Двоичные и текстовые файлы: решение задач

При выполнении заданий, связанных с обработкой файлов, будем использовать файловые потоки ввода-вывода, реализованные в стандартном заголовочном файле *fstream*, не применяя стандартные функции ввода-вывода языка C (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.

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, B_1, C_1, A_2, B_2, C_2, … .

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.

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*.

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 будет. 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.

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.

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 дей-

ствий 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.

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).

Текстовые файлы: решение задач

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 содержание 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*.

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 пропус-

кается, 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 недостатка, 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*.

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*. 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.

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

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.