

Тема: Файловый ввод/вывод в C++

- Основные понятия
- Файловый ввод/вывод с помощью потоков
- Дополнительные функции файлового ввода / вывода
- Режимы файлов
- Двоичные файлы
- Произвольный доступ к файлам

Файлы

Некоторые понятия ООП

Класс является описанием совокупности сходных между собой объектов.

Объект класса часто также называют **экземпляром** класса.

Программные объекты схожи с объектами реального мира.

Файлы в C++

1. Основные понятия

Связь с внешними источниками, приемниками и носителями информации в Си++ осуществляется только с помощью **файлов**. **Файлом в C++** считается также **любое внешнее устройство**, по своему назначению являющееся источником или приемником информации, например, клавиатура, принтер, диск и т. д.

Такое устройство принято называть **логическим**, поскольку учитывается только его главная функция, а не физические характеристики.

Файлы

ЗАМЕЧАНИЕ: До начала операции ввода-вывода конкретному внешнему файлу должна быть поставлена в соответствие **специальная переменная в программе**.

Обычно различают **текстовые** и **двоичные файлы**.

Текстовые файлы состоят из строк, которые завершаются символом конца строки - '\n'.

Файлы

2. Файловый ввод/вывод с помощью потоков

Файл рассматривается как поток (*stream*), представляющий собой последовательность считываемых или записываемых байтов.

ЗАМЕЧАНИЕ: расшифровка смысла записанных последовательностей в байте лежит в программе.

Файлы

Для работы с файлами необходимо подключить заголовочный файл *fstream.h*.

ЗАМЕЧАНИЕ: файл *iostream.h* автоматически включается, т.е. его не нужно подключать явно.

В Си++ определены три класса файлового ввода/вывода:

- *ifstream* – входные файлы для чтения;
- *ofstream* – выходные файлы для записи;
- *fstream* – файлы для чтения и записи.

Файлы

Этапы записи (ввода) информации в файл:

1) Создать переменную (объект) типа *ofstream* для управления потоком вывода

Например:

```
ofstream fout;
```

ЗАМЕЧАНИЕ: Имя объекта может быть любым допустимым именем C++.

2) Поставить в соответствие объекту определенный файл (связать объект с файлом). Это можно сделать с помощью функции *open()*:

```
fout.open("test.txt");
```

Файлы

ЗАМЕЧАНИЕ: 1 и 2 этап можно объединить оператором:

```
ofstream fout("test.txt");
```

3) Использовать созданный объект аналогично, как `cout`:

```
int i = 1, j = 25;
```

```
double a = 25e6;
```

```
char s[10] = "строка";
```

```
fout<<i<<' '<<j<<' '<<a<<' '<<s<<endl;
```

В файле после закрытия получим текст:

```
1 25 2.5e+07 строка
```


Файлы

ЗАМЕЧАНИЕ:

Открытие файла таким способом позволяет создать новый файл, если файла с таким именем не существует.

Если же такой файл уже есть, то до открытия для вывода этот файл урезается до нулевого размера и информация начинает выводиться в пустой файл.

Чтобы избежать перезаписи (а дописать в конец файла), следует указать флаг:

```
fout.open("test.txt", ios::app);
```

Файлы

Этапы чтения (вывода) из файла:

1) Создать переменную (объект) типа *ifstream* для управления потоком ввода

Например:

```
ifstream fin;
```

2) Поставить этот объект в соответствие определенному файлу. Это можно сделать с помощью функции *open()*:

```
fin.open("test.txt");
```

ЗАМЕЧАНИЕ: 1 и 2 этап можно объединить

оператором: **`ifstream fin("test.txt");`**

Файлы

3) Использовать созданный объект аналогично, как `cin`:

```
int x;  
float f;  
char st[20];  
fin >> x >> f; //чтение из файла  
           //числовых данных  
fin.getline(st,20); //чтение строки  
fin.close();
```

ЗАМЕЧАНИЕ: Если необходимо одновременно работать с несколькими открытыми файлами, то для каждого из них нужно создавать отдельный поток.

Файлы

ЗАМЕЧАНИЕ:

Для проверки наличия нужного файла на диске (в случае отсутствия файла значение объекта потока равно 0):

```
if (!fin) //или (!fin.is_open())
{
    cout << "Ошибка!!!\n";
    return;
}
```

Файлы

3. Дополнительные функции файлового ввода / вывода

- Возвращает указатель на буфер, связанный с потоком:

```
filebuf* rdbuf();
```

- Осуществляет проверку того, было ли успешным открытие файла. Возвращает нулевое значение в случае ошибки:

```
int is_open();
```

Пример: `if (!fin.rdbuf()->is_open())`

```
    cerr<<"Не удалось открыть файл...\n";
```

Файлы

ЗАМЕЧАНИЕ:

- **cin** – объект класса `istream`, соответствующий стандартному вводу. В общем случае он позволяет читать данные с терминала пользователя;
- **cout** – объект класса `ostream`, соответствующий стандартному выводу. В общем случае он позволяет выводить данные на терминал пользователя;
- **cerr** – объект класса `ostream`, соответствующий стандартному выводу для ошибок. В этот поток мы направляем сообщения об ошибках программы.

Файлы

- Возвращает ненулевое значение, если достигнут конец файла:

```
int eof();
```

Пример:

```
while (!fin.eof()) fin >> x;
```

- Устанавливает состояние потока в ноль.

```
void clear(int = 0);
```

ЗАМЕЧАНИЕ: Эту функцию необходимо вызывать, если работу с потоком нужно продолжать после возникновения таких ситуаций, как достижение конца файла, ошибка при обмене с потоком и т. п.

Файлы

- Выводит в поток один символ:

```
ostream& put(char) ;
```

Допускает сцепленный вызов:

```
fout.put('A').put('\n') ;
```

- Выводит в файл из символьного массива, на который указывает первый параметр, число символов, указанных вторым параметром:

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

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


Файлы

Например, оператор

```
fout.write(s, 5);
```

записывает в поток `fout` 5 символов из массива `s`.

Причем эти символы никак не обрабатываются, а просто выводятся в качестве сырых байтов данных.

Среди этих символов, например, может встретиться в любом месте нулевой символ, но он не будет рассматриваться как признак конца строки.

Файлы

- Аналогичный метод для чтения данных в символьный массив также без всякой обработки:

```
istream& read(signed char*,int) ;  
istream& read(unsigned char*,int) ;
```

- Возвращает количество символов, действительно прочитанных последней операцией ввода. Используется совместно с предыдущей функцией:

```
int gcount() ;
```

Файлы

- Читает одиночный символ из указанного потока (даже если это символ-разделитель) и возвращает этот символ в качестве значения вызова функции:

```
int get();
```

ЗАМЕЧАНИЕ: Если в потоке встретился признак конца файла, возвращает значение *EOF*.
Данный вариант функции удобно использовать для поиска в файле какого-то ключевого символа.

Файлы

Например, цикл поиска в файле символа «\$» можно организовать следующим образом:

```
char c;  
while ((c = fin.get()) != EOF)  
    if (c=='$')  
        break;  
...
```

Файлы

- Вводит очередной символ из входного потока (даже если это символ-разделитель) и сохраняет его в символьном аргументе:

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

```
istream& get(signed char&);
```

ЗАМЕЧАНИЕ: Этот вариант функции возвращает 0, когда встречается признак конца файла; в остальных случаях возвращается ссылка на тот поток, для которого вызывалась функция.

Файлы

Предыдущий пример с использованием данного варианта функции *get()* можно было бы переписать так:

```
while (fin.get(c))
```

```
...
```

Файлы

- Третий вариант функции *get()* принимает три параметра: символьный массив, максимальное число элементов и символ-ограничитель (по умолчанию символ перевода строки `'\n'`):

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

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

Файлы

ЗАМЕЧАНИЕ: Символы читаются из входного потока до тех пор, пока не достигается число символов, на 1 меньшее указанного максимального числа, или пока не считывается ограничитель. Затем для завершения введенной строки в символьный массив помещается нулевой символ. Ограничитель в символьный массив не помещается, а остается во входном потоке (он будет следующим считываемым символом). Таким образом, результатом второго подряд использования функции `get()` явится пустая строка, если только ограничитель не удалить из входного потока.

Файлы

- Действует подобно предыдущему (третьему) варианту функции *get()*, но удаляет символ-ограничитель из потока (т. е. читает этот символ и отбрасывает его); этот символ не сохраняется в символьном массиве:

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

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

.

Файлы

ЗАМЕЧАНИЕ: С помощью *getline()* можно следующим образом записать цикл чтения файла по строкам:

```
while (!fin.eof())  
{  
    fin.getline(s, 80);  
    ... //обработка строки  
}
```

.

Файлы

4. Режимы файлов

Режим файла описывает, как используется файл: для чтения, для записи, для добавления и т. д.

Когда поток связывается с файлом, а также при инициализации файлового потокового объекта именем файла или при работе с функцией `open()`, можно использовать и второй аргумент, задающий режим файла:

```
ifstream fin("file1.txt", ios::in) ;  
ofstream fout ;  
fout.open("file2.dat", ios::app) ;
```

Файлы

Список констант (флагов), задающих режим файла

Константа	Значение
<code>ios::in</code>	Открыть файл для чтения
<code>ios::out</code>	Открыть файл для записи
<code>ios::ate</code>	Переместить указатель в конец файла после открытия
<code>ios::app</code>	Добавить информацию к концу файла
<code>ios::trunc</code>	Урезать файл, если он существует
<code>ios::nocreate</code>	Не открывать новый файл (для несуществующего файла функция <code>open()</code> выдаст ошибку)
<code>ios::noreplace</code>	Не открывать существующий файл (для существующего выходного файла, не имеющего режимов <code>ate</code> или <code>app</code> , выдать ошибку)
<code>ios::binary</code>	Двоичный файл

Файлы

Чтобы дописать в конец файла:

```
ofstream fout;
fout.open("test.txt", ios::app);
if (fout.is_open())
{
    fout << "Hi!!!";
    fout.close();
} else
{
    cout << "Ошибка!!!\n";
    return;
}
```

Файлы

ЗАМЕЧАНИЕ

1) Если при связывании потока с файлом необходимо указать одновременно несколько режимов, их следует перечислять через | (операция «битовое ИЛИ»).

Например, чтобы открыть файл для добавления данных, нужно использовать следующий оператор:

```
ofstream fout("myfile.txt",  
             ios::out|ios::app);
```

Файлы

2) По умолчанию, при связывании файла с потоком **ВВОДА** используется константа **`ios::in`** (открыть для чтения), а при связывании с потоком **ВЫВОДА** – **`ios::out|ios::trunc`** (открыть файл для записи и стереть его содержимое).

Файлы

5. Двоичные файлы

Данные в файле можно сохранить в текстовой форме или двоичном формате.

Текстовая форма означает, что все данные сохраняются как текст, даже числа.

Например, сохранение значения $-2.324216e+07$ в текстовой форме означает сохранение 13 символов, из которых состоит данное число. Или число 12345678 записывается, как 8 символов, а это 8 байт данных, несмотря на то, что число помещается в целый тип.

Файлы

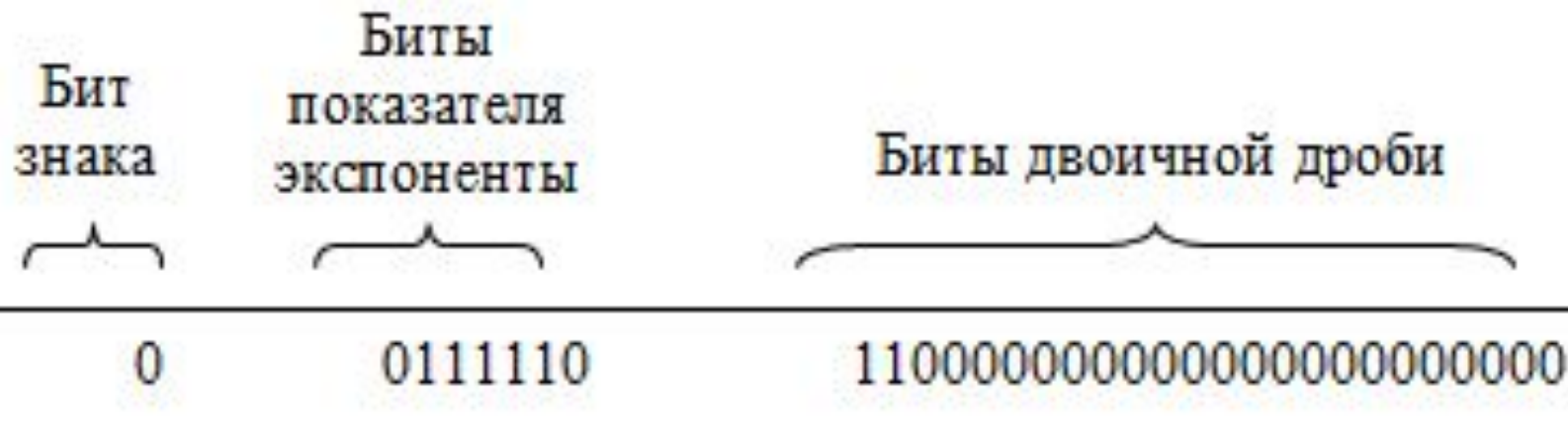
Двоичный формат означает, что число сохраняется во внутреннем представлении, т. е. вместо символов сохраняется 64-разрядное представление числа типа *double*.

Для символа двоичное представление совпадает с его текстовым – двоичным представлением ASCII-кода (или его эквивалента) символа.

ЗАМЕЧАНИЕ: для чисел двоичное представление очень сильно отличается от их текстового представления

Файлы

Например, двоичное представление числа **0.375**



Текстовое представление числа 0.375

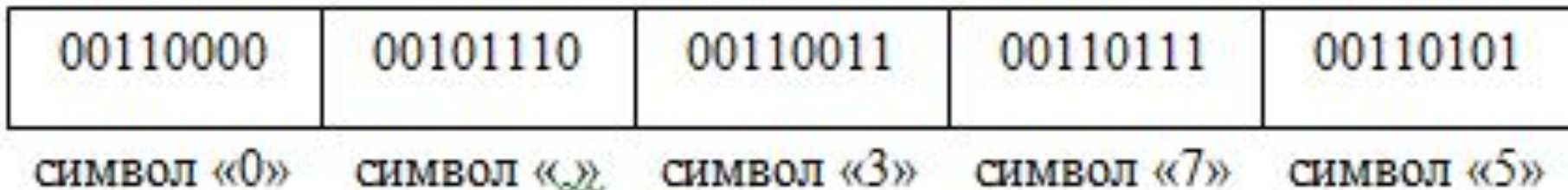


Рисунок 1 – Двоичное и текстовое представление числа с плавающей точкой

Файлы

Особенности сохранения в двоичном формате чисел:

- Числа сохраняются более точно, поскольку он позволяет сохранить точное внутреннее представление числа.
- Нет ошибок преобразования и округления.
- Сохранение данных может происходить быстрее, т. к. при этом не происходит преобразования и данные можно сохранять большими блоками.
- обычно занимает меньше места.

Файлы

Режим для работы с двоичными файлами:

`ios::binary`

1) для сохранения данных в **двоичном формате** используется функция ***write()***, которая копирует определенное число байтов из памяти в файл - она может копировать любой тип данных байт в байт, не производя преобразования.

Файлы

Недостатки: адрес переменной необходимо преобразовать к типу *указатель-на-`char`*.

Чтобы узнать размер переменной в байтах, можно воспользоваться операцией **`sizeof`**.

Пример:

```
long x = 10L;  
ofstream fout("file1.dat",  
ios::out|ios::binary);  
fout.write((char*)&x, sizeof(x));
```

Файлы

2) для чтения данных из **двоичного файла**

Используется соответствующая функция *read()* с объектом типа *ifstream*:

```
ifstream
```

```
fin ("file1.dat", ios::in | ios::binary) ;
```

```
fin.read ( (char*) &x, sizeof (x) ) ;
```

Данный блок кода копирует количество байтов *sizeof(x)* из файла в переменную *x*.

ЗАМЕЧАНИЕ: Подобным образом можно сохранять в файлы и читать из них и переменные более сложных типов, например, структуры.

Файлы

Пример 1: записать число в бинарный файл.
Прочитать число из бинарного файла

```
int y = 0; //Y будем записывать в файл
int x = 0; //X будем считывать из файла
cout << "Y = ";
cin >> y; //Вводим число, //которое
        нужно сохранить в файл
ofstream out("C:/1.txt",
ios::binary|ios::out); //Открываем
//файл в двоичном режиме для записи
//Записываем в файл число y
out.write((char*)&y, sizeof y);
```

Файлы

```
out.close(); //Закрываем файл
//Показываем X до его изменений
cout << "x = " << x << endl;
ifstream in("C:/1.txt",
ios::binary|ios::in); //Открываем файл
// в двоичном режиме только для чтения
in.read((char*)&x,sizeof x); //Читаем
//оттуда информацию и запоминаем её в X
in.close(); //Закрываем файл

//Показываем X после изменения
cout << "x = " << x << endl;
```


Файлы

Пример 2: Запись объекта структуры в бинарный файл. Чтение объекта структуры из бинарного файла

```
//Исходная структура  
struct MyStruct  
{  
    char *Name;  
    int size;  
};
```

Файлы

```
MyStruct X,Y; //Создали два объекта,  
//соответствующие структуре. Например  
//объект X имеет такие параметры  
X.Name = "Иванов";  
X.size = 100;  
//Открываем файл для записи в бинарном  
режиме  
ofstream out("C:/2.txt",  
ios::binary|ios::out);  
//Записываем объект X в открытый файл  
out.write((char*)&X,sizeof X);  
out.close(); //Закрываем открытый файл
```

Файлы

```
//Открываем файл только для чтения,  
открываем в бинарном режиме  
fstream in("C:/2.txt",ios::  
    binary|ios::in);  
    //Считываем информацию в объект Y  
in.read((char*)&Y,sizeof Y);  
in.close(); //Закрываем открытый файл  
  
//Показываем объект Y по его составным  
// частям  
cout << Y.Name << "\n";  
cout << Y.size << "\n";
```

Файлы

Пример 3: создаем два двоичных файла из одного массива

```
int m[10]={0};  
    /* заполняем массив m числами */  
for(int i = 0; i < 10; i++)  
{  
    m[i] = i;  
    cout << m[i] << ' ';  
    // контрольный вывод на экран  
}  
cout << '\n';
```

Файлы

```
/* открываем файл для записи */  
ofstream outstrm  
("c:/binfiles/oonumber1.bin",  
    std::ios::binary);  
if (outstrm.is_open())  
{  
    //поэлементно выводим массив в  
        //файл  
    for (int i = 0; i < 10; i++)  
        outstrm.write((char *)&m[i],  
            sizeof(int));  
    outstrm.close();  
}
```

Файлы

```
/* открываем другой файл для записи */  
outstrm.open("c:/binfiles/oonumber2.bin",  
std::ios::binary);  
if(outstrm.is_open())  
{  
    // выводим массив в файл  
    ostrm.write((char*)m, sizeof(m));  
    ostrm.close();  
}
```

Файлы

```
// вывод двоичного файла на экран
// открываем второй файл для чтения
{
    ifstream instrm
        ("c:/binfiles/oonumber2.bin",
         std::ios::binary);
    int a = 0;
// читаем числа по одному из файла и
// выводим
while(instrm.read((char *)&a,
                 sizeof(int)))
    cout << a << ' ';
cout << '\n';
}
```

Файлы

```
// открываем первый файл для чтения
ifstream instrm
    ("c:/binfiles/oonumber1.bin",
     std::ios::binary);
int t[10] = {0};
// чтение файла в массив
instrm.read((char *)t, sizeof(t));
instrm.close(); // закрываем
for(int i = 0; i < 10; i++)
    cout << t[i] << ' ';
cout << '\n';
```


Файлы

В этом примере два двоичных файла из одного массива создаются разными способами: в файл `oopumber1.bin` массив выводится поэлементно, а в файл `oopumber2.bin` — сразу целиком одним оператором.

В каталоге `BinFiles` эти два файла имеют одинаковый размер в 40 байт.

Файлы

Затем те же файлы открываются как входные, читаются и выводятся на экран. Сначала открывается файл `oopnumber2.bin` (в который мы писали массив целиком), и чтение из него выполняется по одному числу.

Нетрудно вместо вывода на экран выполнять в цикле, например, суммирование чисел, записанных в этот файл.

Файлы

Первый файл `oopnumber1.bin`, который записывался в цикле по одному числу, читается сразу целиком в массив `t` одним оператором, и поток тут же закрывается. Такое «смешение» для двоичных файлов безопасно, так как и в памяти, и на диске размеры данных равны `sizeof(тип) * n`, где `n` — количество элементов, участвующих в обмене.

Файлы

Код С++ Запись объекта класса в бинарный файл. Чтение объекта класса из бинарного файла

=====

```
#include <stdlib.h>
```

```
#include <iostream.h>
```

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

```
class MyClass
```

```
{
```

```
    int z; //Недоступен ничему кроме своего класса
```

```
public:
```

```
    int x;
```

```
    int y;
```

```
    MyClass() {z=100;} //Инициализация z с
```

Файлы

6. Произвольный доступ к файлам

Произвольный доступ к файлам предоставляет возможность переместиться в любое место файла сразу, вместо последовательного передвижения по нему.

ЗАМЕЧАНИЕ:

- 1) Используется часто при обработке файлов баз данных.
- 2) Этот подход проще реализовать, если файл состоит из набора записей одинакового типа (или хотя бы размера).

Файлы

Функции для реализации «передвижения» по файлу:

- **seekg ()** – передвигает указатель ввода (используется с объектом типа *ifstream*)
- **seekp ()** – указатель вывода в определенную точку файла (используется с объектом типа *ofstream*).

Файлы

Прототипы функции передвижения указателя ввода:

- `istream& seekg(long) ;` – устанавливает указатель чтения входного потока на абсолютную позицию, заданную параметром.

ЗАМЕЧАНИЕ: Эта позиция отстоит от начала файла на указанное количество байтов, т. е. значение позиции можно трактовать как смещение от начала файла, где первый байт имеет индекс 0.

Поэтому оператор `fin.seekg(112) ;` передвигает файловый указатель на 112-й байт, являющийся реальным 113-м байтом файла.

Файлы

- `istream& seekg(long, seek_dir);` – перемещает указатель чтения входного потока на число байтов, заданное первым параметром.

ЗАМЕЧАНИЕ: Вторым параметром задается точка отсчета: значение `ios::beg` означает, что смещение отсчитывается от начала файла, `ios::cur` – от текущей позиции, `ios::end` – от конца файла.

Файлы

Примеры вызова функции:

//30 байтов от начала файла

```
fin.seekg(30, ios::beg) ;
```

//один байт назад от текущей позиции

```
fin.seekg(-1, ios::cur) ;
```

//переход к концу файла

```
fin.seekg(0, ios::end) ;
```

Файлы

Прототипы функции передвижения указателя вывода:

- `ostream& seekp(long) ;`
- `ostream& seekp(long, seek_dir) ;`

ЗАМЕЧАНИЕ: Принципы работы этой функции полностью идентичны предыдущей, за исключением того, что она работает с объектом потока вывода:

```
fout.seekp(20, ios::beg) ;
```

Файлы

Функции для проверки текущей позиции файлового указателя:

- `tellg()` - для потока ввода
- `tellp()` – для потока вывода

ЗАМЕЧАНИЕ: Каждый из них возвращает типа *long*, представляющее собой текущее смещение указателя от начала файла в байтах.

Когда создается объект типа *fstream*, входной и выходной указатели передвигаются одновременно, поэтому в таком случае функции `tellg()` и `tellp()` возвращают одинаковое значение.

Файлы

Но если используется объект типа *ifstream* для управления потоком ввода и объект типа *ofstream* для управления потоком вывода, входной и выходной указатели передвигаются независимо друг от друга и функции *tellg()* и *tellp()* могут возвращать разные значения.