианте решения задания Text4). Для того чтобы при формировании результирующего файла избежать дублирования символов, удобно воспользоваться вспомогательным логическим массивом *a* из 224 элементов, в котором каждый элемент соответствует одному символу из кодовой таблицы: *a*[*0*] — пробелу, *a*[*1*] — «!», …, *a*[*223*] — «я». Таким образом, каждому символу *c* будет соответствовать одно логическое значение из массива *a* с индексом *static_cast<int>(c) – static_cast<int>(' ')*. В случае, когда символы требуется расположить в порядке их *первого появления* в тексте (Text54), для каждого прочитанного из текстового файла символа

надо проверять, установлено ли значение *true* для данного символа, и если нет, то записывать его в результирующий файл, а также устанавливать значение *true* для соответствующего элемента в массиве *a*. В случае, когда символы требуется расположить *по возрастанию/убыванию их кодов* (Text55–56), надо вначале *сформировать* массив *a*, установив значение *true* для всех символов, встретившихся в исходном файле, а затем, перебрав все элементы массива *a* по возрастанию (Text55) или по убыванию (Text56) индексов, записать в файл те из них, для которых установлено значение *true*.

Text57–58. Организуйте посимвольное считывание данных из исходного файла (как во втором варианте решения задания Text4). Для хранения информации о количестве появлений русских букв удобно воспользоваться вспомогательным целочисленным массивом *a* из 32 элементов, в котором каждый элемент соответствует одной из строчных русских букв: *a*[*0*] — букве «а», *a*[*1*] — «б», …, *a*[*31*] — «я» (буква «ё», как и в прочих заданиях, не учитывается). Таким образом, строчной русской букве *c* будет соответствовать элемент массива *a* с индексом *static_cast<int>(c) – static_cast<int>('a')*. Массив *a* заполняется при считывании символов исходного файла таким образом, чтобы после завершения считывания данных в каждом элементе массива содержалось *число появлений* в файле буквы, соответствующей этому элементу (например, в элементе *a*[*0*] должно содержаться число появлений в файле буквы «а»). Теперь, если строки результирующего файла требуется упорядочить *по возрастанию кодов букв* (Text57), достаточно последовательно просмотреть все элементы массива *a*, записывая в результирующий файл строки, соответствующие *ненулевым* элементам этого массива. Если же требуется упорядочить строки *по убыванию числа появлений букв* (Text58), то необходимо просматривать массив несколько раз. При каждом просмотре надо найти элемент с наибольшим значением, записать

информацию о нем в результирующий файл и после этого *обнулить* данный элемент (просмотр элементов массива осуществляется до тех пор, пока в нем остаются ненулевые элементы).

Text59. Ср. с String63. Организуйте посимвольное считывание данных из исходного файла (как во втором варианте решения задания Text4). При шифровании символов используйте выражения, аналогичные приведенным в указании к String3–5. Для нахождения величины смещения, задаваемой символом строки *s* с индексом *k*, достаточно использовать выражение *static_cast<int>(s*[*k*]) – *static_cast<int>('0')*.

Text60. Задание можно решить аналогично Text59, если предварительно найти ключ *S* (для этого достаточно сравнить символы исходной строки с символами первой строки исходного файла).

4.4. Тест рубежного контроля

Тест содержит 6 заданий, на выполнение которых отводится 3 минуты. Выберите правильный, по вашему мнению, вариант ответа и отметьте его любым значком в бланке ответов.

1. Дан текстовый файл, содержащий 10 строк. Каждая строка состоит из 20 символов '*' (звездочка) и завершается маркером конца строки. Каков размер данного файла в байтах?			
(1)	200	(2)	210
(3)	220	(4)	420
2. Укажите верное утверждение.			
(1)	Текстовый файл можно открыть на дополнение.	(2)	Текстовый файл можно открыть одновременно на чтение и запись.
(3)	Для текстовых файлов возмо-	(4)	После каждой строки в тексто-

	жен прямой доступ к их элементам-строкам.		вом файле присутствует маркер конца строки.
3. Закончите фразу: «Манипулятор *setw*…			
(1)	действует на все данные, выводимые после него»	(2)	действует на один элемент, выводимый после него»
(3)	действует на все данные, вплоть до вывода в поток манипулятора *unsetw*»	(4)	в стандартной библиотеке C++ отсутствует»
4. Закончите фразу: «Для чтения в строку типа *string* текстовых данных из потока вплоть до маркера конца строки используется…			
(1)	функция-член *getline* класса *istream*»	(2)	внешняя функция *getline*»
(3)	функция-член *readstring* класса *istream*»	(4)	внешняя функция *readstring*»
5. Данные каких типов можно читать из текстового потока?			
(1)	только символьных и строковых типов	(2)	любых типов
(3)	данные тех типов, для которых определена операция >>	(4)	только символьного типа
6. Закончите фразу: «Манипулятор *precision*…			
(1)	используется для указания ширины поля вывода»	(2)	используется для указания ширины поля для вывода дробной части»
(3)	точности чтения вещественных данных»	(4)	в стандартной библиотеке C++ отсутствует»

5. Приложение А. Язык C++: работа с массивами, строками и файлами

5.1. Описание массива и его инициализация. Параметры-массивы

В C++, как и в C, имеются только одномерные массивы, для доступа к элементам которых используется единственный целочисленный индекс. Многомерные, в частности, двумерные, массивы представляют собой массивы массивов; по этой причине и при описании, и при обращении к элементам подобных массивов каждый индекс должен заключаться в отдельные квадратные скобки.

Индексы массивов в C++ всегда имеют *фиксированную нижнюю границу*, равную 0.

В последующих примерах предполагается, что массив *a* является одномерным массивом вещественных чисел и имеет размер 5, а массив *b* является двумерным массивом-матрицей целых чисел размера 3 на 4.

При описании массива всегда указывается тип его элементов и, как правило, количество его элементов (в квадратных скобках):

```
double a[5];
int b[3][4];
```

Если массив описан как глобальная переменная, то его элементы по умолчанию инициализируются нулевыми значениями. Если массив описан внутри функции (и не имеет атрибута *static*), то начальные значения его элементов не определены.

Элементы любого массива можно инициализировать явно в момент его описания; для этого достаточно указать значения элементов в фигурных скобках, перечислив их через запятую:

```
double a[5] = {0, 1, 2, 3, 4};
```

Если в списке указаны значения не всех элементов, то оставшиеся элементы получают нулевые значения; в частности, следующий оператор обеспечит инициализацию всех элементов массива *a* нулями даже если этот массив является локальной переменной:

```
double a[5] = {};
```

С другой стороны, при указании списка значений элементов массива можно не указывать его размер (который определяется по числу элементов), например:

```
double a[] = {0, 1, 2, 3, 4};
```

Аналогичным образом можно инициализировать и многомерные массивы; следует лишь учитывать *стандартный порядок перебора* элементов многомерных массивов, который определяется следующим образом: *при переборе элементов быстрее изменяются их правые индексы* (в частности, двумерные массивы-матрицы перебираются *по строкам*). Таким образом, в приведенном ниже примере элементы первой строки массива-матрицы *b* получат последовательные значения 0, 1, 2, 3 и, кроме того, первый элемент второй строки (то есть элемент *b*[*1*][*0*]) получит значение 4 (прочие элементы массива *b* будут инициализированы нулевыми значениями):

```
int b[3][4] = {0, 1, 2, 3, 4};
```

Для инициализации всех элементов многомерного массива нулевыми значениями достаточно указать пустой список значений:

```
int b[3][4] = {};
```

Кроме того, можно использовать *вложенные* списки значений. В следующем примере начальные элементы каждой строки матрицы *b* инициализируются значением 1, а все остальные элементы (по умолчанию) — значением 0:

```
int b[3][4] = {{1}, {1}, {1}};
```

При подобном способе инициализации допустимо не указывать размер массива по первому (левому) индексу; однако все прочие размеры указывать обязательно. Так, приводимое ниже описание будет равносильно предыдущему:

```
int b[][4] = {{1}, {1}, {1}};
```

С помощью операции *sizeof*, примененной к переменной-массиву, можно определить размер памяти (в байтах), выделенной для данного массива. Поделив этот размер на размер отдельного элемента, можно определить число элементов в массиве. Так, выражение *sizeof(a)/sizeof(double)* вернет 5, а *sizeof(b)/sizeof(int)* вернет 12. Для двумерных массивов-матриц аналогичным образом можно определить число их столбцов (то есть длину каждой строки). Например, число столбцов матрицы *b* (равное 4) можно узнать с помощью выражения *sizeof(b*[*0*]*)/sizeof(int)* (здесь мы использовали тот факт, что элемент *b*[*0*] сам является одномерным массивом).

Следует заметить, что при описании одномерного массива в качестве параметра функции можно не указывать его размер, ограничившись указанием типа элементов и квадратными скобками, например:

```
void f(double a[])
{…}
```

Это связано с тем, что при передаче массива в качестве параметра фактически передается только *указатель на его начало* (см. следующий пункт). Копирования самих элементов в формальный параметр-массив не производится, поэтому для успешной компиляции функции знать размер формального параметра-массива не требуется. Данный факт означает также, что массивы всегда

передаются в функцию *по ссылке*, то есть все изменения элементов производятся не для копии массива, а для того же самого массива, который передан в функцию как фактический параметр.

Если в функцию требуется передать многомерный массив, то в этом случае допустимо не указывать размер только по первому (левому) индексу, так как знание размеров по другим индексам необходимо для правильного определения смещения элементов относительно начала массива:

```
void f(double b[][4])
{…}
```

При работе с массивами в C++ следует учитывать, что выход за допустимый диапазон индексов для массивов *не контролируется*. При подобном выходе программа просто обращается к участку памяти, не связанному с массивом, и может либо прочесть, либо попытаться изменить его содержимое. Разумеется, в результате программа будет работать неправильно, однако ошибка может проявиться не сразу, что серьезно затруднит поиск ее причины.

5.2. Массивы и указатели

Переменная-массив в C++ представляет собой типизированный указатель на начало участка памяти, выделенного для данного массива. Если, например, массив *a* описан как *double a*[*5*], то переменная *a* является указателем на вещественное число, а после применения к ней *операции разыменования* * мы получим значение первого элемента массива (с индексом 0). Таким образом, выражение **a* эквивалентно *a*[*0*]. Более того, поскольку в C++ определено сложение указателя и целого числа (означающее перемещение в памяти на указанное количество элементов данного типа), для выражения *a*[*i*] тоже можно указать эквивалентное ему выражение в терминах разыменованного указателя: **(a + i)*.

Отмеченная эквивалентность массивов и указателей проявляется и в другую сторону: для создания массива с элементами требуемого типа достаточно описать указатель на этот тип и выделить для него участок памяти требуемого

размера с помощью операции *new*[] — модификации «обычной» операции выделения памяти *new*. Например, для описания массива *a* вещественных чисел размера 5 можно использовать следующий оператор:

```
double* a = new double[5];
```

После подобного описания к элементам массива a можно обращаться, используя обычную операцию индексирования, например, *a*[*2*].

Что касается многомерных массивов, то, учитывая, что они являются, по существу, массивами массивов, связанные с ними переменные можно интерпретировать как *указатели на указатели*. В частности, двумерный массив-матрицу целых чисел *b* размера 3 на 4 можно описать следующим образом:

```
int** b = new int*[3];
for (int i = 0; i < 3; ++i)
    b[i] = new int[4];
```

При создании массивов с помощью описания указателей и явного выделения для них памяти операцией *new*[] следует учитывать следующие особенности:

- при подобном способе создания массивов их элементы *нельзя инициализировать* с помощью списка начальных значений;
- операция *sizeof*, примененная к указателям, возвращает размер *одного* элемента данных, на который они указывают, даже если при выделении памяти вместо операции *new* была использована операция *new*[];
- при явном выделении памяти операцией *new*[] требуется в дальнейшем выполнить и *явное ее освобождение* операцией *delete*[] (например, *delete*[] *a*).

При создании массива с помощью операции *new*[] в качестве его размера можно указывать любое выражение целого типа (а не только константное выражение, как при обычном описании массива). Это позволяет создавать масси-

вы, размер которых определяется на предыдущих этапах работы программы, например:

```
int n;
cin >> n;
double* a = new double[n];
```

Отметим, что с помощью указателей и операции *new*[] можно создавать *непрямоугольные* многомерные массивы, например, двумерные массивы для хранения *треугольных матриц*, в которых каждая строка матрицы содержит различное количество элементов. В качестве примера приведем фрагмент, создающий массив *d* для хранения ненулевых элементов $d_{0,0}$, $d_{1,0}$, $d_{1,1}$, $d_{2,0}$, $d_{2,1}$, $d_{2,2}$ целочисленной нижнетреугольной матрицы порядка 3:

```
int** d = new int*[3];
for (int i = 0; i < 3; ++i)
    d[i] = new int[i + 1];
```

5.3. Массивы и строки

Из языка C в C++ перешел особый способ хранения текстовых строк: в виде набора символов, оканчивающегося особым нулевым символом '\0'. При этом данный набор (называемый *C-строкой* — см. п. 5.6) может размещаться либо в обычном массиве типа *char*[], либо в памяти, выделенной с помощью операции *new*[] для указателя типа *char**. При описании C-строки как массива *char*[] ее можно инициализировать *строковым литералом*, то есть текстом, заключенным в двойные кавычки, например:

```
char s1[8] = "Example";
```

Указанное выражение эквивалентно следующему:

```
char s1[8] = {'E', 'x', 'a', 'm', 'p', 'l', 'e', '\0'};
```

При этом необходимо следить, чтобы размер массива был достаточным для хранения не только всех символов строки-литерала, но и завершающего ее нулевого символа.

Если С-строка описана как *char**, то ей также можно присвоить символьный литерал, например,

```
char* s2 = "Example";
```

Однако результат будет другим: произойдет не копирование символов в новую область памяти (как в предыдущем случае), а присваивание указателю *s адреса* того участка памяти, в котором размещен строковый литерал. В дальнейшем С-строку *s2* можно будет использовать *только для чтения*, как и «обычные» строковые литералы. Для «настоящего» копирования С-строк (в том числе строковых литералов) необходимо использовать стандартную функцию *strcpy*; при этом предварительно необходимо предусмотреть достаточно места в строковом массиве-«приемнике», например:

```
char* s2 = new char[8];
strcpy(s2, "Example");
```

После выполнения данных операторов С-строка *s2* будет содержать текст «Example», доступный как для чтения, так и для изменения.

Дополнительные сведения о С-строках приводятся в п. 5.6. Следует заметить, что для обработки строк в С++ рекомендуется применять класс *string* (см. п. 5.7).

5.4. Кодовые таблицы символов

Символы-литералы в языке С++ заключаются в *одинарные* кавычки: 'A', '1' (в отличие от строк-литералов, заключаемых в *двойные* кавычки).

При работе с символами с использованием типа *char* в рассматриваемых реализациях С++ используется 8-битная кодировка, соответствующая кодировке ASCII (коды с 1 по 127) и той кодовой странице, которая устанавливается по умолчанию для операционной системы Windows (коды с 128 по 255). Для русской версии Windows это кодовая страница 1251 «Windows Cyrillic».

Таблица 1 содержит символы кодировки ASCII. Поскольку коды 1–31 генерируются, как правило, при нажатии *управляющих клавиш*, соответствующие им символы в таблице не указаны.

Таблица 2 содержит символы второй половины кодовой таблицы в *ANSI-кодировке* CP1251 «Windows Cyrillic», применяемой в русской версии Windows.

Следует заметить, что, поскольку тип *char* является знаковым, при преобразовании выражений этого типа, содержащих символы из таблицы 2, к типу *int* будут возвращаться *отрицательные* значения, равные коду соответствующего символа минус 256, например, (*int*)'Б' вернет – 63, а (*int*)'я' вернет –1.

Кроме типа *char*, который иногда называют *суженным символьным типом*, язык C++ предоставляет средства и для работы с *расширенным символьным типом wchar_t*, позволяющим обрабатывать более обширные символьные наборы (в рассматриваемых реализациях C++ тип *wchar_t* имеет размер 2 байта и позволяет работать с кодовым набором Unicode). В настоящем пособии работа с расширенными символьными типами не рассматривается.

Таблица 1. Кодировка ASCII

32	! 33	" 34	# 35	$ 36	% 37	& 38	' 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	127

Таблица 2. ANSI-кодировка кириллицы (CP1251)

Ђ 128	Ѓ 129	‚ 130	ѓ 131	„ 132	… 133	† 134	‡ 135	€ 136	‰ 137	Љ 138	‹ 139	Њ 140	Ќ 141	Ћ 142	Џ 143
ђ 144	‘ 145	’ 146	“ 147	” 148	• 149	– 150	— 151	□ 152	™ 153	љ 154	› 155	њ 156	ќ 157	ћ 158	џ 159
160	Ў 161	ў 162	Ј 163	¤ 164	Ґ 165	¦ 166	§ 167	Ё 168	© 169	Є 170	« 171	¬ 172	173	® 174	Ї 175
° 176	± 177	І 178	і 179	ґ 180	µ 181	¶ 182	· 183	ё 184	№ 185	є 186	» 187	ј 188	Ѕ 189	ѕ 190	ї 191
А 192	Б 193	В 194	Г 195	Д 196	Е 197	Ж 198	З 199	И 200	Й 201	К 202	Л 203	М 204	Н 205	О 206	П 207
Р 208	С 209	Т 210	У 211	Ф 212	Х 213	Ц 214	Ч 215	Ш 216	Щ 217	Ъ 218	Ы 219	Ь 220	Э 221	Ю 222	Я 223
а 224	б 225	в 226	г 227	д 228	е 229	ж 230	з 231	и 232	й 233	к 234	л 235	м 236	н 237	о 238	п 239
р 240	с 241	т 242	у 243	ф 244	х 245	ц 246	ч 247	ш 248	щ 249	ъ 250	ы 251	ь 252	э 253	ю 254	я 255

5.5. Функции, связанные с обработкой символов

Функции, связанные с обработкой символов, реализованы в двух заголовочных файлах: *cctype* (перенесен из библиотеки C) и *locale* (входит в состав библиотеки C++, основанной на шаблонах). Средства из файла *locale* ориентированы, прежде всего, на обработку многобайтовых символьных наборов, однако они могут оказаться полезны и при анализе однобайтовых символов, если они не входят в таблицу ASCII (то есть не принадлежат диапазону 1–127). Данное обстоятельство связано с тем, что функции из файла *cctype* корректно обрабатывают только символы из таблицы ASCII.

<cctype>

```
int iscntrl(int c);
int isdigit(int c);
int isalpha(int c);
int islower(int c);
```

```
int isupper(int c);
int isalnum(int c);
int ispunct(int c);
int isspace(int c);
```

Позволяют определить вид символа с кодом *c*; при утвердительном результате возвращают ненулевое значение (не обязательно 1), при отрицательном результате возвращают 0: *iscntrl — управляющий символ* (символы с кодами в диапазонах 0–31 и 127); *isdigit — цифра* (символы '0'–'9'); *isalpha* — латинская буква, *islower — строчная* (маленькая) латинская буква; *isupper — прописная* (заглавная) латинская буква; *isalnum* — латинская буква или цифра, *ispunct — знак пунктуации* (символы из диапазона 0–127, не являющиеся управляющими символами, цифрами, буквами и пробелом); *isspace — пробельный символ* (символы с кодами 9 (знак табуляции), 10 (новая строка), 11–12, 13 (возврат каретки), 32 (пробел)).

Если параметр *c* лежит вне диапазона 0–127 (в частности, если в качестве параметра указана русская буква, то любая из перечисленных функций возвращает 0.

<cctype>

```
int tolower(int c);
int toupper(int c);
```

Преобразует символ с кодом *c* к нижнему или верхнему регистру соответственно и возвращает код преобразованного символа (если символ не является *латинской* буквой, то возвращается код символа *c*).

Если в ходе анализа и преобразования символов требуется обрабатывать русские буквы, то вместо перечисленных выше функций из файла *cctype* следует использовать одноименные функции из файла *locale*. Все эти функции имеют второй параметр, определяющий *локаль*, то есть совокупность языковых настроек, связанных как с самим языком, так и со страной, в которой он исполь-

зуется. Для создания локали *loc*, связанной с российскими настройками, достаточно воспользоваться следующим оператором (в нем описывается и инициализируется объект *loc* типа *locale*, причем при инициализации ему передается строка, определяющая требуемую локаль):

```
locale loc("Russian_Russia");
```

После создания локали *loc* ее можно указывать во всех функциях, перечисленных выше. Например, вызов *isalpha*('Ф', *loc*) вернет ненулевое значение, а вызов *tolower*('Ф', *loc*) вернет символ 'ф' (в ANSI-кодировке).

5.6. Строки как символьные массивы. Строковые литералы

В языке C — предшественнике языка C++ — для хранения и обработки строк используются *символьные массивы*. Поскольку длина строки, хранящейся в массиве символов, могла быть меньше размера массива, для указания конца строки применяется особый нулевой символ '\0', добавляемый в массив после собственно символов строки (это означает, в частности, что размер символьного массива должен быть по крайней мере на 1 больше максимального размера строки, которую требуется хранить в этом массиве). Аналогичным образом хранятся в памяти строковые литералы, то есть последовательности символов, заключенные в двойные кавычки, например, "abcd". Любой строковый литерал представляет собой символьный массив, содержащий все символы данного литерала плюс дополнительный нулевой символ. Тип строкового литерала — *char** (это означает, в частности, что операция + не может применяться к двум строковым литералам, так как, например, выражение "abcd" + "efgh" означало бы, что требуется сложить два указателя типа *char**, а операция сложения двух указателей не определена).

☑ Тем не менее, в языке C++ имеется возможность сцепления строковых литералов; для этого их достаточно *расположить рядом друг с другом*, например, "abcd""efgh"

(допустимо также разделять их одним или несколькими пробельными символами, в частности, пробелом или символом перехода на другую строку).

В дальнейшем символьный массив, предназначенный для хранения строки и обязательно содержащий нулевой элемент, будем называть *С-строкой*.

Для обработки С-строк в языке С++ предусмотрены функции, включенные в заголовочный файл *cstring*. Хотя данный файл добавлен в С++, в основном, в целях совместимости с программами, реализованными «в стиле языка С», в некоторых случаях функции из него могут оказаться полезными. Ниже описываются основные функции из заголовочного файла *cstring*, предназначенные для обработки С-строк, то есть массивов типа *char**, содержащих в качестве признака окончания строки символ '\0' (заметим, что все подобные функции имеют префикс *str*). Ознакомиться с описанием хотя бы некоторых функций из файла *cstring* полезно еще и потому, что они демонстрируют дополнительные проблемы, возникающие при обработке С-строк (в основном эти проблемы связаны с необходимостью постоянно следить за тем, чтобы символьные массивы содержали достаточное число элементов для хранения строковых данных).

☑ В последних версиях Visual C++ функции из файла *cstring* считаются «не рекомендованными к использованию» (deprecated), и при попытке их вызова компилятор выводит соответствующее предупреждение. Вместо них рекомендуется использовать «безопасные аналоги», реализованные в аналогичных заголовочных файлах библиотеки Visual C++. Однако, поскольку эта библиотека не является стандартной библиотекой языка C++, в данном пособии она не рассматривается. Собственно говоря, наиболее «безопасным» является отказ от любых подобных функций в пользу функций-членов класса *string* (см. следующий пункт).

<cstring>

```
size_t strlen(char* s);
```

Возвращает длину строки из массива *s*, то есть количество элементов массива, расположенных перед элементом с нулевым символом. Если в массиве *s*

отсутствует нулевой символ, то предсказать значение возвращаемого результата нельзя. Тип *size_t* является беззнаковым целым типом.

<cstring>

```
char* strcat(char* dst, char* src);
char* strncat(char* dst, char* src, size_t n);
```

Дописывает строку из массива *src* к строке, хранящейся в массиве *dst*, перезаписывая при этом нулевой символ, завершающий исходную строку *dst* (вариант с целочисленным параметром *n* обеспечивает дописывание не более чем *n* символов, включая нулевой). При вызове следует гарантировать, что массив *dst* содержит достаточное число элементов, чтобы вместить все дописанные символы (включая завершающий нулевой символ). Возвращает указатель на начало массива *dst* (иными словами, возвращаемое значение совпадает со значением первого параметра функции).

<cstring>

```
char* strcpy(char* dst, char* src);
char* strncpy(char* dst, char* src, size_t n);
```

Копирует строку из массива *src* в массив *dst* (вариант с целочисленным параметром *n* обеспечивает копирование *ровно n* символов). Функция *strcpy* всегда копирует строку вместе с завершающим нулевым символом. Функция *strncpy* копирует нулевой символ *только* в случае, если длина L строки *src меньше n* (в этой ситуации в массив *dst* после копирования строки *src* записывается $n - L$ нулевых символов, чтобы суммарное количество скопированных символов равнялось *n*).

При вызове следует гарантировать, что массив *dst* содержит достаточное число элементов, чтобы вместить все скопированные символы. Возвращает указатель на начало массива *dst* (иными словами, возвращаемое значение совпадает со значением первого параметра функции).

<cstring>

```
char* strchr(char* s, int c);
char* strrchr(char* s, int c);
```

Возвращает указатель на первое (функция *strchr*) или последнее (функция *strrchr*) вхождение символа с кодом *c* в строке из массива *s* (перед использованием параметра *c* его значение преобразуется к типу *unsigned char*). Если в строке такого символа нет, то возвращается нулевой указатель *NULL*.

<cstring>

```
char* strstr(char* s, int subs);
```

Возвращает указатель на первое вхождение строки из массива *subs* в строке из массива *s*. Если строка *s* не содержит вхождений строки *subs*, то возвращается нулевой указатель *NULL*.

<cstring>

```
int strcmp(char* s1, char* s2);
int strncmp(char* s1, char* s2, size_t n);
```

Возвращает результат сравнения строк, содержащихся в массивах *s1* и *s2*: 0, если строки равны, некоторое отрицательное число, если *s1* меньше, чем *s2*, некоторое положительное число, если *s1* больше, чем *s2*. Строки сравниваются посимвольно до обнаружения первой пары различных символов (в варианте с целочисленным параметром *n* анализируются не более *n* начальных символов строк); перед сравнением элементы массивов преобразуются к типу *unsigned char*, меньшим считается символ с меньшим кодом. Если одна из строк совпадает с начальной частью другой строки, то меньшей считается более короткая строка.

5.7. Класс string

Заголовочный файл *string*, входящий в шаблонную библиотеку C++, содержит реализации классов, предназначенных для эффективной и надежной

обработки строковых данных. Помимо класса *string*, предназначенного для работы со строками, состоящими из однобайтных символов типа *char*, в файле *string* реализован класс *wstring*, предназначенный для обработки строк с многобайтными символами *wchar_t*. Класс *wstring* здесь не рассматривается, отметим лишь, что он имеет такой же набор функций-членов, что и класс *string* (поскольку оба класса являются *специализациями* одного шаблонного класса *basic_string*).

5.7.1. Связь объектов-строк и символьных массивов. Варианты инициализации объектов-строк

Важнейшей особенностью объектов класса *string* является их совместимость (в обе стороны) с С-строками. Это особенно важно в силу того обстоятельства, что в других заголовочных файлах стандартной библиотеки С++ в качестве строковых данных используются, как правило, именно С-строки, то есть данные типа *char**.

С одной стороны, С-строку (в частности, строковый литерал) можно использовать для инициализации объекта типа *string* или последующего присваивания этому объекту нового значения:

```
string s = "abcd";
. . .
s = "1234";
```

С другой стороны, в классе *string* имеется функция-член *c_str*, позволяющая выполнить обратное преобразование.

---

string **<string>**

```
char* c_str();
```

Используя данную функцию, из любого объекта типа *string* можно получить символьный массив с той же строкой, дополненной нулевым символом, и, например, указать этот массив в качестве параметра типа *char**:

```
fstream f(s.c_str());
```

☑ Символьный массив, возвращаемый функцией *c_str*, является *константным* массивом *const char**, то есть не допускает изменения своих элементов (подобно литеральным строкам). Для получения «обычного» символьного массива *char** можно, например, создать копию С-строки, возвращаемой функцией *c_str*, используя функцию *strcpy* (см. п. 5.6).

Основным доводом в пользу применения объектов типа *string* (по сравнению с С-строками) является то, что при работе с ними можно не заботиться о распределении памяти. В частности, уже при применении простейшего варианта описания объекта типа *string* — *string s*; — созданный объект уже будет «готов к использованию» (он будет содержать пустую строку, а его функция *c_str* вернет указатель на символьный массив, содержащий единственный нулевой символ).

Приведем упрощенные описания еще нескольких конструкторов класса *string*.

string **<string>**

```
string(char* cstr, size_type count);
string(string& s, size_type start = 0,
               size_type count = npos);
string(size_type count, char c);
```

Первый вариант конструктора позволяет инициализировать объект *string* первыми *count* символами С-строки *cstr* (если строка в массиве *cstr* содержит менее *count* символов, то копируются все символы строки). Напомним, что тип *size_type* является беззнаковым целым типом, определенным в классе *string*, а константа *npos*, также определенная в классе *string*, соответствует особому значению типа *size_type*.

Второй вариант представляет собой несколько вариантов инициализации одного объекта типа *string* другим объектом этого же типа. В простейшем случае, когда второй и третий параметры отсутствуют, создаваемый объект будет

содержать копию строки *s*. Второй параметр (*start*) определяет, начиная с какого символа строки *s* будут копироваться символы в создаваемую строку, а третий параметр (*count*) определяет количество копируемых символов. Если третий параметр отсутствует или равен особому значению *string*::*npos*, то будут скопированы все символы строки *s*, начиная с символа с номером *start*.

Наконец, третий вариант конструктора позволяет инициализировать объект *string* строкой длины *count*, состоящей из одинаковых символов *c*.

При использовании описанных конструкторов следует применять стандартный синтаксис языка C++, например:

```
string s1(4,'a'),
    s2(s1,1),
    s3(s2);       // или   s3 = s2;
```

Приведенный фрагмент программы создает строку *s1*, содержащую 4 символа «а» («аааа») и строки *s2* и *s3*, каждая из которых содержит 3 символа «а» (поскольку в строку *s2* копируются все символы строки *s1*, начиная с символа, имеющего индекс 1, то есть со *второго* символа строки).

При работе с объектами класса *string* следует учитывать, что присваивание одному объекту другого приводит к *копированию* его содержимого. Например, если *s1* и *s2* описаны как *string*, то после выполнения операции *s2* = *s1* в объекте *s2* будет содержаться копия строки из *s1*, и эта копия не изменится, даже если будет изменен объект *s1*. При работе с С-строками ситуация будет иной. Если *cstr1* и *cstr2* описаны как символьные массивы (например, *char *cstr1*, * *cstr2* или *char cstr*[*10*], *cstr*[*10*]), то после присваивания *cstr2* = *cstr1* переменные *cstr1* и *cstr2* будут указывать на *один и тот же* участок памяти, выделенный под символьный массив (при этом, разумеется любое изменение строкового массива, сделанное с помощью переменной *cstr1*, приведет к тому, что строковый массив *cstr2* также изменится).

5.7.2. Операции, определенные для объектов-строк

Операция присваивания = позволяет присваивать объекту *s* типа *string* данные следующих типов:

- объект типа *string*;
- С-строка, то есть символьный массив;
- отдельный символ типа *char*.

В последнем случае объекту *s* будет присвоена односимвольная строка, содержащая указанный символ. Возможность присваивания объекту *string* отдельного символа, в совокупности с механизмом неявного преобразования типов, приводит к тому, что разрешенными являются и присваивания объекту *string* целых и даже вещественных чисел. В результате данный объект будет содержать односимвольную строку, состоящую из символа, код которого равен указанному числу (у вещественных чисел предварительно отбрасывается дробная часть). Так, любое из присваиваний *s1* = 33; *s2* = 33.4; *s3* = 33.8 приведут к тому, что соответствующая строка будет равна «!» (кроме того, при обработке последних двух присваиваний компилятор выведет предупреждение о возможной потере информации при приведении типа).

Операция сцепления + позволяет объединять операнды *s1* и *s2* в одну новую строку. Одним из операндов должен быть объект типа *string*, другим операндом может быть объект любого из типов, перечисленных выше при описании операции присваивания.

Операция дополнения += является комбинированной операцией, обеспечивающей дополнение строки типа *string*: выполнение оператора *s1* += *s2* приводит к тому же результату, что и выполнение оператора *s1* = *s1* + *s2*, в котором используются ранее описанные операции = и +. В классе *string* имеется функция *append*, также позволяющий дополнять строку новыми данными (типа *string*, *char** или *char*). В отличие от операции +=, в функции *append* при добавлении С-строки можно указывать количество добавляемых символов, а при до-

бавлении строки *string* номер первого из добавляемых символов и их количество.

Операции сравнения (==, !=, <, >, <=, >=) позволяют сравнивать строки между собой. Как обычно, строки сравниваются посимвольно до обнаружения первой пары различных символов; если одна из строк совпадает с начальной частью другой строки, то меньшей считается более короткая строка. Одним из операндов должна быть строка типа *string*; в качестве второго операнда можно указывать строку типа *string* или С-строку. Данные операции возвращают значение логического типа *bool*. Заметим, что в классе *string* имеется функция *compare*, также предназначенная для сравнения строк и позволяющая, в отличие от операций сравнения, указывать позиции в строках, с которых требуется проводить сравнение, и количество анализируемых при сравнении символов.

Операция индексирования [] позволяет выделять из строки типа *string* символ с требуемым номером (символы строки нумеруются от 0). В качестве индекса допустимо указывать числа в диапазоне 0–*L*, где *L* — длина строки; при этом символом с индексом *L* всегда будет нулевой символ. При попытке использования других индексов программа будет вести себя непредсказуемым образом. Важной особенностью операции индексирования является то, что с ее помощью можно также *изменять* требуемые символы строки, указывая выражение вида *s*[*i*] в левой части операции присваивания (в качестве индекса *i* в этом случае следует указывать числа в диапазоне от 0 до *L*–1; попытка заменить нулевой символ *s*[*L*] приведет к непредсказуемым результатам). Заметим, что в классе *string* имеется функция *at*, также позволяющая выделить из строки нужный символ. Так, выражения *s*[*3*] и *s.at*(*3*) эквивалентны (в частности, любое из них можно указать в левой части операции присваивания).

5.7.3. Длина строки, ее определение и изменение

В этом и двух последующих пунктах приводятся некоторые полезные функции класса *string*, причем для них указываются не все возможные перегруженные варианты.

---

string **<string>**

```
size_type length();
```

```
size_type size();
```

Любая из указанных функций возвращает текущую длину строки.

---

string **<string>**

```
size_type max_size();
```

Возвращает максимально возможную длину строки (при попытке создания строки большей длины возникает исключение *length_error*). В рассматриваемых реализациях строки типа *string* могут содержать до 4 гигабайт символов.

---

string **<string>**

```
void resize(size_type count, char ch = ' ');
```

Изменяет размер строки, полагая его равным *count*. Прежнее содержимое оставшейся части строки сохраняется. При увеличении размера строки к ней добавляется нужное число символов *ch* (если данный параметр не указан, то строка дополняется пробелами).

---

string **<string>**

```
void clear();
```

Очищает строку, полагая ее длину равной 0.

---

string **<string>**

```
bool empty();
```

Возвращает *true*, если строка является пустой, и *false* в противном случае. Вызов *s.empty*() эквивалентен проверке *s.length*() == 0 и выполняется быстрее, чем проверка вида s == "".

5.7.4. Преобразование строки

string **<string>**

```
string& insert(size_type pos, size_type count, char ch);
string& insert(size_type pos, char* cstr[,
               size_type count]);
string& insert(size_type pos, string& str[,
               size_type start, size_type count]);
```

Вставляет, начиная с указанной позиции *pos* строки, символ *ch*, С-строку *cstr* или строку *str*. Для С-строки *cstr* можно дополнительно указать количество *count* вставляемых символов; для строки *str* можно дополнительно указать номер *start* первого из добавляемых символов и их количество *count* (нумерация символов в строке, как обычно, начинается с нуля). Если количество символов, имеющихся в С-строке *cstr* или строке *str* (после символа с номером *pos*), меньше *count*, то вставляются все имеющиеся символы. Если параметр *pos* превосходит длину изменяемой строки или параметр *start* превосходит длину строки *str*, то возникает исключение *out_of_range*.

Возвращаемым значением функции является сама измененная строка.

Заметим, что вызов данной функции с параметром *pos*, равным длине исходной строки, эквивалентен вызову функции *append* с теми же параметрами, кроме параметра *pos* (поскольку фактически приводит к *дополнению* строки новыми данными).

string **<string>**

```
string& erase(size_type pos = 0, size_type count = npos);
```

Удаляет из строки *count* символов, начиная с позиции *pos*. Если *pos* больше длины строки, то возникает исключение *out_of_range*. Если параметр *count* отсутствует, равен *string::npos* или превышает число символов, расположенных после символа с номером *pos*, то удаляются все символы, начиная с символа с

номером *pos*. Заметим, что вызов функции без параметров эквивалентен вызову функции *clear*.

Возвращаемым значением функции является сама измененная строка.

---

string **<string>**

```
string substr(size_type start = 0,
              size_type count = npos);
```

Возвращает строку, содержащую *count* символов исходной строки, начиная с символа с номером *start*. Если *pos* больше длины строки, то возникает исключение *out_of_range*. Если параметр *count* отсутствует, равен *string::npos* или превышает число символов, расположенных после символа с номером *pos*, то в результирующую строку копируются все символы, начиная с символа с номером *pos*. Вызов функции без параметров позволяет получить копию исходной строки.

---

string **<string>**

```
string& replace(size_type pos1, size_type num1,
              size_type count, char ch);
string& replace(size_type pos1, size_type num1,
              char* cstr[, size_type num2]);
string& replace(size_type pos1, size_type num1,
              string& str[,
              size_type pos2, size_type num2]);
```

Комбинированная функция, позволяющая сразу выполнить удаление части исходной строки и вставку на место удаленной части новых данных. В любом из вариантов функции вначале выполняется удаление *num1* символов исходной строки, начиная с позиции *pos1* (если *num1* превышает число символов в строке, расположенных после символа с номером *pos1*, то удаляются все эти символы). Затем, начиная с позиции *pos1*, в строку вставляются новые данные, а именно:

- *count* копий символа *ch* в первом варианте функции,
- С-строка *cstr* (или *num2* начальных символов С-строки, если указан параметр *num2*) во втором варианте функции,
- строка *str* (или подстрока строки *str*, начинающаяся с позиции *pos2* и имеющая длину *num2*, если указаны параметры *pos2* и *num2*) в третьем варианте функции.

Если значение параметра *num2* превышает количество символов в С-строке *cstr* или символов в строке *str*, расположенных после символа с номером *pos2*, то вставляются все имеющиеся символы.

Если параметр *pos1* больше длины исходной строки (а также если параметр *pos2* больше длины строки *str*), то возникает исключение *out_of_range*.

Возвращаемым значением функции является сама измененная строка.

5.7.5. Поиск в строке

string **<string>**

```
size_type find(char ch, size_type pos = 0);
size_type find(char* cstr, size_type pos = 0);
size_type find(string& str, size_type pos = 0);
```

Возвращает номер позиции, начиная с которой в исходной строке содержится первое вхождение символа *ch*, С-строки *cstr* или строки *str*. Поиск начинается с позиции *pos* или с начальной позиции 0, если параметр *pos* не указан. Если параметр *pos* больше длины строки, то возникает исключение *out_of_range*. Если в строке отсутствуют вхождения требуемых данных, то возвращается *string::npos*.

Имеются также функции *find_first_of* и *find_first_not_of* с теми же вариантами наборов параметров. Функция *find_first_of* возвращает номер первого символа исходной строки, который совпадает с символом *ch* или содержится среди символов С-строки *cstr* либо строки *str*. Функция *find_first_not_of* возвращает

номер первого символа исходной строки, который не совпадает с символом *ch* или не содержится среди символов С-строки *cstr* либо строки *str*. Если требуемый символ отсутствует, то эти функции возвращают *string::npos*.

string **<string>**

```
size_type rfind(char ch, size_type start = npos);
size_type rfind(char* cstr, size_type start = npos);
size_type rfind(string& str, size_type start = npos);
```

Возвращает номер позиции, начиная с которой в исходной строке содержится последнее вхождение символа *ch*, С-строки *cstr* или строки *str*. Поиск начинается с позиции *pos* или с конечной позиции, если параметр *pos* не указан или равен *string::npos*. Если параметр *pos* больше длины строки, то возникает исключение *out_of_range*. Если в строке отсутствуют вхождения требуемых данных, то возвращается *string::npos*.

Имеются также функции *find_last_of* и *find_last_not_of* с теми же вариантами наборов параметров. Функция *find_last_of* возвращает номер последнего символа исходной строки, который совпадает с символом *ch* или содержится среди символов С-строки *cstr* либо строки *str*. Функция *find_last_not_of* возвращает номер последнего символа исходной строки, который не совпадает с символом *ch* или не содержится среди символов С-строки *cstr* либо строки *str*. Если требуемый символ отсутствует, то эти функции возвращают *string::npos*.

5.8. Заголовочные файлы и классы C++, связанные с потоками

Ввод-вывод в программах на языке С++ рекомендуется выполнять с помощью специальных классов — *потоков ввода-вывода*, реализованных в нескольких стандартных заголовочных файлах, из которых основными являются *iostream* и *fstream*. Указанные файлы предоставляют средства для организации ввода-вывода из стандартного устройства (консольный ввод-вывод) и из фай-

лов. Кроме того, потоки можно использовать для форматированного вывода данных в текстовые строки и чтения данных из текстовых строк; для этого предназначены строковые потоки ввода-вывода, реализованные в заголовочных файлах *sstream* и *strstream*.

Все классы-потоки связаны отношениями наследования и образуют достаточно сложную иерархию.

Базовым предком всех классов-потоков является класс *ios_base*. От этого класса порождается базовый *шаблонный* класс *basic_ios*, от которого, в свою очередь, порождается большой набор других шаблонных классов. Поскольку в программах обычно используются конкретные *специализации* шаблонов, сразу отметим, что двумя основными специализациями шаблона *basic_ios* являются *ios* (базовый класс для всех потоков, основанных на однобайтных символьных данных char) и *wios* (базовый класс для всех потоков, основанных на многобайтных символьных данных *wchar_t*). Эти специализации, наряду с набором функций-манипуляторов (см. п. 5.13), определены в заголовочном файле *ios*.

Для простоты ограничимся иерархией, основанной на классе *ios*. От этого класса порождаются базовый поток ввода *istream* и базовый поток вывода *ostream*. Определяется также базовый поток ввода-вывода *iostream*, который порождается сразу от двух классов: *istream* и *ostream*. Классы *istream*, *ostream* и *iostream* определены в заголовочном файле *iostream*. Именно в этих классах определены все необходимые операции и функции ввода-вывода; потомки этих классов лишь изменяют реализации этих функций в зависимости от природы источника или приемника данных.

Первая группа потомков этих классов обеспечивает ввод-вывод из файлов (классы из этой группы определены в заголовочном файле *fstream*):

- *ifstream* — файловый поток ввода (потомок *istream*),
- *ofstream* — файловый поток вывода (потомок *ostream*),
- *fstream* — файловый поток ввода-вывода (потомок *iostream*).

Вторая группа потомков этих классов обеспечивает ввод-вывод из объектов типа *string* (классы из этой группы определены в заголовочном файле *sstream*):

- *istringstream* — строковый поток ввода (потомок *istream*),
- *ostringstream* — строковый поток вывода (потомок *ostream*),
- *stringstream* — строковый поток ввода-вывода (потомок *iostream*).

Подчеркнем, что все указанные классы являются *специализациями* соответствующих шаблонных классов, однако при их использовании в программе это обстоятельство никак не проявляется, и можно считать, что данные классы являются обычными потоковыми классами, ориентированными на потоки, содержащие однобайтовые символы *char*.

☑ Заметим, что имена соответствующих *шаблонных* классов получаются из имен описанных выше классов присоединением префикса *basic_* (например, *basic_iostream*, *basic_ifstream*, *basic_ostringstream* и т. д.), а классов-специализаций, ориентированных на потоки *многобайтовых символов wchar_t*, получаются из имен описанных выше классов присоединением префикса *w* (например, *wiostream*, *wifstream*, *wostringstream* и т. д.).

5.9. Обработка ошибок ввода-вывода

Стандартная библиотека потокового ввода-вывода для языка C++ ориентирована, прежде всего, на следующую схему работы с потоком ввода: «читать данные из потока, пока очередная операция чтения не закончится неудачей». Поэтому никакая ошибка, связанная с потоками, не приводит к аварийному завершению программы. Вместо этого изменяется состояние соответствующего потока, *после чего все операции, связанные с этим потоком, блокируются*.

☑ Следует отметить, что подобный подход не вполне соответствует современной практике обработки ошибок, основанной на исключениях, а также другим, достаточно естественным, схемам работы с потоком, в которых конец потока распознается еще до того, как очередная операция чтения приведет к ошибке.

Для обнаружения ошибки программа должна явным образом проверять состояние потока, а для продолжения работы с потоком (если это возможно) — сбрасывать состояние ошибки.

Опишем средства, предусмотренные для проверки и изменения состояния потока; все эти средства реализованы в классе *ios* (точнее, в классе-шаблоне *basic_ios*) и поэтому доступны для всех рассматриваемых далее потоков.

---

ios **<ios>**

```
bool good();
bool bad();
bool fail();
bool eof();
```

Логические функции, позволяющие проверить текущее состояние потока. Опишем для каждой функции, что означает ситуация, при которой эта функция возвращает значение *true*:

- *good*: в данный момент ошибок, связанных с потоком, не обнаружено;
- *bad*: при работе с потоком произошла *фатальная ошибка*, которую, скорее всего, не удастся исправить;
- *eof*: прочесть очередной элемент данных не удалось, так как обнаружен *конец потока*;
- *fail*: произошла какая-то ошибка; возможно, это фатальная ошибка (тогда значение *true* вернет и функция *bad*), возможно, это ошибка, связанная с концом потока (тогда значение *true* вернет и функция *eof*), возможно, это другая *ошибка ввода-вывода или преобразования данных* (в этой ситуации можно попытаться продолжить работу с потоком, сбросив состояние ошибки).

Важно подчеркнуть, что функция *eof* возвращает *true* после попытки прочесть очередной элемент за концом потока. Эту особенность следует учиты-

вать, поскольку во многих языках программирования (например, в Паскале) функции, проверяющие конец потока (в частности, файла) ведут себя по-другому: возвращают значение *true*, как только прочитан последний элемент из потока, и поэтому механический перенос соответствующих конструкций в программу на C++ может привести к ее неверной работе (см. пример в конце данного пункта).

Следует также заметить, что фатальные ошибки при работе с потоками возникают достаточно редко; даже при попытке открыть несуществующий файл или создать файл с недопустимым именем функция *bad* соответствующего файлового потока возвращает *false* (хотя функция *fail*, естественно, возвращает *true*).

---

ios **<ios>**

```
void clear();
```

Сбрасывает состояние ошибки для данного потока (в результате функция *good* вернет значение *true*, а функции *eof*, *bad* и *fail* — значение *false*). Данную функцию следует использовать в ситуации, когда можно рассчитывать на восстановление потока после обнаружения ошибки.

☑ На самом деле функция *clear* позволяет установить поток в любое состояние, указав комбинацию соответствующих *флагов состояния* в качестве своего параметра, однако эту возможность, как и другие возможности, связанные с использование флагов состояния, мы не будем описывать.

Для того чтобы еще более упростить проверку состояния потока, для него определены две операции.

Операция ! («отрицания»), примененная к самому потоку, возвращает *true*, если поток находится в состоянии ошибки, и *false* в противном случае.

Операция преобразования (*void**) (то есть преобразования к нетипизированному указателю), примененная к потоку, возвращает нулевой указатель (то есть 0), если поток находится в состоянии ошибки, и ненулевое значение в про-

тивном случае. Для применения подобной операции достаточно указать имя потока в условии оператора *if* или *while*, например, *while (f)* … (где *f* — некоторый поток).

Приведем пример, иллюстрирующий применение описанных выше средств. Предположим, что поток *f* содержит 4 символа: «A», «B», «C», «D» и требуется сформировать строку, содержащую эти символы в том же порядке. Для чтения символов из потока будем использовать функцию *get*. В приводимых далее фрагментах программ предполагается, что поток *f* уже создан и доступен для чтения, а переменная *s* описана как *string*.

Вначале приведем ошибочный вариант решения, который, однако, представляется весьма «естественным» для программистов, знакомых с языком Паскаль или Visual Basic:

```
while (!f.eof())
    s += f.get();
```

В результате строка s получит значение "ABCD ", то есть будет содержать пять символов, последним из которых является *пробел*. Объясняется это тем, что после чтения последнего символа («D») из потока, поток еще не перейдет в состояние «обнаружен конец потока», поэтому при следующем вызове функция *eof* вернет *false*, и будет выполнена еще одна (лишняя) итерация цикла. На этой итерации будет предпринята попытка прочесть символ за концом потока. Разумеется, эта попытка закончится неудачей; в результате функция *get* вернет пробел, а поток *f* перейдет в состояние «обнаружен конец потока». При очередном вызове функции *eof* она вернет *true*, что обеспечит завершение цикла *while*.

Заметим, что аналогичный результат будет получен и при использовании следующих вариантов заголовка цикла: *while (!f.fail)* … или *while (f)* … (в последнем случае к потоку *f* будет неявно применена операция (*void**) преобразования к нетипизированному указателю).

Для правильного формирования строки можно выполнять «упреждающее» считывание символа из потока:

```
char c = f1.get();
while (f1)
{
    s += c;
    c = f1.get();
}
```

В этом варианте мы использовали наиболее краткое из возможных условий в цикле *while*. Приведенный вариант решения приводит к необходимости дублирования функции *get*, обеспечивающей считывание символа из файла, а также описания вспомогательной символьной переменной *c*.

От дублирования функции *get* можно избавиться, если использовать ее вариант с одним символьным параметром *get(c)*; в этом варианте прочитанный из потока символ записывается в параметр *c*, а функция возвращает сам поток (при попытке чтения за концом потока значение параметра *c* не изменяется):

```
char c;
while (f1.get(c))
    s += c;
```

Это, по-видимому, наиболее краткое решение, реализованное в рамках схемы «читать до первой ошибки». В этом решении нельзя избавиться от вспомогательной переменной *c*, которая, к тому же, является внешней по отношению к циклу *while*. Заметим также, что можно привести примеры задач, в которых не слишком удобно размещать операторы ввода данных в условии цикла.

В заключение приведем правильный вариант решения, реализованный в рамках схемы «читать до конца потока» (подобная схема является традиционной для многих других языков программирования, в том числе Паскаля, Visual

Basic и языков платформы .NET). Для реализации этой схемы средствами стандартной библиотеки C++ приходится пользоваться функцией *peek*:

```
while (f1.peek() != -1)
    s += f1.get();
```

5.10. Перечисления, связанные с обработкой потоков

В классе *ios* определено несколько перечислимых типов, связанных с обработкой потоков, а также константы, соответствующие элементам этих типов. Ниже приводится описание некоторых из перечислимых типов и связанных с ними констант.

5.10.1. ios::openmode

Задает режим, в котором открывается и функционирует поток. Включает константы:

- *ios::app* — обеспечивает переход в конец потока перед каждой операцией записи;
- *ios::ate* — обеспечивает переход в конец потока при открытии потока;
- *ios::in* — включает режим чтения при открытии потока;
- *ios::out* — включает режим записи при открытии потока;
- *ios::trunc* — обеспечивает удаление прежнего содержимого потока при открытии потока;
- *ios::binary* — включает режим двоичного ввода-вывода при открытии потока (по умолчанию для потока устанавливается текстовый режим).

Константы, связанные с режимом потока, можно комбинировать с помощью операции | (побитовое ИЛИ). Например, режим двоичного ввода-вывода устанавливается комбинацией *ios::in* | *ios::out* | *ios::binary*. Как правило, режим

потока указывается в качестве параметра его конструктора или другой функции, обеспечивающей открытие потока.

5.10.2. ios::seekdir

Определяет позицию, от которой отсчитывается смещение потокового указателя при выполнении прямых переходов в потоке. Включает константы:

- *ios::beg* — смещение определяется относительно начала потока;
- *ios::cur* — смещение определяется относительно текущего положения указателя, связанного с потоком;
- *ios::end* — смещение определяется относительно конца потока.

Указанные константы используются в функциях *seekg* и *seekp*.

5.10.3. ios::fmtflags

Устанавливает значения свойств форматирования при вводе и выводе данных. Флаги форматирования разделены на несколько групп: основание системы счисления, формат вывода вещественных чисел, выравнивание полей. Есть также набор независимых флагов. Часть флагов используется только при выводе данных, некоторые предназначены для настройки потоков ввода, остальные могут использоваться как при вводе, так и при выводе.

Основание системы счисления при вводе и выводе целочисленных значений задается константами:

- *ios::dec* — десятичная система счисления (используется по умолчанию);
- *ios::hex* — шестнадцатеричная система счисления;
- *ios::oct* — восьмеричная система счисления.

При установке значений флагов этой группы удобно использовать битовую маску *ios::basefield*.

При выводе вещественных чисел используются следующие флаги форматирования:

- *ios::fixed* — формат вывода с фиксированной точкой;
- *ios::scientific* — формат вывода в научной нотации (напр. 1.23e+26).

При установке значений флагов этой группы удобно использовать битовую маску *ios::floatfield*.

Выравнивание полей вывода определяется константами:

- *ios::internal* — поле вывода заполняется *символами-заполнителями* в указанной позиции;
- *ios::left* — поле вывода заполняется *символами-заполнителями* справа (то есть выводимое значение выравнивается по левому краю);
- *ios::right* — поле вывода заполняется *символами-заполнителями* слева (то есть выводимое значение выравнивается по правому краю).

При установке значений флагов этой группы удобно использовать битовую маску *ios::adjustfield*.

К независимым флагам форматирования относятся:

- *ios::boolalpha* — ввод и вывод логических значений в символьной форме *true* и *false* (по умолчанию используется значение 0 для *false* и 1 для *true*);
- *ios::showbase* — вывод целочисленных значений с указанием соответствующей системы счисления;
- *ios::showpoint* — вывод вещественных значений с обязательным указанием десятичной точки (по умолчанию, если вещественное число имеет нулевую дробную часть, она не выводится);
- *ios::showpos* — вывод неотрицательных значений с явным указанием знака «+»;
- *ios::skipws* — автоматический пропуск пробельных символов при чтении из потока ввода операцией >>;
- *ios::uppercase* — вывод всех символов в верхнем регистре;

- *ios::unitbuf* — автоматическое сбрасывание (*flush*) выходного буфера после каждой операции записи в поток.

Способы установки и снятия флагов форматирования описаны в п. 5.13.1.

5.11. Базовый поток ввода: класс istream

Базовым предком потоков ввода является класс *istream*, порожденный от класса *ios* и реализованный в заголовочном файле *istream*. Среди его потомков следует отметить классы *ifstream* (файловый поток ввода) и *istringstream* (строковый поток ввода).

Для ввода данных из потока *istream* (и всех его потомков) предусмотрена операция >> и набор функций, основные из которых описываются в данном пункте.

---

istream **<istream>**

`операция >>`

Данная операция обеспечивает форматированный ввод данных из потока ввода *f* в указанную переменную *v* и имеет вид *f* >> *v*. Она перегружена для всех базовых типов C++, в том числе типов *char*, *int*, *double*. Кроме того, для ввода строковых данных в качестве переменной *v* можно указывать символьный массив и переменную типа *string*.

Возвращаемым значением операции >> является поток, для которого она вызвана, что позволяет использовать для ввода нескольких данных выражения, содержащие несколько операций >>, например:

```
f >> v1 >> v2 >> v3;
```

В указанном операторе вначале выполняется операция *f* >> *v1*, считывающая элемент данных в переменную *v1*. Эта операция возвращает тот же поток *f*, который, таким образом, оказывается левым операндом следующей операции >>, вводящей данные в переменную *v2*. Эта операция, в свою очередь, возвращает поток *f*, который оказывается операндом последней операции >>. Данный

процесс можно сделать более наглядным, если явным образом расставить скобки в приведенном выражении:

```
((f >> v1) >> v2) >> v3;
```

В качестве правого операнда операции >> можно также использовать манипуляторы (см. п. 5.13).

Целый ряд функций обеспечивает *неформатированный* ввод. Это означает, что во всех таких функциях предоставляется буфер для заполнения данными, считываемыми из потока. В качестве буфера могут использоваться символы (то есть, фактически, байты), символьные массивы и строки, а с учетом операций преобразования типа буфером может стать любой массив примитивных типов. Первая часть таких функций ориентирована на ввод символьных данных.

---

istream **<istream>**

```
int get();
istream& get(char& c);
istream& get(char* cstr, streamsize count);
istream& get(char* cstr, streamsize count, char delim);
istream& getline(char* cstr, streamsize count);
istream& getline(char* cstr, streamsize count,
                 char delim);
```

Функция *get* возвращает код символа, прочитанного из потока ввода. При попытке чтения за концом потока поток приходит в ошибочное состояние. Вторая форма функции *get* принимает ссылку на символ *c* в качестве параметра и присваивает ему введенное значение. Третья и четвертая формы предназначены для чтения из потока ввода последовательности символов вплоть до обнаружения символа-разделителя, в качестве которого используется либо символ перевода строки (в третьей форме), либо символ *delim* (в четвертой форме). Сам символ-разделитель *остается в потоке* и в строку не попадает. В конец строки автоматически дописывается символ завершения строки '\0'. Функции *getline*

действуют похожим образом, за тем исключением, что символ-разделитель *считывается из потока*, хотя в результирующую строку не попадает. Параметр *count* ограничивает максимальное количество вводимых символов. Программист должен обеспечить соответствующий размер символьного буфера с учетом добавляемого терминального символа '\0'. Отметим, что функции *getline* являются предпочтительным способом для чтения С-строк из потока, поскольку они ограничивают максимальное количество вводимых символов. При использовании операции >> таких ограничений нет, что может привести к переполнению буфера и неопределенным последствиям.

<string>

```
istream& getline(istream& strm, string& str);
istream& getline(istream& strm, string& str, char delim);
```

Эти функции используются для чтения из потока строк типа *string*. Строка читается вплоть до разделителя (символ перевода строки или *delim*), сам разделитель считывается из потока, но в строку не попадает. Эти функции удобно использовать для построчного чтения потока:

```
string s;
while (getline(cin, s))
{
  // Обработка очередной строки s
}
```

---

istream **<istream>**

```
istream& read(char* cstr, streamsize count);
streamsize readsome(char* cstr, streamsize count);
```

Функции *read* и *readsome* предназначены для чтения символов из потока без учета разделителей. Параметр *count* ограничивает максимальное количество считываемых символов. При отсутствии в потоке требуемого количества символов функция *read* переводит его в ошибочное состояние. В противополож-

ность этому функция *readsome* читает из потока столько символов, сколько возможно, и возвращает количество реально прочитанных символов. Вместо символьного буфера можно использовать любой другой буфер, преобразовав указатель на него к типу *char**, при этом важно не ошибиться в указании размера буфера:

```
double* p = new double[100];
is.read(static_cast<char*>(p), sizeof(double)*100);
```

Функции *read* и *readsome* считают символы перевода строки обычными символами, они также *не* добавляют к результирующему буферу терминальные символы.

---

istream **<istream>**

```
istream& ignore();
istream& ignore(streamsize count);
istream& ignore(streamsize count, int delim);
```

Функции *ignore* позволяют пропустить (извлечь из потока, но никуда не сохранить) символы, находящиеся в потоке. В первой форме пропускается один символ, во второй — *count* символов, а в третьей пропускаются все символы вплоть до символа-разделителя *delim*, включая его, причем пропускается не более чем *count* символов.

---

istream **<istream>**

```
int peek();
istream& unget();
istream& putback(char c);
```

Функция *peek* возвращает символ, который должен быть прочитан следующей функцией чтения потока. В частности, это позволяет распознать конец потока (см. п. 5.9.). Функция *unget* помещает назад в поток последний прочитанный из него символ. Функция *putback* позволяет поместить в поток ввода

произвольный символ *c*. Отметим, что не все потоки позволяют выполнять эти операции. Если поток их не поддерживает, то при попытке их вызова он переходит в состояние *фатальной* ошибки.

istream **<istream>**

```
istream& seekg(streampos pos);
istream& seekg(streamoff off, ios_base::seekdir dir);
streampos tellg();
```

Описание и примеры использования этих функций приводятся в п. 3.2.2.

Все функции, определенные в классе *istream*, надлежащим образом переопределяются в любом из его потомков, таким образом, их можно использовать также для файловых и строковых потоков ввода (см. п. 5.14–5.15).

5.12. Базовый поток вывода: класс ostream

Базовым предком потоков вывода является класс *ostream*, порожденный от класса *ios* и реализованный в заголовочном файле *ostream*. Среди его потомков следует отметить классы *ofstream* (файловый поток ввода) и *ostringstream* (строковый поток ввода).

Для вывода данных в поток *ostream* (и все его потомки) предусмотрена операция << и набор функций, некоторые из которых описываются в данном пункте.

ostream **<ostream>**

```
операция <<
```

Операция << является основным механизмом форматированного вывода данных примитивных типов и типов, определенных пользователем. Ее использование аналогично использованию операции >> для потоков ввода:

```
int a = 7;
double b = 12.43;
string s = "Hello";
```

```
os << a << b << s;
```

Здесь в поток вывода *os* выводятся целое число, вещественное число и строка. Управление правилами вывода осуществляется с помощью функций форматирования и манипуляторов (см. п. 5.13).

ostream **<ostream>**

```
ostream& put(char c);
ostream& write(const char* str, streamsize count);
```

С помощью этих функций можно вывести в поток символ (функция *put*) или массив символов (функция *write*). С помощью преобразования типов функция *write* позволяет также записать в поток буфер с данными любых типов. Параметр *count* определяет размер выводимого буфера в байтах. Функция *write* обычно используется для работы с двоичными файлами.

ostream **<ostream>**

```
ostream& flush();
```

Данные, выводимые в поток несколькими операциями, обычно накапливаются в некотором буфере в памяти, и только после этого выводятся на диск или на терминал, то есть на то устройство, с которым связан поток. При необходимости сбросить все содержимое буфера следует вызвать функцию *flush*. Отметим, что при закрытии файловых потоков сбрасывание буфера на диск происходит автоматически.

ostream **<ostream>**

```
ostream& seekp(streampos pos);
ostream& seekp(streamoff off, ios_base::seekdir dir);
streampos tellp();
```

Описание и примеры использования этих функций приводится в п. 3.2.2.

5.13. Форматирование данных. Манипуляторы

Библиотека ввода/вывода языка C++ предоставляет две основные возможности для управления форматом вводимых и выводимых данных: флаги форматирования и манипуляторы. В качестве конкретных примеров правил форматирования можно привести регулировку количества знаков, выводимых после десятичной точки, или установку общей ширины поля вывода и его выравнивания.

5.13.1. Чтение и установка флагов форматирования

Флаги форматирования управляют правилами ввода и вывода значений разных типов, они объявлены в перечислении *ios::fmtflags* (см. п. 5.10.3). Для установки и снятия этих флагов предусмотрен целый ряд функций.

Первая группа таких функций позволяет добавить и снять значения конкретных флагов.

ios_base **<ios>**

```
fmtflags setf(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl, fmtflags mask);
void unsetf(fmtflags mask);
```

Функция *setf* добавляет к набору флагов потока указанный флаг или их комбинацию:

```
cout.setf(ios::showpos | ios::uppercase);
```

Для установки флагов, принадлежащих одной группе (см. п. 5.10.3), следует пользоваться второй формой функции *setf* с указанием битовой маски. Так, в следующей строке устанавливается шестнадцатеричный формат для вводимых целых чисел:

```
cin.setf(ios::hex, ios::basefield);
```

Обе формы функции *setf* возвращают общий набор флагов, установленных для потока.

Функция *unsetf* снимает значение переданного ей флага:

```
cout.unsetf(ios::uppercase);
```

Второй способ задания флагов форматирования позволяет заменить весь набор новым значением.

---

ios_base **<ios>**

```
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
```

Функция *flags* возвращает весь набор флагов, установленных для потока. Эта же функция во второй форме позволяет установить полностью новый набор. Пара этих функций обычно используется для сохранения значения флагов (перед их временным изменением) и восстановления сохраненных значений:

```
ios::fmtflags oldFlags = cout.flags();
cout.setf(ios::showpos | ios::showbase |
    ios::uppercase);
cout.setf(ios::internal, ios::adjustfield);
cout << hex << x << endl;
cout.flags(oldFlags);
```

Эти же функции можно использовать для задания флагов форматирования, которые уже были установлены для другого потока. Однако ту же задачу можно решить проще с помощью функции *copyfmt*, специально предназначенной для копирования форматных настроек.

---

ios **<ios>**

```
ios& copyfmt(const ios& rhs);
```

Функция *copyfmt* позволяет скопировать значения всех флагов форматирования, установленных для другого потока, передаваемого ей в качестве параметра.

5.13.2. Манипуляторы

Манипуляторы представляют собой функции специального типа, которые можно «выводить» в поток и «вводить» из него. Они позволяют выполнять основные операции форматирования проще, чем средствами флагов форматирования. Простейшие манипуляторы, не принимающие никаких параметров, объявляются вместе с классами потоков, с которыми они должны использоваться. Таким образом, никакие дополнительные заголовочные файлы не нужны. Все манипуляторы, принимающие параметры, объявлены в заголовочном файле *iomanip*.

Приведем примеры некоторых простейших манипуляторов.

<ostream>

```
endl
ends
flush
```

Манипулятор *endl* выводит в поток вывода символ перевода строки. Манипулятор *ends* позволяет вывести в поток терминальный символ '\0' (это имеет смысл при формировании символьных буферов средствами потоков). Манипулятор *flush* выполняет задачу, аналогичную функции *flush*, то есть принудительно сбрасывает на устройство, связанное с потоком, буфер памяти, в котором накапливалась выводимая информация.

С помощью манипуляторов можно также выполнять операции установки и снятия значений флагов форматирования.

<iomanip>

```
setiosflags(ios_base::fmtflags mask)
resetiosflags(ios_base::fmtflags mask)
```

Манипулятор *setiosflags* устанавливает значения переданных ему флагов для того потока, в который он выводится или из которого вводится, а манипу-

лятор *resetiosflags* снимает их:

```
cout << resetiosflags(ios::adjustfield)
     << setiosflags(ios::left);
```

Для большинства задач форматирования существуют специализированные манипуляторы, пользоваться которыми гораздо удобнее.

5.13.3. Форматирование полей вывода

Поле для вывода значения любого типа может быть охарактеризовано следующими достаточно общими параметрами: шириной (количеством символов, отводимых на вывод этого значения), символом-заполнителем (это символ, который должен выводиться в позиции, не заполненные самим значением) и выравниванием (размещением выводимого значения относительно границ поля вывода). Для управления этими параметрами могут использоваться как специальные функции-члены потоковых классов, так и манипуляторы.

---

ios_base **<ios>**

```
streamsize width() const;
streamsize width(streamsize wide);
```

Функция *width* позволяет узнать (в первой форме) и изменить (во второй форме) ширину поля вывода. Вторая форма возвращает значение ширины, которое было задано до вызова этой функции. Заметим, что функция *width* устанавливает *минимальный* размер поля вывода; если выводимое значение оказывается шире, то этот размер игнорируется. По умолчанию значение ширины поля вывода равно 0, что означает, что минимальный размер поля не ограничен. После осуществления операции вывода, перед которой было вызвано изменение ширины, значение по умолчанию восстанавливается. Таким образом, функция *width* влияет только на одно значение, выводимое после ее вызова.

ios **<ios>**

```
char fill() const;
char fill(char fillch);
```

Функция *fill* позволяет установить символ-заполнитель для полей вывода. В отличие от функции *width*, ее действие продолжается вплоть до установки другого символа-заполнителя.

<iomanip>

```
setw (int n)
setfill (char c)
```

Манипуляторы *setw* и *setfill* можно использовать в качестве полного аналога соответствующих функций:

```
cout << setw(5) << 12 << 6 << setw(10) << 6.7 << endl;
```

Здесь на вывод числа 12 отводится 5 позиций, а на вывод числа 6.7 — 10 позиций. Заметим, что число 6 выводится в поле, ширина которого не ограничивается (то есть для него действует значение по умолчанию).

Выравнивание поля вывода может устанавливаться с помощью флагов форматирования из группы *ios::adjustfield*, а также посредством манипуляторов.

<ios>

```
left
right
```

Эти манипуляторы предназначены для установки выравнивания поля вывода по левому краю или по правому краю.

5.13.4. Форматирование целых и вещественных чисел

Среди средств форматирования числовых данных можно отметить ввод и вывод целых чисел в разных системах счисления или регулировку количества символов, выводимых в вещественном числе после десятичной точки. Управ-

ление этими средствами может осуществляться флагами форматирования из групп *ios::basefield* и *ios::floatfield*, а также специальными функциями и манипуляторами.

<ios>

```
dec
oct
hex
showbase
noshowbase
```

Манипуляторы *dec*, *oct* и *hex* переводят поток в режим ввода или вывода целых чисел, записанных в десятичной, восьмеричной и шестнадцатеричной системах счисления. По умолчанию все целые числа выводятся в десятичной системе счисления. Манипуляторы *showbase* и *noshowbase* позволяют соответственно выводить и не выводить признак системы счисления ('0x' для шестнадцатеричной системы счисления и '0' для восьмеричной).

---

ios_base **<ios>**

```
streamsize precision() const;
streamsize precision(streamsize prec);
```

Функция *precision* устанавливает количество символов, выводимых после десятичной точки, возвращая значение, установленное ранее.

<ios>

```
showpoint
noshowpoint
fixed
scientific
```

Манипуляторы *showpoint* и *noshowpoint* управляют отображением десятичной точки в вещественных числах с нулевой дробной частью. Манипулятор

fixed устанавливает формат вывода вещественных чисел с фиксированной точкой, а манипулятор *scientific* позволяет выводить числа в научной нотации (например, 1.87e+21).

<iomanip>

```
setprecision(int n)
```

Этот манипулятор аналогичен действию функции *precision*.

<ios>

```
showpos
noshowpos
```

С помощью этих манипуляторов можно включать (*showpos*) и выключать (*noshowpos*) вывод знака '+' для неотрицательных чисел. По умолчанию знак '+' не выводится.

5.14. Файловые потоки ввода-вывода

Файловые потоки являются производными от общих потоковых классов *istream*, *ostream* и *iostream*, поэтому они наследуют все их возможности. Укажем здесь только то, что их отличает. Здесь приводится описание нескольких функций класса *fstream*, функции классов *ifstream* и *ofstream* почти полностью аналогичны.

---

fstream **<fstream>**

```
explicit fstream(const char* filename,
                 ios_base::openmode mode = ios_base::in |
                 ios_base::out);
void open(const char* filename,
                 ios_base::openmode mode = ios_base::in |
                 ios_base::out);
```

```
void close();
bool is_open();
```

Конструктор класса *fstream* позволяет указать имя файла, с которым связывается поток. Если имя файла содержится в переменной типа *string*, то необходимо преобразовать его в форму *char**:

```
string filename = "file.dat";
fstream f(filename.c_str());
```

Функция *open* с теми же параметрами позволяет открыть файл после создания объекта потока. Файловый поток должен закрываться функцией *close*. Проверить, открыт ли файл в настоящее время, можно с помощью функции *is_open*.

После открытия файловый поток *fstream* может выполнять как операции чтения, так и операции записи. Файловый поток ввода *ifstream* открывается в режиме чтения, а поток вывода *ofstream* в режиме записи.

5.15. Строковые потоки ввода-вывода

Строковые потоки предназначены для чтения и записи данных в строки. Фактически строки играют здесь роль буфера для хранения данных. Обычно строковые потоки используют для предварительной подготовки данных перед их выводом или для реализации отложенного ввода. Еще один вариант применения — преобразование значений разных типов в строковое представление. Так как строковые потоки являются наследниками общих потоковых классов, для работы с ними можно использовать все рассмотренные ранее приемы, в том числе операции >> и <<, флаги форматирования и манипуляторы. Специальные функции строковых потоков позволяют указать строку, используемую в качестве буфера данных, а также прочесть эту строку с накопленными в ней данными.

istringstream **<sstream>**

```
explicit istringstream(openmode which = ios_base::in);
explicit istringstream(const string & str,
                       openmode which = ios_base::in);
```

Эти конструкторы создают объект класса *istringstream*. Вторая форма конструктора позволяет указать строку, используемую в качестве источника ввода.

Поток *istringstream* может использоваться для преобразования строкового представления любого примитивного типа в соответствующее значение:

```
float f;
string s = "3.7";
istringstream is(s);
is >> f;
```

istringstream **<sstream>**

```
string str() const;
void str(const string & s);
```

Функция *str* предназначена для чтения строки, используемой потоком в качестве буфера, а также для ее установки.

ostringstream **<sstream>**

```
explicit ostringstream(openmode which = ios_base::out);
explicit ostringstream(const string & str,
                       openmode which = ios_base::out);
```

Эти конструкторы создают объект класса *ostringstream*. Вторая форма конструктора позволяет указать строку, используемую в качестве приемника выводимых данных.

ostringstream **<sstream>**

```
string str() const;
void str(const string & s);
```

Функция *str* предназначена для чтения строки, используемой потоком в качестве буфера, а также для ее установки.

Поток *ostringstream* может использоваться для преобразования значения примитивного типа в строковое представление:

```
float f = 3.7;
ostringstream os;
os << f;
string s = os.str();
```

Отметим, что перед выводом числа в строковый поток можно настроить любые свойства форматирования.

Литература

1. *Абрамян М. Э.* Язык программирования C++. Часть 1. Базовые типы и управляющие операторы. — Ростов н/Д., 2008. — 116 с.
2. *Абрамян М. Э.* Язык программирования C++. Часть 3. Динамические структуры данных. — Ростов н/Д., 2008. — 125 с.
3. *Абрамян М. Э.* 1000 задач по программированию. Часть II: Минимумы и максимумы, одномерные и двумерные массивы, символы и строки, двоичные файлы. — Ростов н/Д.: УПЛ РГУ, 2004. — 42 с.
4. *Абрамян М. Э.* 1000 задач по программированию. Часть III: Текстовые файлы, составные типы данных в процедурах и функциях, рекурсия, указатели и динамические структуры. — Ростов н/Д.: УПЛ РГУ, 2004. — 43 с.
5. *Вирт Н.* Алгоритмы и структуры данных. — М.: Мир, 1989. — 360 с.
6. *Михалкович С. С.* Основы программирования: Файлы. Рекурсия. — Ростов н/Д.: УПЛ ЮФУ, 2007. — 40 с.
7. *Сафонцев С. А.* Кредитно-модульная рейтинговая технология: Учебно-методическое пособие. — Ростов н/Д., 2008. — 16 с.

Содержание