Лабораторная работа 7

Обработка символьных строк

Задание.

Провести кодирование и декодирование текста (массива символов) при помощи кода Цезаря[1] с переменным сдвигом по таблице ASCII-кодов. Величина сдвига для каждой позиции в исходном тексте - сумма (по модулю 256) кодов символов слова кодового блокнота, стоящего в блокноте на той же позиции. Если кодовый блокнот имеет слов меньше, чем количество символов в исходном тексте, то по исчерпании слов в нём перейти к первому слову и продолжить. (На основе кодового блокнота целесообразно сначала сформировать по заданному правилу целочисленный массив ключей, который затем использовать при кодировании. Эти действия оформить в виде отдельной функции.)

Исследовать повторяемость символов в закодированном тексте (сколько каких кодов одного и того же исходного символа получено) в зависимости от кодового блокнота и длины исходного текста. Результаты исследования представить в виде таблицы (продумать формат таблицы). Исследование и вывод таблицы результатов следует выполнять в режиме диалога, последовательно вычисляя и выводя результаты для запрашиваемого символа. Статистические данные хранить в массиве int stat[256], Для большей достоверности статистических результатов в качестве исходного текста и кодового блокнота использовать текстовые файлы размером около 1 Кбайта.

Пример программы работы с символьными строками.

!!! Перед выполнением работы выполните приведенный ниже пример программы работы с символьными строками. Особое внимание обратите на используемую последовательность шагов разработки программы и на применение библиотечных функций. Затраты времени на изучение функций библиотеки языка программирования, имеющих отношение к решаемой задаче, окупятся при разработке программы.

Написать программу, которая определяет, сколько раз встретилось заданное слово в текстовом файле, длина строки в котором не превышает 80 символов. Текст не содержит переносов слов.

Определим слово как последовательность алфавитно-цифровых символов, после которых следует знак пунктуации, разделитель или признак конца строки. Слово может находиться либо в начале строки, либо после разделителя или знака пунктуации. Это можно записать следующим образом (фигурные скобки и вертикальная черта означают выбор из альтернатив):

```
слово = {начало строки | знак пунктуации | разделитель} символы, составляющие слово
        {конец строки | знак пунктуации | разделитель}
```

I. ***Исходные данные и результаты***

Исходные данные:

1. Текстовый файл неизвестного размера, состоящий из строк длиной не более 80 символов. Поскольку по условию переносы отсутствуют, можно ограничиться поиском слова в каждой строке отдельно. Для ее хранения выделим строку длиной 81 символ.
2. Слово для поиска, вводимое с клавиатуры. Для его хранения также выделим строку длиной 81 символ.

Результатом работы программы является количество вхождений слова в текст. Представим его в программе в виде целой переменной.

---

[1] Цезарь для шифрования своих посланий использовал следующий прием. При кодировании и декодировании писем он заменял буквы в письме на следующие по алфавиту буквы с постоянным смещением, равным 13, то есть к порядковому номеру буквы в латинском алфавите, содержащим 26 букв, он прибавлял 13 и получал порядковый номер буквы в кодируемом (декодируемом) письме. Если полученный таким образом номер буквы был больше 26, то он уменьшался на 26.

Для хранения длины строки будем использовать именованную константу, а для хранения фактического количества символов в слове — переменную целого типа. Для работы с файлом потребуется служебная переменная соответствующего типа.

II. ***Алгоритм решения задачи***

- Построчно считывать текст из файла.
- Просматривая каждую строку, искать в ней заданное слово. При каждом нахождении слова увеличивать счетчик.

Детализируем второй пункт алгоритма. Очевидно, что слово может встречаться в строке многократно, поэтому для поиска следует организовать цикл просмотра строки, который будет работать, пока происходит обнаружение в строке последовательности символов, составляющих слово.

При обнаружении совпадения с символами, составляющими слово, требуется определить, является ли оно отдельным словом, а не частью другого. (Это один из возможных вариантов решения задачи, не самый лучший. Другой вариант – сначала выделить слово, а затем сравнивать его с заданным). Например, мы задали слово «кот». Эта последовательность символов содержится, например, в словах «котенок», «трикотаж», «трескотня» и «апперкот». Следовательно, требуется проверить символ, стоящий после слова, а в случае, когда слово не находится в начале строки — еще и символ перед словом. Эти символы проверяются на принадлежность множеству знаков пунктуации и разделителей.

III. ***Программа и тестовые примеры***

Разобьем написание программы на последовательность шагов.

Шаг 1. Ввести «скелет» программы (директивы #include, функцию main(), описание переменных, открытие файла). Добавить контрольный вывод введенного слова. Запустив программу, проверить ввод слова и успешность открытия файла. Для проверки вывода сообщения об ошибке следует выполнить программу еще раз, задав имя несуществующего файла.

```
#include <fstream.h>
int main()
{
const int len = 81;
char word[len],line[len];
cout << " Input the word for search:  ";
cin >> word;
ifstream fin("text.txt",ios::in);
if (!fin) {
   cout << "Error of file opening."<< endl;
   return 1; }
return 0;
}
```

Шаг 2. Добавить в программу цикл чтения из файла, внутри цикла поставить контрольный вывод считанной строки:

```
#include <fstream.h>
int main()
{
const int len = 81;
char word[len], line[len];
cout << "Input the word for search: ";
cin >> word;
ifstream fin("text.txt", ios::in);
```

```
if (!fin) {
   cout << "Error of file opening."<< endl;
   return 1;}
while (fin.getline(line, len))
   cout << line << endl;
return 0;
}
```

Шаг 3. Добавить в программу цикл поиска последовательности символов, составляющих слово, с контрольным выводом:

```
#include <fstream.h>
#include <string.h>
int main()
{
const int len = 81;
char word[len], line[len];
cout << "Input the word for search: ";
cin >>word;
int l_word = strlen(word);
ifstream fin("text.txt", ios::in);
if (!fin) {
   cout << "Error of file opening."<< endl;
   return 1; }
int count =0;
while (fin.getline(line, len)) {
   char *p = line;
   while( p = strstr(p, word)) {
       cout << "coincidence: " << p << endl;
       p += l_word;
       count++;
   }
}
cout << count << endl;
return 0;
}
```

Для многократного поиска вхождения подстроки в заголовке цикла используется функция strstr. Очередной поиск должен выполняться с позиции, следующей за найденной на предыдущем проходе подстрокой. Для хранения этой позиции определяется вспомогательный указатель p, который на каждой итерации цикла наращивается на длину подстроки. Также вводится счетчик количества совпадений. На данном этапе он считает не количество слов, а количество вхождений последовательности символов, составляющих слово.

Шаг 4. Добавить в программу анализ принадлежности символов, находящихся перед словом и после него, множеству знаков пунктуации и разделителей:

```
#include <fstream.h>
#include <string.h>
#include <ctype.h>
int main()
{
const int len = 81;
```

```
char word[len], line[len];
cout << "Input the word for search: ";
cin >>word;
int l_word = strlen(word);
ifstream fin("text.txt", ios::in);
if (!fin) {
    cout << "Error of file opening."<< endl;
    return 1; }
int count =0;
while (fin.getline(line, len)) {
    char *p = line;
    while( p = strstr(p, word)) {
        char *c = p;     //c-начало подстроки совпадения
        p += l_word;     //p-конец подстроки совпадения
         // подстрока не в начале строки?
        if (c != line)
         // символ перед подстрокой совпадения не разделитель?
           if ( !ispunct(*(c - 1) ) && !isspace(*(c - 1))) continue;
         // символ после слова разделитель?
         if ( ispunct(*p) || isspace(*p) || (*p == '\0') ) count++;
         //подстрока – отдельное слово
    }
}
cout << "Number of entering word: "<< count << endl;
return 0;
}
```

Здесь вводится служебная переменная ***c*** для хранения адреса начала вхождения подстроки. Символы, ограничивающие слово, проверяются с помощью функций ispunct и isspace, прототипы которых хранятся в заголовочном файле <ctype.h>. Символ, стоящий после слова, проверяется также на признак конца строки (для случая, когда искомое слово находится в конце строки).

Для тестирования программы требуется создать файл с текстом, в котором заданное слово встречается:

- в начале строки;
- в конце строки;
- в середине строки;
- несколько раз в одной строке;
- как часть других слов, находящаяся в начале, середине и конце этих слов;
- в скобках, кавычках и других разделителях.

Длина хотя бы одной из строк должна быть равна 80 символам. Для тестирования программы следует выполнить ее по крайней мере два раза: введя с клавиатуры слово, содержащееся в файле, и слово, которого в нем нет.

Давайте теперь рассмотрим другой вариант решения этой задачи. В библиотеке есть функция strtok, которая разбивает переданную ей строку на лексемы в соответствии с заданным набором разделителей. Если мы воспользуемся этой функцией, нам не придется «вручную» выделять и проверять начало и конец слова, потребуется лишь сравнить с искомым словом слово, выделенное с помощью strtok. Правда, список разделителей придется задать вручную:

```
#include <fstream.h>
```

```
#include <string.h>
int main()
{
const int len = 81;
char word[len], line[len];
char *delims = ".,!? /<>|)(*::\"";
cout << "Input the word for search: ";  cin >> word;
ifstream fin("text.txt", ios::in);
if (!fin) { cout << "Error of file opening."<< endl; return 1; }
char *token;
int count = 0;
while (fin.getline(line, len)) {
    token = strtok( line, delims );           // 1
    while( token != NULL ) {
       if ( !strcmp (token, word) )  count++;      // 2
       token = strtok( NULL, delims );             // 3
   }
}
cout << "Number of entering word: "<<count << endl;
return 0;
}
```

Первый вызов функции strtok в операторе 1 формирует адрес первой лексемы (слова) строки line. Он сохраняется в переменной token. Функция strtok заменяет на NULL разделитель, находящийся после найденного слова, поэтому в операторе 2 можно сравнить на равенство искомое и выделенное слово. В операторе 3 выполняется поиск следующей лексемы в той же строке. Для этого следует задать в функции strtok в качестве первого параметра NULL (так запрограммирована функция).

Как видите, программа стала короче и яснее. На этом примере можно видеть, что средства, предоставляемые языком, влияют на алгоритм решения задачи, и поэтому перед тем, как продумывать алгоритм, необходимо эти средства изучить. Представьте, во что бы вылилась программа без использования функций работы со строками и символами!

Работа с файлами

Информация во внешней памяти (на диске, на магнитных лентах и т.п.) сохраняется в виде файлов - именованных объектов, доступ к которым обеспечивает операционная система ЭВМ. Основное отличие внешней памяти ЭВМ от основной (иначе оперативной) памяти - возможность сохранения информации при отключении ЭВМ. Поддержка операционной системы состоит в том, что в ней имеются средства:

- создания файлов;
- уничтожения файлов;
- поиска файлов на внешнем носителе информации (на диске);
- чтения и записи данных из файлов и в файлы;
- открытия файлов;
- закрытия файлов;
- позиционирования файлов.

Библиотека ввода-вывода Си++ включает средства для работы с последовательными файлами. Логически последовательный файл можно представить как именованную цепочку (ленту, строку) байтов, имеющую начало и конец. Последовательный файл отличается от файлов с другой организацией тем простым свойством, что чтение (или

запись) из файла (в файл) ведутся байт за байтом от начала к концу. В каждый момент позиции в файле, откуда выполняется чтение и куда производится запись, определяются значениями указателей позиций записи и чтения файла. Позиционирование указателей записи и чтения (т.е. установка на нужные байты) выполняется либо автоматически, либо за счет явного управления их положением. В стандартной библиотеке ввода-вывода Си++ имеются соответствующие средства.

Взаимосвязь файлов с потоками ввода-вывода осуществляется с помощью следующих действий:

1 - создание файла;

2 - создание потока;

3 - открытие файла;

4 - "присоединение" файла к потоку;

5 - обмены с файлом с помощью потока;

6 - "отсоединение" потока от файла;

7 - закрытие файла;

8 - уничтожение файла.

Все перечисленные действия могут быть выполнены с помощью средств библиотеки классов ввода-вывода языка Си++. Однако существует несколько альтернативных вариантов их выполнения.

Потоки для работы с файлами создаются как объекты следующих классов:

- ofstream - для вывода (записи) данных в файл;
- ifstream - для ввода (чтения) данных из файла;
- fstream -для чтения и для записи данных (двунаправленный обмен).

Чтобы использовать эти классы, в текст программы необходимо включить дополнительный заголовочный файл fstream.h. После этого в программе можно определять конкретные файловые потоки, соответствующих типов (объекты классов ofstream, ifstream, fstream), например, таким образом:

ofstream outFile; // Определяется выходной файловый поток

ifstream inFile; // Определяется входной файловый поток

f**stream** ioFile; // Определяется файловый поток для ввода и вывода

Создание файлового потока (объекта соответствующего класса) связывает имя потока с выделяемым для него буфером и инициализирует переменные состояния потока. Так как перечисленные классы файловых потоков наследуют свойства класса ios, то и переменные состояния каждого файлового потока наследуются из этого базового класса. Так как файловые классы являются производными от классов ostream (класс **ofstream),** istream (класс ifstream), stream (класс **fstream),** то они поддерживают описанный ниже форматированный и бесформатный обмен с файлами для этих классов. Однако прежде чем выполнить обмен, необходимо открыть соответствующий файл и связать его с файловым потоком.

Открытие файла в самом общем смысле означает процедуру, информирующую систему о тех действиях, которые предполагается выполнять с файлом. Для работы с файловыми потоками библиотеки ввода-вывода языка Си++ удобно пользоваться компонентными функциями (методами) соответствующих классов.

Создав файловый поток, можно "присоединить" его к конкретному файлу с помощью компонентной функции **open**(). Функция **open**() унаследована каждым из файловых классов **ofstream, ifsream, fstream** от класса f**streambase**. С ее помощью можно не только открыть файл, но и связать его с уже определенным потоком. Формат функции:

```
void open(const char *fileName, int mode = умалчиваемое_значение,
int protection = умалчиваемое_значение);
```

Первый параметр - fileName - имя уже существующего или создаваемого заново файла. Это строка, определяющая полное или сокращенное имя файла в формате, регламентированном операционной системой. Второй параметр - mode *(режим)* - дизъюнкция флагов, определяющих режим работы с открываемым файлом (например, только запись или только чтение). Флаги определены следующим образом:

```
enum ios::open_mode {
in  = 0x01,          // Открыть только для чтения
out = 0x02,          // Открыть только для записи
app = 0x08,          // Дописывать данные в конец файла
trunc = 0x10,        // Вместо существующего создать новый файл
binary = 0x80,       // Открыть для двоичного (не текстового) обмена
};
```

Назначения флагов поясняют комментарии, однако надеяться, что именно такое действие на поток будет оказывать тот или иной флаг в конкретной реализации библиотеки ввода-вывода, нельзя. Как пишет автор языка Си++ [26], "смысл значений open_mode скорее всего зависит от реализации". Умалчиваемое значение параметра mode зависит от типа потока, для которого вызывается функция open ().

Третий параметр - protection *(защита)* - определяет защиту и достаточно редко используется. Точнее, он устанавливается по умолчанию и умалчиваемое значение обычно устраивает программиста.

Как обычно вызов функции open () осуществляется с помощью уточненного имени

имя_объекта_класса.вызов_принадлежащей_классу_функции

Итак, открытие и присоединение файла к конкретному файловому потоку обеспечивается таким вызовом функции open ():

имя потока.open(имя_файла, режим, защита);

Здесь имя_потока - имя одного из объектов, принадлежащих классам ofstream, ifstream, fstream. Примеры вызовов для определенных выше потоков:

```
outFile.open("C:\\USER\\RESULT.DAT");
inFile.open("DATA.TXT");
ioFile.open("CHANGE.DAT",ios::out);
```

При открытии файлов с потоками класса **ofstream** второй параметр по умолчанию устанавливается равным **ios:: out**, т.е. файл открывается только для вывода. Таким образом, файл **c: \user\resalt.dat** после удачного выполнения функции **open ()** будет при необходимости (если он не существовал ранее) создан, а затем открыт для вывода (записи) данных в текстовом режиме обмена и присоединен к потоку **outFile**. Теперь к потоку **outFile** может применяться, например, операция включения <<, как к стандартным выходным потокам **cout, cerr**.

Поток **inFile** класса **ifstream** в нашем примере присоединяется функцией **open()** к файлу с именем **data.txt**. Этот файл открывается для чтения из него данных в текстовом режиме. Если файла с именем **data.txt** не существует, то попытка вызвать функцию **inFile.open ()** приведет к ошибке.

Для проверки удачности завершения функции open () используется перегруженная операция **!**. Если унарная операция ! применяется к потоку, то результат ненулевой при наличии ошибок. Если ошибок не было, то выражение **!имя_потока** имеет нулевое значение. Таким образом, можно проверить результат выполнения функции **open():**

```
if   (!inFile)
{ cerr << "Ошибка при открытии файла!\n"; exit(1); }
```

Для потоков класса **fstream** второй аргумент функции **open()** должен быть задан явно, так как по умолчанию неясно, в каком направлении предполагается в ыполнять обмен с

потоком. В примере файл **change.dat** открывается для записи и связывается с потоком **ioFile**, который будет выходным потоком до тех пор, пока с помощью повторного открытия файла явно не изменится направление обмена с файлом или потоком.

В классах **ifstream, ofstream, fstream** определены конструкторы, позволяющие по-иному выполнять создание и открытие файлов. Типы конструкторов для потоков разных классов очень похожи:

имя_класса() ;

создает поток, не присоединяя его ни к какому файлу;

имя_класса(int fd);

создает поток и присоединяет его к уже открытому файлу, дескриптор которого используется в качестве параметра **fd**;

имя_класса(int fd, char *buf, int);

создает поток, присоединяя его к уже открытому файлу с дескриптором **fd**, и использует явно заданный буфер (параметр buf);

имя_класса(char *FileName, int mode, int = ...);

создает поток, присоединяет его к файлу с заданным именем Filename, а при необходимости предварительно создает файл с таким именем.

Детали и особенности перечисленных конструкторов лучше изучать по документации конкретной библиотеки ввода-вывода.

Работая со средствами библиотечных классов ввода-вывода, чаще всего употребляют конструктор без параметров и конструктор, в котором явно задано имя файла. Примеры обращений к конструкторам без параметров:

ifstream fi; //Создает входной файловый поток fi

ostream fo; // Создает выходной файловый поток fo

fstream ff; // Создает файловый поток ввода-вывода ff

После выполнения каждого из этих конструкторов файловый поток можно присоединить к конкретному файлу, используя уже упомянутую компонентную функцию **open ()** :

void open (char*FileName, int режим, int защита);

Примеры:

```
fi.open ("Filel.txt",ios::in);      //Поток fi соединен с файлом Filel.txt
fi.close();                         //Разорвана связь потока fi с файлом Filel.txt
fi.open("File2.txt");               //Поток fi присоединен к файлу File2.txt
fo.open("NewFile");                 //Поток fo присоединяется к файлу NewFile; если такой
                                    //файл отсутствует - он будет создан
```

При обращении к конструктору с явным указанием в параметре имени файла остальные параметры можно не указывать, они выбираются по умолчанию.

Примеры:

```
ifstream flowl ("File.l");
```

создает входной файловый поток с именем **flowl** для чтения данных. Разыскивается файл с названием **File.l**. Если такой файл не существует, то конструктор завершает работу аварийно. Проверка:

```
if (!flowl) cerr << "Не открыт файл File.l!";
ofstream flow2 ("File.2");
```

создается выходной файловый поток с именем **flow2** для записи информации. Если файл с названием **File.2** не существует, он будет создан, открыт и соединен с потоком **f low2.** Если файл уже существует, то предыдущий вариант будет удален и пустой файл создается заново. Проверка:

```
if (!flow2) cerr << "Не открыт файл File.2!";
fstream flow3("File.3");
```

создается файловый поток **fiow3**, открывается файл **File.3** и присоединяется к потоку **flow3**.

Все файловые классы унаследовали от базовых классов функцию **close** (), позволяющую очистить буфер потока, отсоединить поток от файла и закрыть файл. Функцию **close** () необходимо явно вызывать при изменении режимов работы с файловым потоком. Автоматически эта функция вызывается только при завершении программы.

В качестве иллюстрации основных особенностей работы с файлами рассмотрим несколько программ.

//Чтение текстового файла с помощью операции >>

```
#include <stdlib.h>       // Для функции exit()
#include <fstream.h>      // Для файловых потоков
const int lenName = 13;   // max длина имени файла
const int lenString = 60; // Длина вспомогательного массива
void main()
{
char source[lenName];     // Массив для имени файла
cout << "\nВведите имя исходного файла: ";
cin >> source;
ifstream inFile;          // Входной файловый поток
// Открыть файл source и связать его с потоком inFile:
inFile.open(source);
if (!inFile)     // Проверить правильность открытия файла
   { cerr « "\nОшибка при открытии файла " << source;
   exit(1);    // Завершение программы
   }
// Вспомогательный массив для чтения:
char string[lenString];
char next;
cout << "\n Текст файла:\n\n";
cin.get();       // Убирает код из потока cin
while(1)         // Неограниченный цикл
{ // Ввод из файла одного слова до пробельного символа либо EOF:
inFile >> string;
// Проверка следующего символа:
next = inFile.peek();
// Выход при достижении конца файла:
if (next == EOF) break;
// Печать с добавлением разделительного пробела:
cout << string <<” “;
if (next == '\n')        // Обработка конца строки
   { cout << '\n';
// 4 - смещение для первой страницы экрана:
static int i = 4;
// Деление по страницам до 20 строк каждая:
if (!(++i % 20))
```

```
    { cout << "\nДля продолжения вывода нажмите ENTER.\n" << endl;
    cin.get();
    }
   }
}
```

Результат выполнения программы - постраничный вывод на экран текстового файла, имя которого набирает на клавиатуре пользователь по "запросу" программы. Размер страницы - 20 строк. В начале первой страницы - результат диалога с пользователем и поэтому из файла читаются и выводятся только первые 16 строк.

Программа демонстрирует неудобства чтения текста из файла с помощью операции извлечения >>, которая реагирует на каждый обобщенный пробельный символ. Между словами, прочитанными из файла, принудительно добавлено по одному пробелу. А сколько их (пробелов) было в исходном тексте, уже не известно. Тем самым искажается содержащийся в файле текст. Читать пробельные символы позволяет компонентная функция **getline()** класса **istream**, наследуемая классом **ifstream**. Текст из файла будет читаться и выводиться на экран (в поток **cout**) без искажений (без пропусков пробелов), если в предыдущей программе чтение и вывод в поток **cout** организовать таким образом:

```
while(1)      // Неограниченный цикл
{
    inFile.getline(string,lenString);
    next = inFile.peek();
    if (next == EOF) break;
    cout << string;
}
```

Потоки ввода-вывода. В соответствии с названием заголовочного файла **iostream.h (stream** - поток; "i" - сокращение от *input - ввод* "o" - сокращение от *output* - вывод) описанные в этом файле средства ввода-вывода обеспечивают программиста механизмами для извлечение данных из потоков и для включения (внесения) данных в потоки. Поток определяется как последовательность байтов (символов) и с точки зрения программы не зависит от тех конкретных устройств (файл на диске, принтер, клавиатура, дисплей, стример и т.п.), с которыми ведется обмен данными. При обмене с потоком часто используется вспомогательный участок основной памяти - буфер потока (рис.1- буфер вывода, рис. **2** - буфер ввода).

Рис. *1. Буферизированный выходной поток*

В буфер потока помещаются выводимые программой данные перед тем, как они будут переданы к внешнему устройству. При вводе данных они вначале помещаются в буфер и только затем передаются в область памяти выполняемой программы. Использование буфера как промежуточной ступени при обменах с внешними устройствами повышает скорость передачи данных, так как реальные пересылки осуществляются только тогда, когда буфер уже заполнен (при выводе) или пуст (при вводе).

Работу, связанную с заполнением и очисткой буферов ввода-вывода, операционная система очень часто берет на себя и выполняет без явного участия программиста. Поэтому поток в прикладной программе обычно можно рассматривать просто как последовательность байтов. При этом очень важно, что никакой связи значений этих байтов с кодами какого-либо алфавита не предусматривается. Задача программиста при вводе-выводе с помощью потоков - установить соответствие между участвующими в обмене типизированными объектами и последовательностью байтов потока, в которой отсутствуют всякие сведения о типах представляемой (передаваемой) информации.

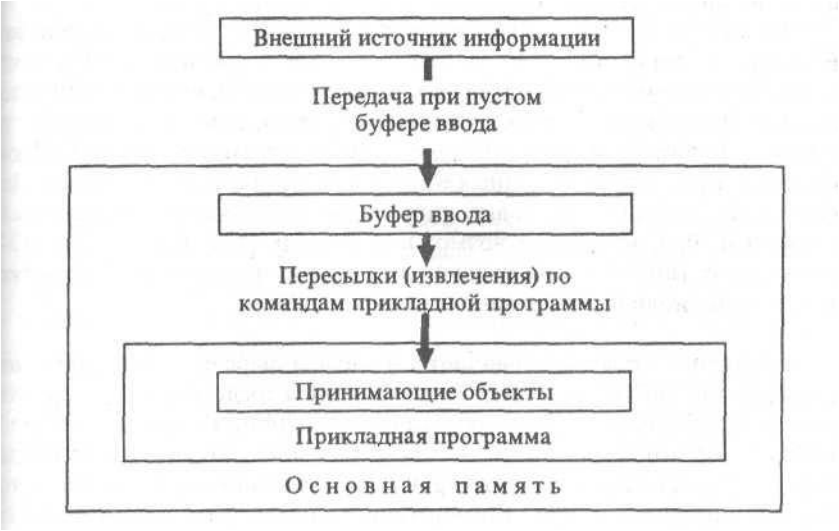

Рис .2*. Буферизированный входной поток*

Используемые в программах потоки логически делятся на три типа:

- входные, из которых читается информация;
- выходные, в которые вводятся данные;
- двунаправленные, допускающие как чтение, так и запись.

Все потоки библиотеки ввода-вывода последовательные, т.е. в каждый момент для потока определены позиции записи и (или) чтения, и эти позиции после обмена перемещаются по потоку на длину переданной порции данных.

Функции для обмена с потоками

Кроме операции включения (записи) в поток << и извлечения (чтения) из потока >>, в классах библиотеки ввода-вывода есть весьма полезные функции, обеспечивающие программиста альтернативными средствами для обмена с потоками.

При выводе в качестве основного класса, формирующего выходные потоки, используется класс **ostream.** В нем определены (ему принадлежат) две функции для двоичного вывода данных:

ostream& ostream::put(char cc);

ostream& ostream::write(const signed char *array, int n);

ostream& ostream::write(const unsigned char *array, int n) ;

Функция **put ()** помещает в тот выходной поток, для которого она вызвана, символ, использованный в качестве фактического параметра.

В этом случае эквивалентны операторы:

cout << 'Z';

и

cout.put('Z');

Функция **write ()** имеет два параметра - указатель **array** на участок памяти, из которого выполняется вывод, и целое значение **n**, определяющее количество выводимых из этого участка символов (байт).

В отличие от операции << включения в поток функции **put ()** и **write** () не обеспечивают форматирования выводимых данных. Например, если при выводе одного символа с помощью операции << можно, используя функцию **width (),** разместить его в поле из нужного количества позиций, то функция **put()** всегда разместит символ в одной позиции выходного потока. Флаги форматирования также не применимы к функциям **put()** и **write().**

Так как функции **put()** и **write**() возвращают ссылки на объект того класса, для

которого они выполняются, то можно организовать цепочку вызовов:

```
char  ss[]  =  "Merci";
cout.put('\n').write(ss,sizeof(ss)-l).put('!').put('\n');
```

На экране (в потоке **cout**) появится:

Merci!

Если необходимо прочитать из входного потока строку символов, содержащую пробелы, то с помощью операции извлечения >> это делать неудобно - каждое чтение строки выполняется до пробела, а ведущие (левые) пробельные символы игнорируются. Если мы хотим, набрав на клавиатуре строку: "Qui vivra verra - будущее покажет (лат.) ", ввести ее в символьный массив, то с помощью операции извлечения >> это сделать несколько хлопотно, все слова будут читаться отдельно (до пробела). Гораздо удобнее воспользоваться функциями бесформатного (двоичного) чтения.

Функции двоичного (бесформатного) чтения данных принадлежат потоку **istream**. Прежде чем перечислить их, отметим основное свойство двоичного чтения данных. Данные читаются без преобразования их из двоичного представления в текстовое. Например, если во входном потоке размещено представление вещественного числа

1.3e-3, то это будет воспринято как последовательность из шести байт, и читать эту последовательность с помощью функций двоичного ввода можно только в символьный массив.

Итак, функции чтения.

Во-первых, это 6 перегруженных функций get (). Две из них имеют следующие прототипы:

istream& get(signed char *array, int max_len, char ='\n');

istream& get(unsigned char *array, int max_len, char ='\n');

Каждая из этих функций выполняет извлечение (чтение) последовательности байтов из стандартного входного потока и перенос их в символьный массив, задаваемый первым параметром. Второй параметр определяет максимально допустимое количество прочитанных байтов. Третий параметр определяет ограничивающий символ (байт), при появлении которого во входном потоке следует завершить чтение. По умолчанию третий параметр имеет значение '\n' - переход на следующую строку, однако при обращении к функции его можно задавать и по-другому. Значение этого третьего параметра из входного потока не удаляется, он в формируемую строку (символьный массив) не переносится, а вместо него автоматически добавляется "концевой" символ строки ' \0'. Если из входного потока извлечены ровно **max_len - 1** символов, однако ограничивающий символ (например, по умолчанию '\n') не встретился, то концевой символ помещается после введенных символов. Массив, в который выполняется чтение, должен иметь длину не менее **max_len** символов. Если из входного потока не извлечено ни одного символа, то устанавливается код ошибки. Если до появления ограничивающего символа и до извлечения **max_len - 1** символов встретился конец файла **EOF**, то чтение прекращается как при появлении ограничивающего символа.

Функция с прототипом

```
istream& get(streambuf& buf,   char =   '\n');
```

извлекает из входного потока символы и помещает их в буфер, определенный первым параметром. Чтение продолжается до появления ограничивающего символа, которым по умолчанию является **'\n'**, но он может быть установлен явно любым образом.

Три следующих варианта функции **get()** позволяют прочесть из входного потока один символ. Функции

```
istream&  get(unsigned char&  cc) ;
```

istream& get(signed char& cc) ;

присваивают извлеченный символ фактическому параметру и возвращают ссылку на поток, из которого выполнено чтение. Функция

int get() ;

получает код извлеченного из потока символа в качестве возвращаемого значения. Если поток пуст, то возвращается код конца файла **EOF**.

Функции "ввода строк":

istream& getline(signed char *array, int len, char= '\n');

istream& getline(unsigned char *array, int len, char= '\n');

подобны функциям **get ()** с теми же сигнатурами, но переносят из входного потока и символ-ограничитель. Функция

int peek () ;

позволяет "взглянуть" на очередной символ входного потока. Точнее, она возвращает код следующего символа потока (или **EOF**, если поток пуст), но оставляет этот символ во входном потоке. При необходимости этот символ можно в дальнейшем извлечь из потока с помощью других средств библиотеки. Например, следующий цикл работает до конца строки (до сигнала от клавиши **Enter):**

```
char cim;
while (cin.peek() != '\n')
{ cin.get(cim);
cout.put(cim) ; }
```

Принадлежащая классу **istream** функция

istream& putback (char cc);

не извлекает ничего из потока, а помещает в него символ **cc,** который становится текущим и будет следующим извлекаемым из потока символом.

Аналогичным образом функция

int gcount ();

подсчитывает количество символов, которые были извлечены из входного потока при последнем обращении к нему. Функция

istream& ignore(int n=l, int EOF);

позволяет извлечь из потока и "опустить" то количество символов **n**, которое определяется первым параметром. Второй параметр определяет символ-ограничитель, при появлении которого выполнение функции нужно прекратить, даже если из потока еще не извлечены все **n** символов. Функции

istream& read(signed char *array, int numb);

istream& read(unsigned char *array, int numb);

выполняют чтение заданного количества **numb** символов в массив **array.**

Полезны следующие функции того же класса **istream**:

istream& seekg(long pos) ;

устанавливает позицию чтения из потока в положение, определяемое значением параметра.

istream& seekg(long pos, seek_dir dir);

выполняет перемещение позиции чтения вдоль потока в направлении, определенном параметром **dir**, принимающим значения из перечисления **enum seek_dir {beg, cur, end }.** Относительная величина перемещения (в байтах) определяется значением параметра **long pos**. Если направление определено как **beg**, то смещение от начала потока; **cur** - от

текущей позиции; **end** - от конца потока;

long tellg()

определяет текущую позицию чтения из потока.

Подобные перечисленным функции класса **ostream**:

long tellp()

определяет текущую позицию записи в поток:

ostream& seekp (long pos, seek_dir dir)

аналогична функции **seekg ()** , но принадлежит классу **ostream** и выполняет относительное перемещение позиции записи в поток;

ostream& seekp(long pos);

устанавливает абсолютную позицию записи в поток.

Использование аргументов командной строки

Данные в функцию main() можно передавать через параметры из командной строки (строки запуска функции на выполнение из операционной системы). Это удобно при отладке программы (чтобы не вводить при каждом запуске одни и те же данные) и может быть удобным при использовании программы.

При использовании функции main() с параметрами в командной строке заголовок функции имеет вид:

int main(int argc, char* argv[]), где:

argc – количество параметров;

argv[] – массив указателей на параметры (каждый параметр – строка).

Первый параметр, argv[0] – указатель на полное имя исполняемого файла. Он передается по умолчанию и не указывается при запуске программы.

Для выполнения программы с аргументами в командной строке нужно уметь:

- запустить программу с параметрами в командной строке;
- проконтролировать из программы количество параметров;
- использовать параметры из командной строки в тексте программы.

Запуск программы с параметрами в командной строке можно осуществить непосредственно из командной строки ДОС или из среды программирования Microsoft Visual C++.

Для запуска из командной строки ДОС нужно выполнить следующие действия:

- на Рабочем столе Windows нажать кнопку ***ПУСК;***
- в раскрывшемся меню выбрать пункт ***ВЫПОЛНИТЬ;***
- с помощью кнопки ***ОБЗОР***… найти нужный файл с расширением .exe;
- в окне **Открыть:** после символа “, стоящего в конце полного имени файла, через пробел ввести необходимые параметры.

Для запуска из Microsoft Visual C++ нужно:

- открыть меню **Project;**
- выбрать пункт **Settings…**;
- в окне **Program settings** выбрать закладку **Debug**;
- в поле **Program arguments** через пробел ввести необходимые параметры.

При контроле числа параметров нужно иметь ввиду, что первый параметр всегда передается по умолчанию и число параметров будет на 1 больше, чем введено. Например, если передается один параметр - имя файла с исходными данными, то argc будет равно 2, argv[0] указывает на полное имя исполняемого модуля (точнее, на список параметров системного окружения, первым из которых является имя исполняемого файла), а argv[1] указывает на введенное имя файла с данными.

При использовании параметров нужно помнить, что каждый передаваемый параметр – это символьная строка. Если надо передать численное значение, то в программе надо преобразовать строку в число с помощью соответствующей типу числа функции (doubl atof(char *str), int atoi(char *str), или long atol(char *str)).

В качестве примера работы с файловыми потоками приведем программу копирования одного файла в другой. Имена файлов берутся из аргументов командной строки:

```
// MyCopy.cpp
#include <iostream>
#include <fstream>
#include "PrintCyr.h"
using namespace std;
int main(int argc, char* argv[])
{
if (argc != 3) cout <<"Неверное число аргументов"<<endl;
ifstream from(argv[1]); // открываем входной файл
if (!from)
{
  cout<<"Входной файл "<< argv[1 ]<<" не найден"<<endl;
return 1;
}
ofstream to(argv[2]); // открываем выходной файл
if (!to)
{
cout<<"Выходной файл "<< argv[2]<< " не открыт"<< endl;
return1;
}
char ch;
while (from.get(ch))
 {
to.put(ch);
if (!to) cout<<"Ошибка записи (диск переполнен).";
}
cout << "Копирование из " << argv[1] << " в " << argv[2] << " завершено." << endl;
return 0;
 }
```

Ввод-вывод строк

Для ввода-вывода строк используются как уже известные нам объекты cin и cout, так и функции, унаследованные из библиотеки C. Рассмотрим сначала первый способ:

```
#include <iostream.h>
int main()
{
 const int n = 80;
char s[n];
cin >> s; cout << s << endl;
return ();
}
```

Как видите, строка вводится точно так же, как и переменные известных нам типов. Запустите программу и введите строку, состоящую из одного слова. Запустите программу повторно и введите строку из нескольких слов. Во втором случае выводится только первое слово. Это связано с тем, что ввод выполняется до первого пробельного символа (то есть пробела, знака табуляции или символа перевода строки '\n')[2]. Можно ввести слова входной строки в отдельные строковые переменные:

```
#include <iostream.h>
 int main()
{
  const int n = 80;
  char s[n], t[n], r[n];
  cin >>s >>t >> r; cout <<s << endl << t << endl << r << endl;
  return 0;
}
```

Если требуется ввести строку, состоящую из нескольких слов, в одну строковую переменную, используются *методы* getline или get класса istream, объектом которого является cin. Во втором семестре мы изучим, что такое методы класса[3], а пока можно пользоваться ими как волшебным заклинанием, не вдумываясь в смысл. Единственное, что нам пока нужно знать, это синтаксис вызова метода — после имени объекта ставится точка, а затем пишется имя метода:

```
#include <iostream.h>
int main()
{
const int n = 80;
char s[n];
cin.getline(s. n);
 cout << s << endl;
cin.get(s, n);
cout << s << endl;
return 0;
}
```

Метод getline считывает из входного потока n - 1 символов или менее (если символ перевода строки встретится раньше) и записывает их в строковую переменную s. Символ перевода строки[4] также считывается (удаляется) из входного потока, но не записывается в строковую переменную, вместо него размещается завершающий 0. Если в строке исходных данных более n-1 символов, следующий ввод будет выполняться из той же строки, начиная с первого несчитанного символа. Метод get с двумя аргументами работает аналогично, но оставляет в потоке символ перевода строки. В строковую переменную добавляется завершающий 0.

Никогда не обращайтесь к разновидности метода get с двумя аргументами два раза

---

[2] Если во вводимой строке больше символов, чем может вместить выделенная для ее хранения область, поведение программы не определено. Скорее всего, она завершится аварийно

[3] Синонимом термина «метод» является «компонентная функция».

[4] Символ перевода строки '\n' появляется во входном потоке, когда вы нажимаете клавишу Enter

подряд, не удалив \n из входного потока. Например:

```
cin.get(s, n);                          // 1- считывание строки
cout << s << endl;                      // 2- вывод строки
cin.get(s, n);                          // 3- считывание строки
cout << s << endl;                      // 4- вывод строки
cin.get(s, n);                          // 5 - считывание строки
cout <<s << endl;                       // 6- вывод строки
cout << "Конец - делу венец" <<endl;  // 7
```

При выполнении этого фрагмента вы увидите на экране первую строку, выведенную оператором 2, а затем завершающее сообщение, выведенное оператором 7. Какие бы прекрасные строки вы ни ввели с клавиатуры в надежде, что они будут прочитаны операторами 3 и 5, метод get в данном случае «уткнется» в символ \n, оставленный во входном потоке от первого вызова этого метода (оператор 1). В результате будут считаны и, соответственно, выведены на экран пустые строки (строки, содержащие 0 символов). А символ \n так и останется «торчать» во входном потоке. Возможное решение этой проблемы — удалить символ \n из входного потока путем вызова метода get без параметров, то есть после операторов 1 и 3 нужно вставить вызов cin.get().

Однако есть и более простое решение — использовать в таких случаях метод getline, который после прочтения строки не оставляет во входном потоке символ \n.

Если в программе требуется ввести несколько строк, метод getline удобно использовать в заголовке цикла, например:

```
#include <iostream.h>
 int main()
{
 const int n = 80;
char s[n];
  while (cin.getline(s, n))
  {
  cout << s << endl;
  …                          // обработка строки
  }
  return 0;
  }
```