

# ЛАБОРАТОРНАЯ РАБОТА №4 ПО ПРЕДМЕТУ «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ И ПРОГРАММИРОВАНИЕ»

## ТЕМА «ОРГАНИЗАЦИЯ РАБОТЫ С ФАЙЛАМИ»

**ЦЕЛЬ РАБОТЫ:** изучить особенности работы с файлами как источниками и хранилищами данных на языке C++.

### ФАЙЛОВЫЙ ВВОД-ВЫВОД

Файлы — источник и хранилище данных, используемых компьютером. Без использования файлов все результаты работы компьютера уничтожались бы при его перезагрузке или при закрытии приложения. Язык C++ позволяет выполнять чтение/запись из/в файлов(ы). Операции над файлами выполняются с использованием **файлового ввода-вывода**.

Файловый поток представляет собой канал для работы с данными, находящимися в файле. Обработка файловых данных состоит в том, что происходит преобразование содержимого файла в поток информации, который можно использовать в приложении, точно также как данные, полученные от пользователя с клавиатуры.

Работа с дисковыми файлами требует использования классов *basic\_ifstream* для реализации ввода данных из файла в переменные и объекты программы, *basic\_ofstream* для реализации вывода данных в файл из переменных и объектов программы, *basic\_fstream* для реализации ввода/вывода данных из/в файла (файл). Объекты этих классов могут быть ассоциированы с дисковыми файлами, а их методы — использоваться для обмена данными.

На рисунке 1 видно, что класс *basic\_ifstream* является наследником класса *basic\_istream*, *basic\_ofstream* — наследником класса *basic\_ostream*, *basic\_fstream* — наследником класса *basic\_iostream*. Родительские классы *basic\_istream*, *basic\_ostream*, *basic\_iostream* в свою очередь, являются наследниками класса *basic\_ios*. Такая иерархия следует из того, что классы, ориентированные на работу с файлами, могут использовать методы более общих базовых классов. Классы *ifstream*, *ofstream* и *fstream* являются псевданимами классов *basic\_ifstream*, *basic\_ofstream*, *basic\_fstream* соответственно для типа *char* и объявлены в файле *<fstream>* (*file stream* – файловый поток).

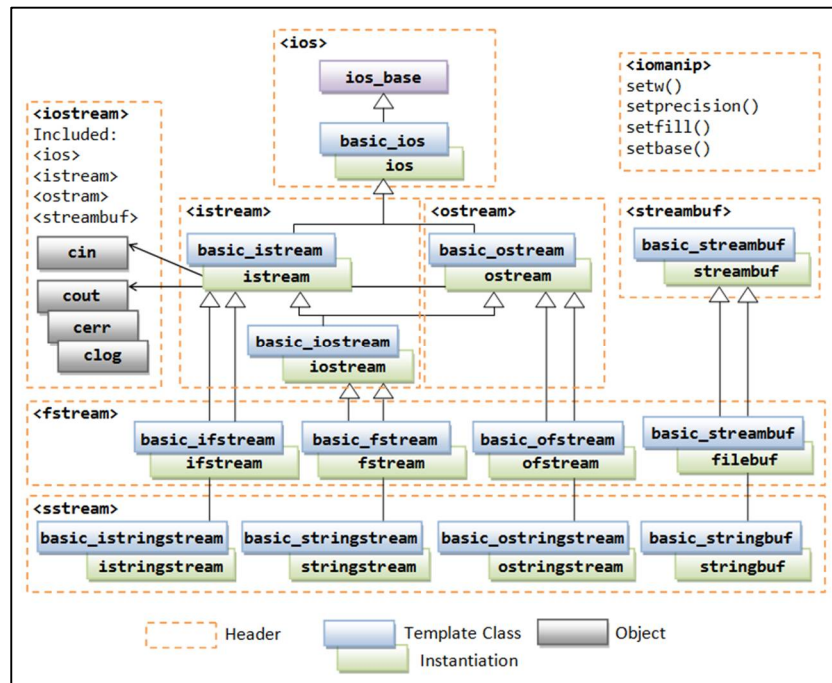


Рисунок 1 – Взаимосвязи основных классов организации потокового ввода-вывода языка C++

## ЗАПИСЬ ДАННЫХ В ФАЙЛ

При форматированном вводе/выводе числа хранятся на диске в виде набора символов. Например, число 6.02 вместо того, чтобы храниться в виде четырехбайтного значения типа *float* или восьмибайтного *double*, хранится в виде последовательности символов '6', '.', '0', '2'.

Следующая программа демонстрирует запись символа, целого числа, числа типа *double* и двух объектов типа *string* в дисковый файл.

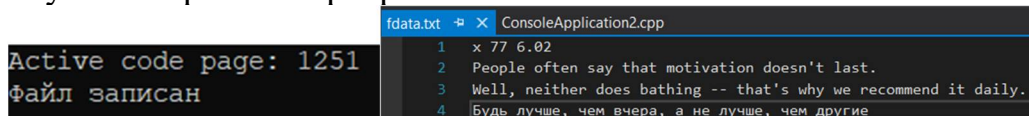
```
//Пример №1. Вывод данных в файл с использованием оператора <<
#include <fstream> //для файлового ввода/вывода
#include <iostream>
#include <string>
using namespace std;
int main() {
    system("chcp 1251");
    char ch = 'x';
    int j = 77;
    double d = 6.02;
    string str1 = "People often say that motivation doesn't last.
\nWell, neither does bathing - that's why we recommend it daily.";
    string str2 = "Будь лучше, чем вчера, а не лучше, чем другие";
    ofstream ofs("fdata.txt");
    if (!ofs.bad()) {
        ofs << ch //записать данные в поток
            << ' ' //записать разделитель
            << j //записать данные в поток
```

```

        << ' '//записать разделитель
        << d//записать данные в поток
        << '\n'//записать разделитель
        << str1//записать данные в поток
        << '\n'//записать разделитель
        << str2; //записать данные в поток
    cout << "Файл записан\n";
    ofs.close();
}
return 0;}

```

Результаты работы программы:



```

fdata.txt  x ConsoleApplication2.cpp
1  x 77 6.02
2  People often say that motivation doesn't last.
3  Well, neither does bathing -- that's why we recommend it daily.
4  Будь лучше, чем вчера, а не лучше, чем другие

```

Здесь определён и инициализирован файлом *fdata.txt* объект *ofs* типа *ofstream*. Этот локальный объект связывается с дисковым файлом с указанным именем. Если файл не существует, то он создается. Если файл уже существует, то он переписывается — новые данные в нем заменяют старые. Объект *ofs* ведет себя подобно объекту *cout*, поэтому можно использовать операцию вставки (<<) для вывода переменных любого стандартного типа в файл. Это работает потому, что оператор вставки перегружен в классе *basic\_ostream*, который является родительским для класса *basic\_ofstream*.

Когда программа завершается, для объекта *ofs* вызывается деструктор, который закрывает файл, но это можно сделать и явным образом с помощью метода *close()*.

Есть несколько потенциальных проблем с форматированным выводом в дисковые файлы. Во-первых, надо разделять числа (например, числа 77 и 6.02) нечисловыми символами. Поскольку они хранятся в виде последовательности символов, а не в виде полей фиксированной длины, то это единственный шанс узнать при извлечении, где заканчивается одно и начинается другое число. Во-вторых, между строками так же должны быть разделители. В этом примере для разделения данных мы использовали пробел и знак перехода на новую строку. Все эти разделители стоит учитывать при считывании данных.

### ЧТЕНИЕ ДАННЫХ ИЗ ФАЙЛА

Прочитать файл, созданный предыдущей программой, можно с использованием объекта типа *ifstream*. Файл автоматически открывается при создании объекта. Затем данные из него можно считать с помощью оператора извлечения из потока (>>).

```

//Пример №2. Чтение данных из файла с помощью оператора считывания из
//потока (>>)
#include <fstream>
#include <iostream>

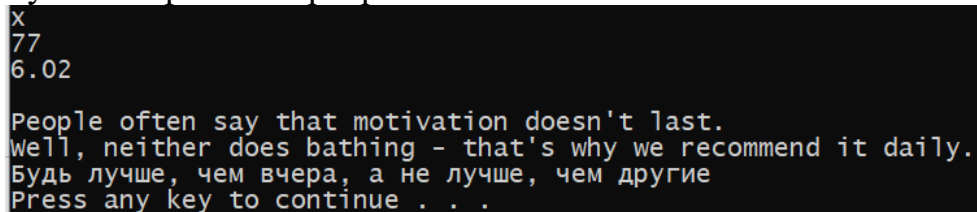
```

```

#include <string>
using namespace std;
void Ex2() {
    system("chcp 1251");
    system("cls");
    char ch;
    int j;
    double d;
    string str1, str2;
    ifstream infile("fdata.txt");//создать объект типа ifstream
    if (!infile.is_open()) {
        cout << "Не получается открыть файл для чтения данных!" <<
endl;
        return;
    }
    infile >> ch;
    infile >> j;
    infile >> d;
    //вывести считанные из файла данные на экран
    cout << ch << endl
        << j << endl
        << d << endl;
    while (getline(infile, str1)) { //пока не достигнут конец файла
        поместить очередную строку в переменную str1
        cout << str1 << endl; //выводим на экран str1
    }
}

```

Результаты работы программы:



```

x
77
6.02

People often say that motivation doesn't last.
Well, neither does bathing - that's why we recommend it daily.
Будь лучше, чем вчера, а не лучше, чем другие
Press any key to continue . . .

```

Эта программа открывает файл *fdata.txt* ассоциируя его с объектом *infile* типа *ifstream*. Поиск открываемого файла осуществляется в каталоге, где находится проект (это **рабочий каталог** проекта). При необходимости можно указать абсолютный (например, *D:\\Projects\\C++\\OOPiP\\FileIO\\fdata.txt*) или относительный (например, *Files\\fdata.txt*) путь к файлу. Программа лишь делает попытку открыть файл: возможно, этот файл не существует или не может быть открыт. Можно проверить результат создания объекта *ifstream* и определить, был ли файл успешно открыт, с помощью метода *is\_open()*.

Объект *infile* типа *ifstream* используется практически так же, как объект *cin*. Если данные корректно записаны в файл, то их можно извлечь с помощью оператора считывания из потока (*>>*). Считываемые данные необходимо

сохранить в соответствующих переменных и использовать в программе. Разумеется, числа приводятся обратно к своему двоичному представлению, чтобы с ними можно было работать в программе. Например, число 77 сохраняется в переменной типа *int*, это уже теперь не две семерки символьного типа, а целое число.

При работе с файлами необходимо писать код, который обрабатывает возможные ошибки. Файлы, например, могут не существовать в месте их поиска, могут быть повреждены или быть недоступными вследствие использования другим процессом системы. Во всех этих случаях операции над файлами завершаются неудачно. Программа должна быть готова к разнообразным ошибкам, вызывающим проблемы при работе с файлом, таким как сбой диска, повреждения файлов, некорректные секторы диска и др.

При работе с файлами можно определить, не произошла ли ошибка в процессе чтения, это можно сделать по значению, которое возвращает операция чтения:

```
if (infile >> number) { cout << "The value is: " << number; }
```

Проверяя результат вызова оператора *infile>>number*, можно обнаружить проблемы, связанные с дисковым накопителем и форматом считываемых данных. Проверяя значение, возвращаемое функцией чтения, можно проанализировать его на истинность (операция прошла успешно и можно пользоваться данными) и ложность (данные не корректны и это следует трактовать как ошибку).

### **ЗАПИСЬ В ФАЙЛ СТРОК С ПРОБЕЛАМИ**

Для работы со строками, содержащими пробелы после каждой строки необходимо писать специальный символ-ограничитель и использовать метод *getline()* для считывания строки.

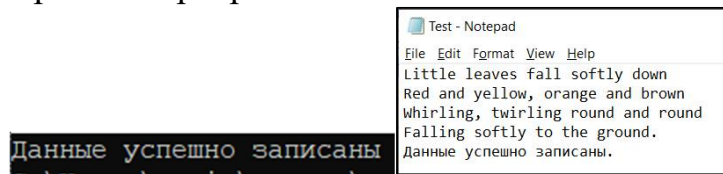
```
//Пример №3. Файловый вывод строковых данных
#include <fstream> //для операций файлового ввода/вывода
#include <iostream>
using namespace std;
void Ex3() {
    system("chcp 1251");
    system("cls");
    ofstream outfile("Test.txt"); //создать выходной файл
    if (outfile.is_open()) {
        //записать текст в файл
        outfile << "Little leaves fall softly down\n";
        outfile << "Red and yellow, orange and brown\n";
        outfile << "Whirling, twirling round and round\n";
        outfile << "Falling softly to the ground.\n";
        outfile << "Данные успешно записаны.\n";
    }
}
```

```

        cout << "Данные успешно записаны";
        outfile.close();
    }
    else cout << "Файл не может быть открыт";
}

```

Результаты работы программы:



После запуска программы строки стихотворения будут записаны в файл. Каждая строка заканчивается символом разделения строк ('`\n`'). Обратите внимание, это строки типа *char\**, а не объекты класса *string*.

Чтобы извлечь строки из файла, создадим *ifstream* и прочтем строчку за строчкой методом *getline()* класса *istream*. Этот метод считывает все символы, включая разделители, пока не дойдет до специального символа '`\n`', затем помещает результат чтения в буфер, переданный ей в качестве аргумента. Максимальный размер буфера передается с помощью второго аргумента.

```

//Пример №4. Файловый ввод строковых данных
#include <fstream>
#include <iostream>
using namespace std;
void Ex4() {
    system("chcp 1251");
    system("cls");
    const int MAX = 80; //размер буфера
    char buffer[MAX]; //массив символов
    ifstream infile("Test.txt"); //создать входной файл
    if (infile.is_open()) {
        while (!infile.eof()) {
            infile.getline(buffer, MAX); //читает строку текста
            cout << buffer << endl;
        }
    }
    infile.close();
}

```

Результаты работы программы:

```

Little leaves fall softly down
Red and yellow, orange and brown
Whirling, twirling round and round
Falling softly to the ground.
Данные успешно записаны.

Press any key to continue . . .

```



В результате работы программы на экране появятся та же часть из стихотворения, которая была записана в файл *Test.txt* в процессе работы предыдущей программы.

Программа продолжает чтение данных до тех пор, пока не встретит признак окончания файла (*EOF*). Эту программу нельзя применять для чтения произвольных файлов, — каждая строка текста должна заканчиваться символом '\n'. При попытке прочесть файл иной структуры программа будет работать некорректно.

### ПРИЗНАК КОНЦА ФАЙЛА (END OF FILE)

Объекты классов заголовка *ios* содержат флаги статуса ошибок, с помощью которых можно проверить результат выполнения операций. При чтении файла порциями будет достигнуто условие окончания файла. Сигнал *EOF* посылается в программу операционной системой, когда больше нет данных для чтения.

Это условие встречается, например, в выражении:

```
while (!infile.eof())//пока в потоке не достигнут EOF
```

Надо учитывать, что, проверяя конкретный флаг признака окончания файла, надо проверять и другие флаги ошибок. Флаги *failbit* и *badbit* тоже могут возникнуть при работе программы.

```
while (infile.good())//пока в потоке нет ошибок
```

Можно также проверять поток напрямую. Любой потоковый объект, например, *infile*, имеет значение, которое может тестироваться на предмет выявления наиболее распространенных ошибок, включая *EOF*. Если какое-либо из условий ошибки имеет значение *true*, то объект возвращает ноль. Если все идет хорошо, то объект возвращает ненулевое значение. Это значение на самом деле является указателем, но возвращаемый «адрес» никакой смысловой нагрузки не несет — он просто должен быть нулем либо не нулем.

```
while (infile)//пока в потоке нет ошибок
```

### ВВОД-ВЫВОД СИМВОЛОВ

Методы *put()* и *get()* классов *basic\_ostream* и *basic\_istream* соответственно, могут быть использованы для ввода и вывода единичных символов.

В следующей программе строка выводится в файл посимвольно.

```
//Пример №5. Посимвольный файловый вывод
#include <fstream>
#include <iostream>
```

```

#include <string>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    string str = "When you start thinking a lot about your past,\n"
        "it becomes your present and you can't see your future\n"
        "without it. Данные записаны.";
    ofstream outfile("Test.txt");//Создать выходной файл
    if (outfile.is_open()) {
        for (int j = 0; j < str.size(); j++)
            outfile.put(str[j]); //записывать строку посимвольно в файл
        cout << "Файл записан\n";
    }
    outfile.close();
    return 0;
}

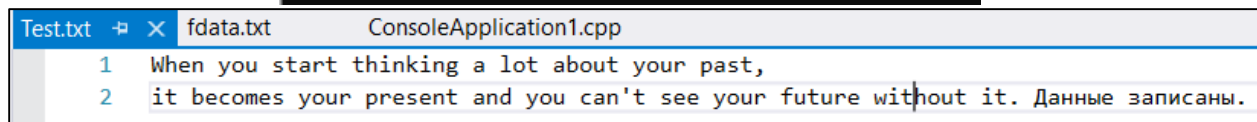
```

Результаты работы программы:

```

Файл записан
Press any key to continue . . .

```



В этой программе создается объект *outfile* класса *ofstream*. Длина строки *str* класса *string* находится с помощью метода *size()*, а символы выводятся в цикле *for* функцией *put()*. Фраза записывается в файл *Test.txt*. Считать и вывести содержимое этого файла можно с помощью следующей программы.

```

//Пример №6. Посимвольное считывание данных из файла
#include <fstream>
#include <iostream>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    char ch; //символ для сохранения считанных данных
    ifstream infile("Test.txt");//Входной файл
    if (infile.is_open()) {
        while (infile) { //читать до EOF или ошибки
            infile.get(ch); //считать символ
            if (infile.eof()) break;
            cout << ch; //вывести символ на экран
        }
    }
    infile.close();
}

```



```
    return 0;
}
```

Результаты работы программы:

```
When you start thinking a lot about your past,
it becomes your present and you can't see your future without it.
Данные записаны.Press any key to continue . . .
```

В программе используется метод *get()*. Чтение производится до признака окончания файла (или возникновения ошибки). Каждый прочитанный символ выводится на экран с помощью объекта *cout*, поэтому на консоль в результате работы программы будет выведен вся фраза, считанная из файла.

Есть и другой способ читать символы из файла — использовать метод *rdbuf()* (он является методом класса *basic\_ios*). Метод *rdbuf()* направляет поток в указанный буфер.

```
//Пример №7. Файловый ввод символов с использованием метода rdbuf()
#include <fstream>
#include <iostream>
using namespace std;
void Ex7() {
    system("chcp 1251");
    system("cls");
    ifstream infile("Test.txt");//создать входной файл
    cout << infile.rdbuf();//передать его буфер в объект cout
    cout << endl;
    infile.close();
}
```

Результат работы этой программы такой же как у предыдущей. Метод *rdbuf()* сам знает, что следует прекратить работу при достижении *EOF*.

## ДВОИЧНЫЙ ФАЙЛОВЫЙ ВВОД-ВЫВОД

Форматированный символьный файловый ввод/вывод чисел целесообразно использовать только при их небольшой величине и малом количестве. В противном случае эффективнее использовать двоичный ввод/вывод, при котором числа хранятся таким же образом, как в оперативной памяти компьютера, а не в виде символьных строк. Целочисленные значения занимают 4 байта, тогда как текстовая версия числа, например, «12345», занимает 5 байтов. Значения типа *float* также всегда занимают 4 байта. А форматированная версия «6.02314e13» занимает 10 байтов.

В следующем примере показано, как в бинарном виде массив целых чисел записывается в файл и читается из него. При этом используются две функции — *write()* (метод класса *basic\_ostream*), а также *read()* (метод класса *basic\_istream*).

Методы *read()* и *write()* работают с данными в виде байтов и предназначены для переноса байт из буфера в файл и обратно. Параметрами этих методов являются адрес буфера и его длина. Адрес должен быть вычислен с использованием оператора *reinterpret\_cast* относительно типа *char\**. Вторым параметром является длина буфера в байтах (а не число элементов в буфере).

```
//Пример №8. Двоичный ввод-вывод целочисленных данных
#include <fstream>
#include <iostream>
using namespace std;
const int MAX = 100; //размер буфера
int buff[MAX]; //буфер для хранения данных
void Ex8() {
    system("chcp 1251");
    system("cls");
    for (int j = 0; j < MAX; j++) //заполнить буфер данными
        buff[j] = j;
    ofstream os("Data.txt", ios::binary); //создание выходного
байтового потока
    if (os.is_open()) {
        //запись данных в файл
        os.write(reinterpret_cast<char*>(buff), MAX * sizeof(int));
        os.close();
    }
    for (int j = 0; j < MAX; j++) //обнулить содержимое буфера
        buff[j] = 0;
    ifstream is("Data.txt", ios::binary); //создание входного
байтового потока
    if (is.is_open()) {
        //чтение данных из файла
        is.read(reinterpret_cast<char*>(buff), MAX * sizeof(int));
        is.close();
    }
    for (int j = 0; j < MAX; j++) //проверка данных
        if (buff[j] != j) {
            cerr << "Данные считаны некорректно!\n";
            return;
        }
    for (int j = 0; j < MAX; j++) { //вывод считанных данных на экран
        cout << buff[j] << " ";
        if (j % 10 == 0 && j != 0) cout << endl;
    }
    cout << "Данные считаны корректно\n";
}
```

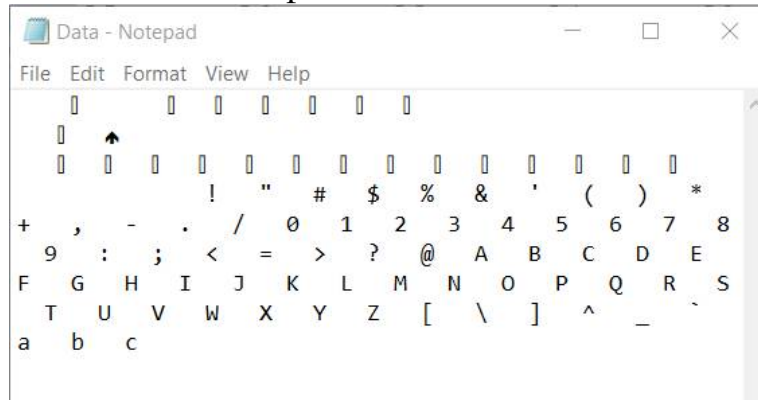
Результаты работы программы:

```

0 1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 Данные считаны корректно
Press any key to continue . . .

```

Содержимое записанного файла:



При работе с бинарными данными в качестве второго параметра функциям *write()* и *read()* следует передавать параметр *ios::binary*. Это необходимо по той причине, что текстовый режим, используемый по умолчанию, допускает иное обращение с данными нежели бинарный. Например, в текстовом режиме специальный символ '\n' занимает два байта (на самом деле это и есть два действия — перевод каретки и перевод строки). Но в бинарном режиме любой байт, *ASCII*-код которого отличен от 10 (код символа '\n', *line feed*, перевод строки), переводится двумя байтами.

### ОПЕРАТОР *reinterpret\_cast*

Оператор *reinterpret\_cast* используется для того, чтобы буфер данных типа *int* выглядел для функций *read()* и *write()* как буфер типа *char*.

```
is.read(reinterpret_cast<char*>(buff), MAX * sizeof(int));
```

**Оператор *reinterpret\_cast* изменяет тип данных в определенной области памяти, не анализируя смысл проводимых преобразований.** Вопрос целесообразности использования этого оператора находится в рамках ответственности программиста.

В рассмотренных примерах можно было вручную не закрывать открытые файлы, так как это делается автоматически при окончании работы с ними, т.е. для соответствующих объектов запускаются деструкторы при выходе из области видимости и закрываются ассоциированные с ними файлы. Если оба потока, входной и выходной, связаны с одним и тем же файлом, то первый поток должен быть закрыт до того, как откроется второй. Для этого

используется функция *close()*, которая вызывается для закрытия файла, не полагаясь на деструктор потока.

## ЗАПИСЬ И ЧТЕНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ТИПОВ ДАННЫХ

Так как язык C++ является объектно-ориентированным, то он требует реализацию возможности записи/чтения объектов в/из файлы/файлов. При записи объектов классов обычно используется бинарный режим. При этом на диск записывается та же битовая конфигурация, что хранится в оперативной памяти. В следующей программе у пользователя запрашивается информация об объекте класса *Person*, который потом записывается в файл *Person.dat*.

```
//Пример №9. Запись и считывание объекта в/из файл/файла
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
class Person {
protected:
    string fullName;//ФИО
    short age;//возраст
public:
    void getData() {//метод получение данных о человеке
        cout << "Введите ФИО: ";
        getline(cin, fullName, '\n');
        cout << "Введите возраст: ";
        cin >> age;
    }
    void showData() {//вывод данных на экран
        cout << "ФИО: " << fullName << endl;
        cout << "Возраст: " << age << endl;
    }
};
void Ex9() {
    system("chcp 1251");
    system("cls");
    Person pers;//создать объект
    pers.getData();
    ofstream outfile("Person.dat", ios::binary);
    outfile.write(reinterpret_cast<char*>(&pers), sizeof(pers));
    cout << "Данные успешно записаны в файл" << endl;
    outfile.close();
    ifstream infile("PERSON.DAT", ios::binary);
    infile.read(reinterpret_cast<char*>(&pers), sizeof(pers));
    pers.showData();
    infile.close();
}
```

Метод *getData()* класса *Person* вызывается для того, чтобы запросить у пользователя информацию, которая помещается в объект *pers*. Содержимое данного объекта записывается на диск с помощью метода *write()*. Для нахождения длины данных объекта *pers* используется оператор *sizeof*. Для чтения данных используется метод *read()*.

Результат работы программы:

```
Введите ФИО: Иванов Иван Петрович
Введите возраст: 20
Данные успешно записаны в файл
ФИО: Иванов Иван Петрович
Возраст: 20
Press any key to continue . . .
```

## СОВМЕСТИМОСТЬ СТРУКТУР ДАННЫХ

Для корректной работы программы чтения и записи объектов должны учитывать особенности различных типов данных. Например, объекты класса *Person* имеют длину 48 байт, из которых 40 отведено под имя человека, 2 — под возраст в формате *short*, остальные используются для реализации выравнивания. Если бы программы не знали длину полей, то данные были бы некорректно записаны или считаны из файла.

**Не имеет значения, какие используются методы в классах т.к. они не записываются в файл вместе с данными. Для данных важен единый формат записи, а разногласие между методами не имеет значения. Это утверждение верно только для классов, в которых не используются виртуальные функции.**

Если производится запись и считывание из файла объектов производных классов, то необходимо учитывать, что в объектах есть число, которое ставится перед началом их области данных в памяти. Оно идентифицирует класс объекта при использовании виртуальных функций. Когда объект записывается в файл, то это число записывается наряду с другими данными. Если изменяются методы класса, идентификатор также изменяется. Если в классе имеются виртуальные функции, то при чтении объекта с теми же данными, но другими методами возникнут несоответствия. **В этом случае, класс, использующийся для чтения объекта, должен быть идентичен классу, использовавшемуся при его записи.**

При попытке дискового ввода/вывода объектов, компонентами которых являются указатели, значения указателей не будут корректными при чтении объекта в другую область памяти.

## ВВОД-ВЫВОД НАБОРА ОБЪЕКТОВ ПОЛЬЗОВАТЕЛЬСКОГО ТИПА

Предыдущие программы записывали и читали только один объект. Рассмотрим запись и считывание набора число объектов.

```

//Пример №10. Чтение из файла и запись в файл нескольких объектов
#include <fstream>
#include <iostream>
using namespace std;
class Person {
protected:
    string name;//имя
    int age;//возраст
public:
    void getData() {//получить данные о человеке
        cout << "\nВведите имя: ";
        cin >> name;
        cout << "Введите возраст: ";
        cin >> age;
    }
    void showData() {//вывод данных о человеке
        cout << "\n Имя: " << name;
        cout << "\n Возраст: " << age;
    }
};
void Ex10() {
    system("chcp 1251");
    system("cls");
    char ch;
    Person pers;
    fstream file;//создать файл для ввода/вывода
    //открыть файл для дозаписи
    file.open("Group.dat", ios::app | ios::out | ios::in |
        ios::binary);
    do {
        cout << "Введите данные о человеке:";
        pers.getData();
        //записать данные в файл
        file.write(reinterpret_cast<char*>(&pers), sizeof(pers));
        cout << "Продолжить ввод (y/n)? ";
        cin >> ch;
    } while (ch == 'y');
    file.seekg(0);//поместить указатель текущей позиции в началофайла
    //считать данные о первом человеке
    file.read(reinterpret_cast<char*>(&pers), sizeof(pers));
    while (!file.eof()) {
        cout << "\nСотрудник:";
        pers.showData();
        file.read(reinterpret_cast<char*>(&pers), sizeof(pers));
    }
    cout << endl;
}

```

Результаты работы программы:

```
Введите данные о человеке:
Введите имя: Александр
Введите возраст: 22
Продолжить ввод (y/n)? y
Введите данные о человеке:
Введите имя: Иван
Введите возраст: 33
Продолжить ввод (y/n)? y
Введите данные о человеке:
Введите имя: Николай
Введите возраст: 30
Продолжить ввод (y/n)? n

Сотрудник:
Имя: Александр
Возраст: 22
Сотрудник:
Имя: Иван
Возраст: 33
Сотрудник:
Имя: Николай
Возраст: 30
```

В программе создан файл, который может быть использован одновременно как входной и выходной. Это должен быть объект класса *fstream*, наследник класса *iostream*, порожденного классами *istream* и *ostream*, чтобы была возможность поддержки одновременно операций ввода и вывода. Для открытия файла в программе использован подход, который одним выражением создает файл, а другим открывает его, используя функцию *open()*.

### БИТЫ РЕЖИМОВ ФАЙЛА

Биты режимов, определенные в классе *ios\_base*, определяют различные способы открытия потоковых объектов (таблица №1).

Таблица 1 – Биты режимов файла

Бит режима	Действие
<i>app</i>	Запись, начиная с конца файла ( <i>APPend</i> (присоединять в конец))
<i>ate</i>	Чтение, начиная с конца файла ( <i>AT End</i> )
<i>binary</i>	Открыть в бинарном режиме
<i>in</i>	Открытие для чтения (по умолчанию для <i>ifstream</i> )
<i>out</i>	Открытие для записи (по умолчанию для <i>ofstream</i> )
<i>trunc</i>	Обрезать файл до нулевой длины, если он уже существует ( <i>TRUNCate</i> (усекать))

В предыдущей программе использовался бит *ios::app*, потому что требовалось сохранить все, что было записано в файл до этого. То есть можно записать что-нибудь в файл, завершить программу, запустить ее заново и продолжать записывать данные, сохранив при этом результаты предыдущей сессии. Биты *ios::in* и *ios::out* используются для указания того, что необходимо осуществлять одновременно ввод и вывод. Бит *ios::binary* необходим для записи объектов в бинарном виде. **Вертикальные слешы между флагами**



**нужны для того, чтобы из битов сформировалось единое целое число. При этом несколько флагов будут применяться одновременно.**

За один раз в файл записывается один объект типа *Person* с помощью метода *write()*. После окончания записи файл читается с начала, для этого нужно установить указатель чтения файла на начало. Это выполняет функция *seekg()*. Затем в цикле *while* считывается объект из файла и выводится на экран. Это продолжается до тех пор, пока не будут прочитаны все объекты класса *Person*, — состояние, определяемое флагом *ios::eofbit*.

### УКАЗАТЕЛИ ТЕКУЩЕЙ ПОЗИЦИИ ФАЙЛА

У каждого файлового объекта есть два ассоциированных с ним значения, называемые **указатель чтения и указатель записи** (текущая позиция чтения и текущая позиция записи). Эти значения определяют номер байта относительно начала файла, с которого будет производиться чтение или запись.

Для реализации возможности чтения и записи в произвольном месте файла используются функции *seekg()* и *tellg()*, позволяющие устанавливать и проверять текущий указатель чтения, и функции *seekp()* и *tellp()* — выполнять те же действия для указателя записи.

Вариант функции *seekg()* имеет аргумент, представляющий собой абсолютную позицию относительно начала файла. Начало принимается за 0. Это условно показано на рисунке 2.

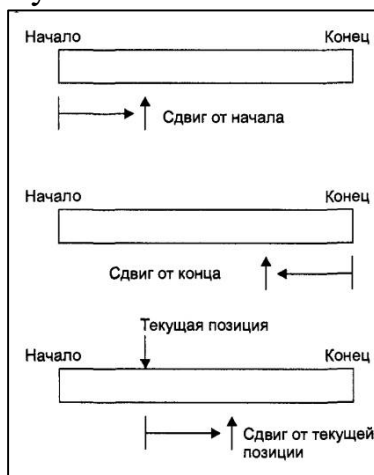


Рисунок 2 – Позиции файлового указателя

Функция *seekg()* может использоваться в двух вариантах. Первый из них использует аргумент для указания позиции относительно начала файла. Второй вариант с двумя аргументами позволяет указать сдвиг относительно определенной позиции в файле (первый аргумент) и позицию, начиная с которой отсчитывается сдвиг (второй аргумент). Второй аргумент может иметь три значения: *beg* означает начало файла (*beginning*), *cur* — текущую позицию указателя файла (*current*), *end* — это конец файла. Например, выражение ниже установит указатель записи за 10 байтов до конца файла.

```
seekg(-10, ios::end);
```

Рассмотрим пример использования двухаргументного варианта функции *seekg()* для нахождения конкретного объекта (человека) класса *Person* в файле *Group.dat* и для вывода данных об этом человеке.

```
//Пример №11. Поиск объекта в файле
#include <fstream>
#include <iostream>
using namespace std;
class Person {
protected:
    string name;//имя
    int age;//возраст
public:
    void getData() { //получить данные о человеке
        cout << "\nВведите имя: ";
        cin >> name;
        cout << "Введите возраст: ";
        cin >> age;
    }
    void showData() { //вывод данных о человеке
        cout << "\n Имя: " << name;
        cout << "\n Возраст: " << age;
    }
};
void Ex11() {
    system("chcp 1251");
    system("cls");
    Person pers;//создать объект person
    ifstream infile;//создать входной файл
    infile.open("GROUP.DAT", ios::in | ios::binary);
    infile.seekg(0, ios::end); //установить указатель на 0 байт от
конца файла
    int endposition = infile.tellg();//найти позицию в файле
    int n = endposition / sizeof(Person); //число человек
    cout << "\nВ файле " << n << " человек(а)";
    cout << "\nВведите номер персоны: ";
    cin >> n;
    int position = (n - 1) * sizeof(Person); //умножить размер
созданных данных на число персон
    infile.seekg(position); //число байт от начала
//прочитать одну персону
    infile.read(reinterpret_cast<char*>(&pers), sizeof(pers));
    pers.showData(); //вывести одну персону
    cout << endl;
}
```

Результат работы программы:

```
В файле 3 человек(а)
Введите номер персоны: 2

Имя: Иван
Возраст: 33
```

Для удобства работы пользователя объекты в файле (персоны) нумеруются, начиная с единицы, а в программе нумерация ведется с нуля. Поэтому персона №2 – это второй из трех человек, записи о которых имеются в файле.

В программе вычисляется количество человек в файле. Она делает это установкой указателя чтения на конец файла с помощью выражения

```
infile.seekg(0, ios::end); //установить указатель на 0 байт от конца
файла
```

Функция *tellg()* возвращает текущую позицию указателя чтения. Программа использует ее для вычисления длины файла в байтах. Так как длина одной записи известна, то, исходя из общей длины файла, можно узнать, сколько всего объектов хранится в файле.

```
int endposition = infile.tellg(); //найти позицию в файле
int n = endposition / sizeof(Person); //число человек
```

После выбора пользователем необходимого объекта, с помощью функции *seekg()* считается, сколько нужно отступить от начала файла для того, чтобы попасть на начало области данных выбранного объекта. Затем вызывается функция *read()* для чтения данных, начиная с этого места.

```
infile.seekg(position); //число байт от начала
//прочитать одну персону
infile.read(reinterpret_cast<char*>(&pers), sizeof(pers));
```

Функция *showData()* выводит информацию на экран.

## ОБРАБОТКА ОШИБОК ФАЙЛОВОГО ВВОДА-ВЫВОДА

При работе с файлами могут возникать следующие ситуации: файл может не существовать; файл может существовать, и программа может перезаписать нужные данные; может закончиться место на диске; диск может оказаться неисправным и т. д. Следовательно, необходимо использовать механизм реагирования на подобного типа ошибки.

В следующем примере рассмотрим возможность проверки выполнения дисковых операций. Если возникла ошибка, выведем сообщение об этом, и программа завершит работу. В примере использована техника, которая заключается в проверке значения, возвращаемого из объекта, и определении

статуса его ошибки. Программа открывает выходной потоковый объект, записывает массив целых чисел вызовом функции *write()* и закрывает объект. Затем открывает входной объект и считывает массив функцией *read()*.

```
//Пример №12. Обработка ошибок файлового ввода-вывода
#include <fstream>
#include <iostream>
#include <process.h> //для вызова функции exit()
using namespace std;
const int MAX = 1000;
int buff[MAX];
int main() {
    system("chcp 1251");
    system("cls");
    for (int j = 0; j < MAX; j++) buff[j] = j;
    ofstream os;
    os.open("Data.dat", ios::trunc | ios::binary);
    if (!os) {
        cerr << "Невозможно открыть выходной файл\n";
        exit(1);
    }
    cout << "Идет запись данных...\n";
    os.write(reinterpret_cast<char*>(buff), MAX * sizeof(int));
    if (!os) {
        cerr << "Запись в файл невозможна\n";
        exit(1);
    }
    os.close();
    for (int j = 0; j < MAX; j++) //обнулить содержимое буфера
        buff[j] = 0;
    ifstream is;
    is.open("Data.dat", ios::binary);
    if (!is) {
        cerr << "Невозможно открыть входной файл\n";
        exit(1);
    }
    cout << "Идет чтение данных...\n"; //чтение файла
    is.read(reinterpret_cast<char*>(buff), MAX * sizeof(int));
    if (!is) {
        cerr << "Невозможно чтение файла\n";
        exit(1);
    }
    for (int j = 0; j < MAX; j++) //проверить данные
        if (buff[j] != j) {
            cerr << "\nДанные некорректны\n";
            exit(1);
        }
}
```

```

    cout << "Данные считаны корректно\n";
    return 0;
}

```

Результаты работы программы:

```

Идет запись данных...
Идет чтение данных...
Данные считаны корректно

```

В вышеприведенном примере определяется наличие ошибки ввода/вывода проверкой значения, возвращаемого потоковым объектом.

```
if (!is) // возникла ошибка
```

Здесь *is* возвращает значение указателя, если все прошло без ошибок. В противном случае возвращается 0. Это недифференцированный подход к определению ошибок: не имеет значения, какая именно ошибка возникла, все ошибки обрабатываются одинаково. С помощью флагов статуса ошибок *ios* можно извлечь более подробную информацию об ошибках файлового ввода/вывода.

В следующем примере показано, как можно использовать их при файловом вводе/выводе.

```

//Пример №13. Проверка ошибок открытия файла
#include <fstream>
#include <iostream>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    ifstream file;
    file.open("a:Test.dat");
    if (!file) cout << "Невозможно открыть Test.dat";
    else cout << "\nФайл открыт без ошибок.";
    cout << "\nКод ошибки = " << file.rdstate();
    cout << "\ngood() = " << file.good();
    cout << "\neof() = " << file.eof();
    cout << "\nfail() = " << file.fail();
    cout << "\nbad() = " << file.bad() << endl;
    file.close();
    return 0;
}

```

Результат работы программы:

```
Невозможно открыть test.dat
Код ошибки = 2
good() = 0
eof() = 0
fail() = 1
bad() = 0
```

В начале программа проверяет значение файлового объекта. Если оно нулевое, значит, файл, возможно, не существует и поэтому не может быть открыт.

Метод *rdstate()* является элементом класса *basic\_ios* и возвращает состояние соответствующего потока. Система ввода-вывода C++ поддерживает информацию о состоянии, касающуюся результата выполнения каждой операции ввода-вывода, которая связана с активным потоком. Функция *rdstate()* возвращает нуль (*ios::goodbit*), когда не обнаружено никакой ошибки; в противном случае устанавливается бит ошибки.

Код ошибки, возвращаемый *rdstate()*, равен двум. Это бит, который говорит о том, что файл не существует. Остальные биты при этом равны нулю. Функция *good()* возвращает 1 (*true*) лишь в том случае, когда не установлены никакие биты ошибок, поэтому в примере она вернула 0 (*false*). Указатель файла находится не на *EOF*, поэтому *eof()* возвращает 0. Флаг *fail()* установлен так как требуемая операция не выполнена, флаг *bad()* *true*, если произошла неустраняемая ошибка. **Оптимально в полноценных проектах использовать все эти функции после каждой операции ввода/вывода.**

## ФАЙЛОВЫЙ ВВОД-ВЫВОД С ПОМОЩЬЮ МЕТОДОВ КЛАССА

При написании сложных классов логично включать операции файлового ввода/вывода в методы класса. Иногда необходимо разрешить каждому компоненту класса читать и записывать самого себя. В следующем примере добавлено два метода — *diskOut()* и *diskIn()* — в класс *Person*. Эти функции позволяют объектам класса *Person* записывать себя в файл и читать себя же из него. Все объекты будут храниться в файле *Persfile.dat*. Новые объекты всегда будут добавляться в конец файла. Аргумент функции *diskIn()* позволяет читать данные о любом человеке из файла. Метод *diskCount()* возвращает число людей, информация о которых хранится в файле. При вводе данных следует использовать только фамилии людей, пробелы не допускаются.

//Пример №14. Файловый ввод-вывод объектов класса Person

```
#include <fstream>
#include <iostream>
using namespace std;
class Person {
protected:
    string name;//имя
    int age;//возраст
public:
    void getData(void) { //получить данные
```

```

        cout << "\nВведите фамилию: "; cin >> name;
        cout << "Введите возраст: "; cin >> age;
    }
    void showData(void) {
        cout << "\n Имя: " << name;
        cout << "\n Возраст: " << age;
    }
    void diskIn(int); //чтение из файла
    void diskOut(); //запись в файл
    static int diskCount(); //число человек в файле
};
void Person::diskIn(int personNumber) {
    ifstream infile;
    infile.open("Persfile.dat", ios::binary);
    infile.seekg(personNumber * sizeof(Person));
    infile.read((char*)this, sizeof(*this));
}
void Person::diskOut() {
    ofstream outfile;
    outfile.open("Persfile.dat", ios::app | ios::binary);
    outfile.write((char*)this, sizeof(*this));
}
int Person::diskCount() { //подсчет число людей в файле
    ifstream infile;
    infile.open("Persfile.dat", ios::binary);
    infile.seekg(0, ios::end); //перейти на позицию «0 байт от конца
    файла»
    return (int)infile.tellg() / sizeof(Person); //количество людей
}
int main() {
    system("chcp 1251");
    system("cls");
    Person person; //создать пустую запись
    char ch;
    do {
        cout << "Введите данные о человеке: ";
        person.getData(); //Получить данные
        person.diskOut(); //записать на диск
        cout << "Продолжить (y/n)? ";
        cin >> ch;
    } while (ch == 'y');
    int n = Person::diskCount(); //сколько людей в файле?
    cout << "В файле " << n << " человек(а)\n";
    for (int j = 0; j < n; j++) {
        cout << "\nПерсона " << j + 1;
        person.diskIn(j); //считать с диска
        person.showData(); //вывести данные
    }
}

```



```

    cout << endl;
    return 0;
}

```

В данном примере особенности дисковых операций невидимы для функции *main()*, они спрятаны внутрь класса *Person*.

Заранее неизвестно, где находятся данные, с которыми необходимо работать, так как каждый объект находится в своей области памяти. Указатель *this* «знает», где находимся объект во время выполнения метода. В потоковых функциях *read()* и *write()* адрес объекта, который будет читаться или записываться, равен *this*, а его размер — *sizeof(\*this)*. Результат работы программы:

```

Введите данные о человеке:
Введите фамилию: Иванов
Введите возраст: 33
Продолжить (y/n)? y
Введите данные о человеке:
Введите фамилию: Гребеньков
Введите возраст: 55
Продолжить (y/n)? y
Введите данные о человеке:
Введите фамилию: Ершова
Введите возраст: 19
Продолжить (y/n)? n
В файле 5 человек(а)

Персона 1
Имя: Малахова
Возраст: 22
Персона 2
Имя: Сидоров
Возраст: 21
Персона 3
Имя: Иванов
Возраст: 33
Персона 4
Имя: Гребеньков
Возраст: 55
Персона 5
Имя: Ершова
Возраст: 19

```

Чтобы пользователь мог ввести собственное имя файла, вместо жесткого закрепления его в программе, следует создать статическую компонентную переменную (например, *charfileName[]*), а также статическую функцию для ее установки.

Предположим, что в памяти хранится большое число объектов, и все их нужно записать в файлы. Недостаточно иметь для каждого из них метод, который откроет файл, запишет в него объект, потом закроет файл, как было в предыдущем примере. Быстрее и логичнее в такой ситуации открыть файл, записать в него все объекты, которые необходимо, и закрыть файл.

## СТАТИЧЕСКИЕ ФУНКЦИИ КЛАССА ПРИ РАБОТЕ С ФАЙЛАМИ

Одним из способов записать за один сеанс множество объектов является употребление статической функции, которая применяется ко всему классу в целом, а не к отдельным его объектам. Такая функция может обращаться к массиву указателей на объекты, который можно хранить в виде статической

переменной. При создании каждого объекта его указатель заносится в этот массив. Статический элемент данных также может хранить данные о том, сколько всего объектов было создано. Статическая функция записи в файл откроет файл, в цикле пройдет по всем ссылкам массива, записывая по очереди все объекты, и после окончания записи закроет файл.

Необходимо учитывать, что объекты, хранящиеся в памяти, могут иметь различные размеры. Типичной предпосылкой этого является создание порожденных классов. Например, рассмотрим программу, где класс *Employee*, является базовым для классов *Manager*, *Scientist*, *Laborer*. Объекты трех порожденных классов имеют разные размеры, так как содержат разные объемы данных. Например, в дополнение к имени и порядковому номеру, которые присущи всем работникам, у менеджера есть такие поля данных, как должность и членские взносы гольф-клуба, а у ученого есть поле данных, содержащее количество его публикаций.

Хотелось бы записать данные из списка, содержащего все три типа порожденных объектов, используя простой цикл и метод *write()* класса *ofstream*. Для этого нужно знать размеры объектов, чтобы передать их в качестве второго аргумента этой функции.

Пусть имеется массив указателей *staff[]*, хранящий ссылки на объекты типа *Employee*. Указатели могут ссылаться, таким образом, и на все три порожденных класса. При использовании виртуальных функций можно использовать выражения типа

```
staff[]->putData();
```

Тогда будет выполнена во время работы программы версия функции *putData()*, соответствующая объекту, на который ссылается указатель, а не та, что соответствует базовому классу. Можно ли использовать *sizeof()* для вычисления размера ссылочного аргумента? То есть можно ли написать такое выражение:

```
ouf.write((char*)staff[j], sizeof(*staff[j]));
```

Такая запись является некорректной, так как *sizeof()* не является виртуальной функцией. Она будет обращаться к типу объекта самого указателя, а не к типу объекта, на который ссылается указатель. Эта функция всегда будет возвращать размер объекта базового класса.

Одним из решений данного вопроса является использование оператора *typeid*, который позволяет определить тип объекта во время выполнения программы. Чтобы использовать *typeid*, надо включить параметр компилятора *RTTI* (это существенно только для компилятора *Microsoft Visual C++*).

Следующий пример показывает, как все вышеописанное работает. Узнав размер объекта, можно использовать его в функции *write()* для записи в файл. Некоторые специфические методы сделаны виртуальными, чтобы можно было пользоваться массивом указателей на объекты.

//Пример №15. Файловый ввод/вывод объектов класса. Поддержка объектов неодинаковых размеров. Использовать для реализации индивидуального задания по лабораторной работе

```
#include <fstream>
#include <iostream>
#include <string>
#include <iomanip>
#include <typeinfo> //для typeid()
#include <process.h>
using namespace std;
const int MAXEMPL = 100; //максимальное число работников
enum EmployeeType { manager, scientist, laborer };
class Employee {
    string name; //фамилия работника
    unsigned long number; //номер работника
    static int emplNum; //текущее количество работников
    static Employee* staff[]; //массив указателей
public:
    Employee() {
        name="N/A";
        number = 0;
    }
    virtual void getData() {
        cin.ignore(10, '\n');
        cout << "Введите фамилию сотрудника: "; getline(cin,name);
        cout << "Введите номер сотрудника: "; cin >> number;
    }
    virtual void putData() {
        cout << "\nФамилия: " << name;
        cout << "\nНомер: " << number;
    }
    virtual EmployeeType getType(); //получить тип сотрудника
    static void add(); //добавить сотрудника в штат
    static void display(); //вывести данные обо всех сотрудниках
    static void read(); //чтение данных из файла
    static void write(); //запись данных в файл
};
int Employee::emplNum; //текущее число работников
Employee* Employee::staff[MAXEMPL]; //массив указателей
class Manager : public Employee {
private:
    string title; //должность ("вице-президент" и т.д.)
```

```

        float dues;//Налоги гольф-клуба
public:
    Manager() {
        title="";
        dues = 0;
    }
    void getData() {
        Employee::getData();
        cin.ignore();
        cout << "Введите должность: "; getline(cin,title);
        cout << "Введите налоги: "; cin >> dues;
    }
    void putData() {
        Employee::putData();
        cout << "\nДолжность: " << title;
        cout << "\nНалоги гольф-клуба: " << dues;
    }
};

class Scientist : public Employee {
private:
    int numbPubl;//количество публикаций
public:
    Scientist() {
        numbPubl = 0;
    }
    void getData() {
        Employee::getData();
        cout << "Введите количество публикаций: ";
        cin >> numbPubl;
    }

    void putData() {
        Employee::putData();
        cout << "\nКоличество публикаций: " << numbPubl;
    }
};

class Laborer : public Employee {};
void Employee::add() {
    char userChoice;
    cout << "'m' для добавления менеджера"
        << "\n's' для добавления ученого"
        << "\n'l' для добавления рабочего"
        << "\nВаш выбор: ";
    cin >> userChoice;
    switch (userChoice) { //создать объект указанного типа
    case 'm': staff[emplNum] = new Manager; break;
    case 's': staff[emplNum] = new Scientist; break;
    case 'l': staff[emplNum] = new Laborer; break;
    }
}

```

```

        default: cout << "\nНеизвестный тип работника\n";
                return;
        }
        staff[emplNum++]>getData();//Получить данные от пользователя
    }
    void Employee::display() { //Вывести данные обо всех работниках
        for (int j = 0; j < emplNum; j++) {
            cout << (j + 1); //вывести номер
            switch (staff[j]>getType()) { //вывести тип работника
                case EmployeeType::manager: cout << ". Тип: Менеджер"; break;
                case EmployeeType::scientist: cout << ". Тип: Ученый"; break;
                case EmployeeType::laborer: cout << ". Тип: Рабочий"; break;
                default: cout << ". Неизвестный тип";
            }
            staff[j]>putData();//Вывод данных
            cout << endl;
        }
    }
    EmployeeType Employee::getType() { //Возврат типа объекта
        if (typeid(*this) == typeid(Manager)) return
            EmployeeType::manager;
        else if (typeid(*this) == typeid(Scientist)) return
            EmployeeType::scientist;
        else if (typeid(*this) == typeid(Laborer)) return
            EmployeeType::laborer;
        else {
            cerr << "\nНеправильный тип работника";
            exit(1);
        }
        return EmployeeType::manager;
    }
    void Employee::write() { //Записать все объекты в файл
        int size = 0;
        cout << "Идет запись " << emplNum << " работников.\n";
        ofstream ouf; //открыть ofstream
        EmployeeType etype; //тип каждого объекта Employee
        ouf.open("EMPLOY.DAT", ios::trunc | ios::binary);
        if (!ouf) {
            cout << "\nНевозможно открыть файл\n";
            return;
        }
        for (int j = 0; j < emplNum; j++) { //Для каждого объекта получить
тип
            etype = staff[j]>getType();
            ouf.write((char*)&etype, sizeof(etype));
            switch (etype) {
                case EmployeeType::manager:
                    size = sizeof(Manager); break;

```

```

        case EmployeeType::scientist:
            size = sizeof(Scientist); break;
        case EmployeeType::laborer:
            size = sizeof(Laborer); break;
    }
    outf.write((char*)(staff[j]), size); //запись объекта Employee
    if (!outf) {
        cout << "\nЗапись в файл невозможна\n";
        return;
    }
}

void Employee::read() { //чтение всех данных из файла
    int size; //размер объекта Employee
    EmployeeType etype; //тип работника
    ifstream inf;
    inf.open("EMPLOY.DAT", ios::binary);
    if (!inf) {
        cout << "\nНевозможно открыть файл\n";
        return;
    }
    emplNum = 0; //В памяти работников нет
    while (true) {
        inf.read((char*)&etype, sizeof(etype)); //чтение типа
        работника
        if (inf.eof()) break;
        if (!inf) { //ошибка чтения типа
            cout << "\nНевозможно чтение типа\n";
            return;
        }
        switch (etype) {
            //создать нового работника
            case EmployeeType::manager: //корректного типа
                staff[emplNum] = new Manager;
                size = sizeof(Manager);
                break;
            case EmployeeType::scientist:
                staff[emplNum] = new Scientist;
                size = sizeof(Scientist);
                break;
            case EmployeeType::laborer:
                staff[emplNum] = new Laborer;
                size = sizeof(Laborer);
                break;
            default: cout << "\nНеизвестный тип в файле\n"; return;
        } //чтение данных из файла
        inf.read((char*)staff[emplNum], size);
        if (!inf) { //ошибка, но не EOF

```

```

        cout << "\nЧтение данных из файла невозможно\n";
        return;
    }
    emplNum++; //счетчик работников увеличить
} //end while
cout << "Идет чтение " << emplNum << " работников\n";
}
int main() {
    system("chcp 1251");
    system("cls");
    char userChoice;
    while (true) {
        cout << "'a' - добавление сведений о работнике"
              << "\n'd' - вывести сведения обо всех работниках"
              << "\n'w' - записать все данные в файл"
              << "\n'r' - прочитать все данные из файла"
              << "\n'x' - выход"
              << "\nВаш выбор: ";
        cin >> userChoice;
        switch (userChoice) {
            case 'a': //добавить работника
                Employee::add(); break;
            case 'd': //вывести все сведения
                Employee::display(); break;
            case 'w': //запись в файл
                Employee::write(); break;
            case 'r': //чтение всех данных из файла
                Employee::read(); break;
            case 'x': exit(0); //выход
            default: cout << "\nНеизвестная команда";
        }
    }
    return 0;
}

```

## КОД ТИПА ОБЪЕКТА

Для создания возможности определения класса объекта при считывании данных из файла, необходимо при записи данных на диск записывать и код типа объекта (перечисляемую переменную типа *EmployeeType*) перед данными объекта. До начала чтения из файла надо прочитать его значение и создать объект соответствующего типа, после этого копировать данные из файла в соответствующий новый объект.

При попытке считывать данные объекта просто «куда-нибудь», например, в массив типа *char*, а затем установки указателя на этот массив, такой объект не будет создан:

```
char someArray[MAX];
```



```
aClass* objPtr;  
objPtr = reinterpret_cast<aClass*>(someArray); //плохой подход
```

Есть только два легитимных способа создать объект. Определить объект вручную, тогда он будет создан на этапе компиляции:

```
aClass anObj;
```

Либо создать объект в процессе работы программы, используя оператор *new* и ассоциировав его адрес с указателем:

```
objPtr = new aClass;
```

При корректном способе создания объекта запускается его конструктор. Это необходимо даже тогда, когда не определен собственный, а используется конструктор по умолчанию.

Рассмотрим взаимодействие с предыдущей программой. В памяти созданы объекты классов *Manager*, *Scientist*, *Laborer*. Они записываются на диск, считываются, выводятся на экран.

```

'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор: a
'm' для добавления менеджера
's' для добавления ученого
'l' для добавления рабочего
Ваш выбор: m
Введите фамилию сотрудника: Лесков
Введите номер сотрудника: 56
Введите должность: старший менеджер
Введите налоги: 23.20
'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор: a
'm' для добавления менеджера
's' для добавления ученого
'l' для добавления рабочего
Ваш выбор: s
Введите фамилию сотрудника: Шкубель
Введите номер сотрудника: 89
Введите количество публикаций: 222
'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор: a
'm' для добавления менеджера
's' для добавления ученого
'l' для добавления рабочего
Ваш выбор: l
Введите фамилию сотрудника: Костицин
Введите номер сотрудника: 111
'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор: w
Идет запись 3 работников.
'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор: r
Идет чтение 3 работников
'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор: d
1. Тип: Менеджер
Фамилия: Лесков
Номер: 56
Должность: старший менеджер
Налоги гольф-клуба: 23.2
2. Тип: Ученый
Фамилия: Шкубель
Номер: 89
Количество публикаций: 222
3. Тип: Рабочий
Фамилия: Костицин
Номер: 111
'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор:

```

Выйти из программы можно сразу после записи на диск. Когда программа повторно запустится, все данные появятся и смогут быть прочитаны. Программу можно расширить добавлением функций удаления работника, извлечения данных об конкретном работнике, поиска работника с конкретными характеристиками и т. д.

## ПЕРЕГРУЗКА ОПЕРАТОРОВ ИЗВЛЕЧЕНИЯ И ВСТАВКИ

Этот прием C++ позволяет поддерживать ввод/вывод пользовательских типов наравне со стандартными типами. Например, если имеется объект класса *Employee*, называющийся *empl*, его можно вывести на экран простым выражением:

```
cout << "\nempl=" << empl;
```

Этот способ не отличается от подобных конструкций для стандартных типов. Операторы извлечения и вставки могут быть перегружены и для работы

с консольным вводом/выводом. Но можно перегрузить их и для работы с дисковыми файлами.

Рассмотрим пример, в котором операторы извлечения и вставки для класса *Distance* перегружены для работы с объектами *cout* и *cin*.

```
//Пример №16. Использование перегрузки операторов вывода в поток (<<)
и считывания из потока (>>)
#include <iostream>
using namespace std;
class Distance { //класс английских расстояний
private:
    int feet;
    float inches;
public:
    Distance() : feet(0), inches(0.0) {}
    Distance(int ft, float in) : feet(ft), inches(in) { }
    friend istream& operator >>(istream& s, Distance& d);
    friend ostream& operator <<(ostream& s, Distance& d);
};
//перегрузка оператора считывания из потока
istream& operator >>(istream& s, Distance& d) {
    cout << "\nВведите футы: ";
    s >> d.feet;
    cout << "Введите дюймы: ";
    s >> d.inches;
    return s;
}
//перегрузка оператора вывода в поток
ostream& operator <<(ostream& s, Distance& d) {
    s << d.feet << "\'-" << d.inches << '\"';
    return s;
}
int main() {
    system("chcp 1251");
    system("cls");
    Distance dist1, dist2; //Определение переменных
    Distance dist3(11, 6.25);
    cout << "Введите два значения расстояний:";
    cin >> dist1 >> dist2; //получение значения
    //вывод расстояний
    cout << "\ndist1 = " << dist1 << "\ndist2 = " << dist2;
    cout << "\ndist3 = " << dist3 << endl;
    return 0;
}
```

Программа запрашивает у пользователя два значения расстояний типа *Distance*, выводит их, а также выводит значения, которые были инициализированы в программе.

```

Введите два значения расстояний:
Введите футы: 10
Введите дюймы: 3.5

Введите футы: 12
Введите дюймы: 3.2

dist1 = 10'-3.5"
dist2 = 12'-3.2"
dist3 = 11'-6.25"

```

Операторы извлечения и вставки перегружаются одинаковыми способами. Они возвращают ссылку на объект типа *istream* (для оператора `>>`) или *ostream* (для оператора `<<`). Возвращаемые значения могут организовываться в цепочки. Операторы имеют по два аргумента, оба передаются по ссылке. Первый аргумент оператора `>>` — объект класса *istream* (например, *cin*). Для оператора `<<` первым аргументом должен быть объект класса *ostream* (например, *cout*). Второй аргумент — объект класса, для которого осуществляется ввод/вывод (в примере это объект типа *Distance*). Оператор ввода `>>` берет входные данные из потока, указанного в первом аргументе, и переносит их в компонентные данные объекта, указанного во втором. По аналогии оператор вывода `<<` берет данные из объекта, указанного во втором аргументе, и посылает их в поток, соответствующий значению первого аргумента.

Функции `operator<<()` и `operator>>()` должны быть дружественными по отношению к классу *Distance*, так как объекты *istream* и *ostream* находятся слева от знака операции.

## ПЕРЕГРУЗКА ОПЕРАТОРОВ << И >> ДЛЯ РАБОТЫ С ФАЙЛАМИ

Рассмотрим перегрузку операторов `<<` и `>>` в классе *Distance* для работы с файловым вводом/выводом.

//Пример №17. Перегрузка операторов `<<` и `>>` для работы с файлами

```

#include <fstream>
#include <iostream>
using namespace std;
class Distance { //класс английских расстояний
private:
    int feet;
    float inches;
public:
    Distance() : feet(0), inches(0.0) { }
    Distance(int ft, float in) : feet(ft), inches(in) { }
    friend istream& operator>>(istream& s, Distance& d);
    friend ostream& operator<<(ostream& s, Distance& d);

```

```

};
//получить данные из файла или с клавиатуры
//для ('), (-) и (") с помощью перегруженного оператора >>
istream& operator>>(istream& s, Distance& d) {
    char dummy;
    s >> d.feet >> dummy >> dummy >> d.inches >> dummy;
    return s;
}
//запись данных в файл или на экран
ostream& operator<<(ostream& s, Distance& d) {
    s << d.feet << "\"'-" << d.inches << "\"";
    return s;
}
int main() {
    system("chcp 1251");
    system("cls");
    char ch;
    Distance dist1;
    ofstream ofile;
    ofile.open("Dist.dat");
    do {
        cout << "Введите расстояние: ";
        cin >> dist1; //получить данные от пользователя
        ofile << dist1; //записать их в выходной поток
        cout << "Продолжать (y/n)? ";
        cin >> ch;
    } while (ch != 'n');
    ofile.close(); //закрыть выходной поток
    ifstream ifile; //создать и открыть
    ifile.open("DIST.DAT"); //входной поток
    cout << "\nСодержимое файла:\n";
    while (true) {
        ifile >> dist1; //чтение данных из потока
        if (ifile.eof()) break;
        cout << "Расстояние = " << dist1 << endl;
    }
    return 0;
}

```

В сами перегружаемые операции были внесены минимальные изменения. Оператор >> больше не просит ввести входные данные, потому что принимает их из файла. Программа предполагает, что пользователь знает, как точно вводить значения футов и дюймов, включая знаки пунктуации. Оператор << остался без изменений. Программа запрашивает у пользователя входные данные и после получения каждого значения записывает его в файл. Когда пользователь заканчивает ввод данных, программа читает из файла и выводит на экран все хранящиеся значения. Результат работы программы:

```

Введите расстояние: 3'-4.5"
Продолжать (y/n)? y
Введите расстояние: 7'-11.25"
Продолжать (y/n)? y
Введите расстояние: 11'-6"
Продолжать (y/n)? n

Содержимое файла:
Расстояние = 3'-4.5"
Расстояние = 7'-11.25"
Расстояние = 11'-6"

```

Значения расстояний записаны в файле символ за символом. Поэтому реальное содержание файла таково:

```

DIST.DAT  Exceptions_Examples.cpp
1  3'-4.5"7'-11.25"11'-6"

```

Если пользователь ошибется при вводе данных, то они не будут записаны корректно в файл и не смогут быть прочитаны оператором <<, для обработки этой ситуации необходимо проверять, правильно ли производится ввод.

### КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНОЙ РАБОТЕ:

1. Назовите потоковые классы, предназначенные для файлового ввода/вывода.
2. Для чего предназначена функция *fopen()*? Напишите её прототип.
3. Для чего предназначена функция *fseek()*? Напишите её прототип.
4. Напишите выражение, создающее объект *saleFile* класса *ofstream*, и ассоциируйте его с файлом *Sales.txt*.
5. Напишите оператор *if*, определяющий, достиг ли объект типа *ifstream* под названием *saleFile* конца файла или же возникла ошибка.
6. Напишите оператор, записывающий единичный символ в объект *fileOut* класса *ofstream*.
7. Напишите операторы, позволяющие считать все содержимое объекта *iFile* класса *ifstream*, в массив *buff*.
8. Какие утверждения про биты режимов *app* и *ate* являются верными:
  - а) определяются в классе *ios*;
  - б) могут устанавливаться для чтения или для записи;
  - в) работают с функциями *put()* и *get()*;
  - г) устанавливают режимы открытия файлов.
9. Что такое «текущая позиция» в файле?
10. Напишите выражение, сдвигающее текущую позицию в файле на 13 байтов назад в потоковом объекте *fl*.

### ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ:

1. Изучить теоретические сведения, полученные на лекции и лабораторной работе, ознакомиться с соответствующими материалами литературных источников.
2. Ответить на контрольные вопросы лабораторной работы.
3. Разработать алгоритм программы по индивидуальному заданию.

4. Написать, отладить и проверить корректность работы созданной программы.

5. Написать электронный отчет по выполненной лабораторной работе.

**Отчет должен быть оформлен по стандарту БГУИР ([Стандарт предприятия СТП 01-2017 "Дипломные проекты \(работы\). Общие требования"](#)) и иметь следующую структуру:**

1. титульный лист
2. цель выполнения лабораторной работы
3. теоретические сведения по лабораторной работе
4. формулировка индивидуального задания
5. весь код решения индивидуального задания, разбитый на необходимые типы файлов
6. скриншоты выполнения индивидуального задания
7. выводы по лабораторной работе

**В РАМКАХ ВСЕГО КУРСА ООП П ВСЕ ЛАБОРАТОРНЫЕ РАБОТЫ ДОЛЖНЫ ХРАНИТЬСЯ В ОДНОМ РЕШЕНИИ (SOLUTION), В КОТОРОМ ДОЛЖНЫ БЫТЬ СОЗДАНЫ ОТДЕЛЬНЫЕ ПРОЕКТЫ (PROJECTS) ДЛЯ КАЖДОЙ ЛАБОРАТОРНОЙ РАБОТЫ. ВО ВСЕХ ПРОЕКТАХ ПОЛЬЗОВАТЕЛЬ ДОЛЖЕН САМ РЕШАТЬ ВЫЙТИ ИЗ ПРОГРАММЫ ИЛИ ПРОДОЛЖИТЬ ВВОД ДАННЫХ. В КАЖДОМ ЗАДАНИИ НЕОБХОДИМО СОЗДАТЬ ДИАГРАММУ КЛАССОВ НА ЯЗЫКЕ UML ДЛЯ ОТОБРАЖЕНИЯ ВСЕХ СОЗДАННЫХ КЛАССОВ И СВЯЗЕЙ МЕЖДУ НИМИ. В ОТЧЁТ ВКЛЮЧИТЬ СОЗДАННУЮ ДИАГРАММУ КЛАССОВ.**

#### **ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ К ЛАБОРАТОРНОЙ РАБОТЕ №4:**

На основе разработанной иерархии классов, реализованной в лабораторной работе №3 «ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ В ЯЗЫКЕ C++», реализовать программу для работы с данными, используя потоки файлового ввода-вывода. Реализовать функций

1. добавления данных в файл,
2. удаления данных из файла,
3. редактирования данных в файле,
4. просмотра данных из файла,
5. поиска данных по необходимым параметрам в файле.

Использовать функции *open()*, *is\_open()*, *bad()*, *good()*, *close()*, *eof()*, *fail()* для анализа состояния файла, перегрузить операторы вывода в поток << и считывания из потока >>, использовать функции *seekg()*, *seekp()*, *tellg()*.