26. ПОТОКОВЫЙ ВВОД-ВЫВОД

Мы уже неоднократно пользовались различными потоками ввода/вывода. Здесь мы рассмотрим работу с потоками более подробно. Ввод/вывод потоков в C++ используется для преобразования типизированных объектов в читаемый текст и обратно.

Классы, связанные с потоками C++, содержат расширяемые библиотеки, позволяющие выполнять форматированный ввод/вывод с контролем типов как для предопределенных, так и для определяемых пользователем типов данных с помощью перегруженных операций и прочих объектно ориентированных методов.

Потоком будем называть понятие, относящееся к любому переносу данных от источника (или поставщика данных) к приемнику (или потребителю) данных. Несмотря на свое имя, класс потока может быть использован для форматирования данных в ситуациях, не подразумевающих реального выполнения ввода/вывода. Так, форматирование в памяти можно применять к символьным массивам и прочим структурам.

В файле iostream имеется два параллельных класса: **streambuf** и **ios**. Оба они являются классами низкого уровня, и каждый выполняет свой круг задач.

Класс streambuf обеспечивает общие правила буферизации и обработки потоков в тех случаях, когда не требуется значительного форматирования этих потоков. Класс streambuf представляет собой базовый класс, используемый другими классами из iostream. Большинство функций-членов streambuf являются встраиваемыми (inline) для обеспечения максимальной эффективности. Классы strstreambuf и filebuf являются производными от streambuf.

Класс ios (и, следовательно, производные от него классы) содержит указатель на streambuf.

Класс ios имеет два производных класса: istream (для ввода) и ostream (для вывода). Другой класс, iostream, является производным классом сразу от istream и ostream вследствие множественного наследования:

```
class ios; class istream: virtual public ios;
class ostream: virtual public ios; class
iostream: public istream, public ostream;
```

Кроме того, существует три класса _withassign, являющихся производными классами от istream, ostream и iostream:

class istream_withassign: public istream; class
ostream_withassign: public ostream; class
iostream_withassign: public iostream;

26.1. Классы потоков

Класс ios содержит переменные состояния для интерфейса с streambuf и обработки ошибок.

Класс istream поддерживает как форматированные, так и неформатированные преобразования потоков символов, извлекаемых из streambuf.

Класс ostream поддерживает как форматированные, так и неформатированные преобразования потоков символов, помещаемых в streambuf.

Класс iostream объединяет классы istream и ostream для двунаправленных операций, в которых один поток действует и как источник, и как приемник.

Производные классы _withassign отвечают за четыре предопределенных стандартных потока (cin, cout, cerr и clog), которые описываются в следующем разделе. Классы _withassign добавляют к соответствующим базовым классам операции присваивания следующим образом: class istream_withassign: public istream { istream_withassign(); istream& operator = (istream&); istream& operator = (streambuf*); } и аналогично для ostream_withassign и iostream_withassign.

Классом потока называется любой класс, производный от классов istream и ostream.

26.2. Стандартные потоки

Выполнение любой программы C++ начинается с открытыми потоками cin, cout, cerr и clog, объявленными как объекты классов _withassign в файле iostream следующим образом:

extern istream_withassign cin; extern
ostream_withassign cout; extern
ostream_withassign cerr; extern
ostream_withassign clog;

Их конструкторы вызываются всякий раз при включении iostream, но фактическая инициализация выполняется только один раз.

Все эти предопределенные стандартные потоки по умолчанию связаны с терминалом.

Четыре стандартных потока предназначены:

- для cin – стандартного ввода;
- cout – стандартного вывода;
- cerr – стандартного вывода ошибок;
- clog – полностью буферизованного вывода ошибок.

В табл. 5 приведено предназначение классов потокового вводавывода.

Таблица 5

Назначение классов потокового ввода-вывода

ios	Потоковый базовый класс
Потоковые классы ввода	
istream	Потоковый класс общего назначения для ввода, являющийся базовым классом для других потоков ввода
ifstream	Потоковый класс для ввода из файла
istream with assign	Потоковый класс ввода для cin
istrstream	Потоковый класс для ввода строк
Потоковые классы вывода	
ostream	Потоковый класс общего назначения для вывода, являющийся базовым классом для других потоков вывода
ofstream	Потоковый класс для вывода в файл
ostream_withassign	Потоковый класс ввода для cout, cerr, and clog
ostrstream	Потоковый класс для вывода строк
Потоковые классы ввода-вывода	
iostream	Потоковый класс общего назначения для вводавывода, являющийся базовым классом для других потоков ввода-вывода
fstream	Потоковый класс для ввода-вывода в файл
sir stream	Потоковый класс для ввода-вывода строк
stdiostream	Класс для ввода-вывода в стандартные файлы вводавывода
Классы буферов для потоков	
Streambuf	Абстрактный базовый класс буфера потока
filebuf	Класс буфера потока для дисковых файлов

s trs treambuf	Класс буфера потока для строк
stdiobuf	Класс буфера потока для стандартных файлов вводавывода

Предназначение почти всех классов следует из их названия. Классы группы _withassign являются производными соответствующих потоковых классов без этого окончания. Они перегружают операцию присваивания, что позволяет изменять указатель на используемый классом буфер.

Потоки ввода-вывода C++ предоставляют некоторые преимущества по сравнению с функциями ввода-вывода библиотеки C.

Безопасность типов. Сравним вызов функций библиотеки C и использование стандартных потоков C++. Вызов функции printf() выглядит следующим образом:

```
#include <stdio.h>
. . .
int n = 12;
char name[] = "Вывод строки на экран\n"; printf("%d
%s", i, name);
```

Этот вызов приведет к следующей правильной печати:

```
12  Вывод строки на экран
```

Но если по невнимательности поменять местами аргументы для printf(), ошибка обнаружится только во время исполнения программы. Может произойти все что угодно – от странного вывода до краха системы. Этого не может случиться в случае использования стандартных потоков:

```
#include <iostream> cout << i
<<' ' << name << '\n';
```

Так как имеются перегруженные версии операции сдвига operator <<(), правая операция всегда будет выполнена. Функция cout << i вызывает operator << (int), а cout << name вызывает operator << (const char*). Следовательно, использование стандартных потоков является безопасным по типам данных.

Расширяемость для новых типов. Другим преимуществом стандартных потоков C++ является то, что определенные пользователем типы данных могут быть без труда в них встроены. Рассмотрим класс Data, данные которого необходимо печатать:

```
struct Data {int x; char* y; };
```

Все, что нужно сделать, это переопределить операцию << для нового типа Data. Соответствующая функция operator<<() может быть реализована так:

```
ostream &operator<<(ostream & out, const Data & p){
return out << p.x << ' ' << p.y;
}
```

После этого станет возможно осуществлять вывод:

```
#include <iostream> struct
Data {int x; char* y;
Data (int x, char* y){this -> x = x;  this -> y = y;}
};
ostream &operator<<(ostream & out, const Data & p){
return out << p.x <<'  '<< p.y;
}
void main( ){ Data
p(1, "Error  "); cout
<< p << '\n';
}
```

26.3. Операции помещения и извлечения из потока

Вывод в поток выполняется с помощью ***операции вставки*** (в поток), которая является перегруженной операцией сдвига влево <<. Левым ее операндом является объект потока вывода. Правым операндом может являться любая переменная, для которой определен вывод в поток (то есть переменная любого встроенного типа или любого определенного пользователем типа, для которого она перегружена). Например, оператор **cout << "Hello!\n"**; приводит к выводу в предопределенный поток **cout** строки “Hello!“.

Операция << возвращает ссылку на объект типа ostream, для которого она вызвана. Это позволяет строить цепочки вызовов операции вставки в поток, и эти операции выполняются слева направо:

```
int i = 5; double d = 2.08; cout << "i = "
<< i << ", d = " << d << '\n';
```

Эти операторы приведут к выводу на экран следующей строки:

```
i = 5, d = 2.08
```

Операция вставки в поток поддерживает следующие встроенные типы данных: char, short, int, long, char* (рассматриваемый как строка), float, double, long double, void*:

```
ostream& operator << (short n);
ostream& operator << (unsigned short n);
ostream& operator << (int n); ostream&
operator << (unsigned int n); ostream&
operator << (long n); ostream& operator
<< (unsigned long n) ; ostream&
operator << (float f); ostream& operator
<< (double f); ostream& opera to r <<
(long double f) ; ostream& operator <<
(const void *p);
```

Целочисленные типы форматируются в соответствии с правилами, принятыми по умолчанию, если они не изменены путем установки различных флагов форматирования. Тип void* используется для отображения адреса:

```
int i;
// Отобразить адрес в 16-ричной форме:  cout
<< &i;
```

Отметим, что переопределение не изменяет нормального приоритета выполнения операции <<, поэтому можно записать cout << "sum = " << x + y << "\n";

без круглых скобок.

Однако в случае cout

```
<< (x & y) << "\n";
```

круглые скобки нужны.

Для ввода информации из входного потока используется ***операция извлечения***, которой является перегруженная операция сдвига вправо >>. Левым операндом операции >> является объект класса istream, который также является и результатом операции. Это позволяет строить цепочки операций извлечения из потока, выполняемых слева направо. Правым операндом может быть любой тип данных, для которого определен поток ввода.

```
istream& operator >> (short& n);
istream& operator >> (unsigned short& n);
```

```
istream& operator >> (int& n); istream&
operator >> (unsigned int& n); istream&
operator >> (long& n); istream& operator
>> (unsigned long& n); istream& operator
>> (float& f); istream& operator >>
(double& f); istreaai& operator >> (long
double& f); istream& operator >> (void*&
p);
```

По умолчанию операция >> пропускает символы-заполнители (по умолчанию – пробельные символы), затем считывает символы, соответствующие типу заданной переменной. Пропуск ведущих символов-заполнителей устанавливается специально для этого предназначенным флагом форматирования. Рассмотрим следующий пример:

```
int i;
double  d;  cin
>> i >> d;
```

Последний оператор приводит к тому, что программа пропускает ведущие символы-заполнители и считывает целое число в переменную i. Затем она игнорирует любые символы-заполнители, следующие за целым числом, и считывает переменную с плавающей точкой d.

Для переменной типа char* (рассматриваемой как строка) оператор >> пропускает символы-заполнители и сохраняет следующие за ними символы, пока не появится следующий символ-заполнитель. Затем в указанную переменную добавляется нуль-символ '\0'.

26.4. Форматирование потока

Для управления форматированием ввода-вывода предусмотрены три вида средств: ***форматирующие функции, флаги*** и ***манипуляторы***. Все эти средства являются членами класса ios и потому доступны для всех потоков.

Рассмотрим вначале ***форматирующие функции-члены***. Их всего три: width(), precision() и fill().

По умолчанию при выводе любого значения оно занимает столько позиций, сколько символов выводится. Функция width() позволяет задать минимальную ширину поля для вывода значения. При вводе она задает максимальное число читаемых символов. Если выводимое значение имеет меньше символов, чем заданная ширина поля, то оно дополняется

символами-заполнителями до заданной ширины (по умолчанию – пробелами). Если же выводимое значение имеет больше символов, чем ширина отведенного ему поля, то поле будет расширено до нужного размера. Эта функция имеет следующие прототипы: int width(int wide); int width() const;

Функция с первым прототипом задает ширину поля wide, а возвращает предыдущее значение ширины поля. Функция со вторым прототипом возвращает текущее значение ширины поля. По умолчанию она равна нулю, то есть вывод не дополняется и не обрезается. В ряде компиляторов после выполнения каждой операции вывода значение ширины поля возвращается к значению, заданному по умолчанию.

Функция precision() позволяет узнать или задать точность (число выводимых цифр после десятичной точки), с которой выводятся числа с плавающей точкой. По умолчанию числа с плавающей точкой выводятся с точностью, равной шести цифрам. Функция precision () имеет следующие прототипы:

int precision(int prec); int
precision() const;

Функция с первым прототипом устанавливает точность в prec и возвращает предыдущую точность. Функция со вторым прототипом возвращает текущую точность.

Функция fill() позволяет прочесть или установить символзаполнитель. Она имеет следующие прототипы:

char fill(char type ch); char
fill() const;

Функция с первым прототипом устанавливает ch в качестве текущего символа-заполнителя и возвращает предыдущий символ-заполнитель. Функция со вторым прототипом возвращает текущий символ-заполнитель.

По умолчанию в качестве символа-заполнителя используется пробел.

Рассмотрим пример программы, в котором используются форматирующие функции:

```
void main(){ double x;
cout.precision(4); cout.fill('0');
cout << "   x     sqrt(x)   x^2\n\n";
for (x = 1.0; x < 6.5; x++){
cout.width(7); cout << x << " ";
cout.width(7); cout << sqrt(x) <<
```

```
" "; cout.width(7); cout << x * x
<< '\n';
}
}
```

Эта программа выводит на экран небольшую таблицу значений переменной x, ее квадратного корня и квадрата:

x	sqrt(x)	x^2
0000001	0000001	0000001
0000002	01.4142	0000004
0000003	01.7321	0000009
0000004	0000002	0000016
0000005	02.2361	0000025
0000006	02.4495	0000036

С каждым потоком связан набор ***флагов***, которые управляют форматированием потока. Они представляют собой битовые маски, которые определены в классе ios как данные перечисления.

Флаги форматирования и их назначение приведены в табл. 6.

Таблица 6

Флаги форматирования и их назначение

Флаг	Назначение
hex	Значения целого типа преобразуются к основанию 16 (как шестнадцатеричные)
dec	Значения целого типа преобразуются к основанию 10
oct	Значения целого типа преобразуются к основанию 8 (как восьмеричные)
fixed	Числа с плавающей точкой выводятся в формате с фиксированной точкой (то есть nnn.ddd)
scientific	Числа с плавающей точкой выводятся в так называемой научной записи (то есть n.xxxEyy)
showbase	Выводится основание системы счисления в виде префикса к целому числовому значению (например, число 1FE выводится как 0x1FE)
showpos	При выводе положительных числовых значений выводится знак плюс
upper case	Заменяет определенные символы нижнего регистра на символы верхнего регистра (символ «e» при выводе чисел в научной нотации на «E» и символ «x» при выводе 16-ричных чисел на «X»)
left	Данные при выводе выравниваются по левому краю поля

right	Данные при выводе выравниваются по правому краю поля
internal	Добавляются символы-заполнители между всеми цифрами и знаками числа для заполнения поля вывода
skipws	Ведущие символы-заполнители (знаки пробела, табуляции и перевода на новую строку) отбрасываются
stdio	Потоки stdout, stderr очищаются после каждой операции вставки
unitbuf	Очищаются все выходные потоки после каждой операции вставки в поток
stdio	Очищаются stdout, stderr после каждой операции вставки в поток

Флаги left и right взаимно исключают друг друга. Флаги dec, oct и hex также взаимно исключают друг друга.

Прочесть текущие установки флагов позволяет функция-член flags() класса ios. Для этого используется следующий прототип этой функции:

long flags();

Функция flags() имеет и вторую форму, которая может использоваться для установки значений флагов. Для этого используется следующий прототип этой функции: long flags(long fmtfl);

В этом случае битовый шаблон копирует fmtfl в переменную, предназначенную для хранения флагов форматирования. Функция возвращает предыдущие значения флагов. Поскольку эта форма функции меняет весь набор флагов, она применяется редко. Вместо нее используется функция-член setf() класса ios, которая позволяет установить значение одного или нескольких флагов. Она имеет следующие прототипы:

long setf (long mask); long setf
(long fmtfl, long mask);

Первая функция-член неявно вызывает функцию flags (mask | flags()) для установки битов, указанных параметром mask, и возвращает предыдущие значения флагов. Второй вариант функции присваивает битам, указанным параметром mask, значения битов параметра fmtfl, а затем возвращает предыдущие значения флагов.

Например, следующий вызов функции setf() устанавливает для потока cout флаги hex и uppercase:

cout.setf(ios::hex | ios::uppercase);

В качестве второго параметра функции setf() можно использовать следующие константы, определенные в классе ios:

```
static  const long basefield;          // = dec | oct | hex static
const long adjustfield;          // = left | right | internal static
const long floatfield;           // = scientific | fixed
```

Сбросить установленные флаги можно с помощью функции-члена unsetf() класса ios, имеющей следующий прототип:

```
void unsetf(long mask);
```

Она сбрасывает флаги, заданные параметром mask. Следующий пример демонстрирует некоторые флаги:

```
double d = 1.321e9;
int  n = 1024; void
main(){ // Вывести
значения
cout << "d = " << d << '\n' << "n = " << n << '\n';
// Изменить флаги и вывести значения снова
cout.setf(ios::hex | ios::uppercase);
cout.setf(ios::showpos); cout <<
"d = " << d << '\n' ; cout << "n = "
<< n << '\n';}
```

При выполнении программа выводит на экран:

```
d = 1.321e+09
n = 1024  d =
+1.321E+09  n
= 400
```

Система ввода-вывода C++ предусматривает еще один способ форматирования потока. Этот способ основан на использовании *манипуляторов ввода-вывода.* Список манипуляторов и их назначение приведены в табл. 7. Манипуляторы ввода-вывода представляют собой просто вид функций-членов класса ios, которые, в отличие от обычных функций-членов, могут располагаться *внутри* инструкций вводавывода. В связи с этим ими пользоваться обычно удобнее.

Таблица 7

Манипуляторы ввода-вывода и их назначение

Манипулятор	Использование	Назначение
dec	Ввод-вывод	Устанавливает флаг dec
endl	Вывод	Вставляет символ новой строки и очищает буфер
ends	Вывод	Вставляет символ конца

flush	Вывод	Очищает буфер потока
hex	Ввод-вывод	Устанавливает флаг hex
oct	Ввод-вывод	Устанавливает флаг oct
resetiosflags (iosbase::long mask)	Ввод-вывод	Сбрасывает ios-флаги в соответствии с mask
Setbase (int base)	Ввод-вывод	Задает основание системы счисления для целых (8, 10, 16)
Setfill (int c)	Ввод-вывод	Устанавливает символзаполнитель
setiosflags (iosbase::long mask)	Ввод-вывод	Устанавливает ios-флаги в соответствии с mask

Окончание табл. 7

Манипулятор	Использование	Назначение
setprecision (int n)	Ввод-вывод	Устанавливает точность чисел с плавающей точкой
setw(int n)	Ввод-вывод	Устанавливает минимальную ширину поля
ws	Ввод	Устанавливает пропуск символов-заполнителей

За исключением setw(), все изменения в потоке, внесенные манипулятором, сохраняются до следующей установки.

Для доступа к манипуляторам с параметрами необходимо включить в программу стандартный заголовочный файл iomanip.h. При использовании манипулятора без параметров скобки за ним не ставятся, так как на самом деле он представляет собой указатель на функциючлен, который передается перегруженному оператору <<.

Рассмотрим пример, демонстрирующий использование манипуляторов.

```
#include <iostream>
#include <iomanip> #include <math.h> void main(){ double x; cout << setprecision(4); cout << setfill('0'); cout << "  x   sqrt(x)  x^2\n\n"; for (x = 1.0; x < 6.5; x++){ cout << setw(7) << x << "  "; cout << setw(7) << sqrt(x) << "  "; cout << setw(7) << x * x << "\n";
```

```
}
}
```

Этот пример функционально полностью эквивалентен приведенному ранее, но для управления форматом вывода использует манипуляторы, а не функции форматирования.

Манипулятор setw(), как и форматирующая функция width(), может помочь избежать переполнения строки-приемника при вводе символьных строк: const int SIZE = 50;

```
. . .
char array[SIZE];
cin >> setw(sizeof(array));  // Или cin.width(sizeof(array));
                             // Ограничивает число вводимых символов
. . .                        // и позволяет избежать выхода
                             // за границу массива.
cin >> array;
```

26.5. Файловый ввод-вывод с использованием потоков

Для осуществления операций с файлами предусмотрено три класса: **ifstream**, **ofstream** и **fstream**. Эти классы являются производными, соответственно, от классов **istream**, **ostream** и **iostream**. Поскольку эти последние классы, в свою очередь, являются производными от класса **ios**, классы файловых потоков наследуют все функциональные возможности своих родителей (перегруженные операции << и >> для встроенных типов, функции и флаги форматирования, манипуляторы и пр.). Для реализации файлового ввода-вывода нужно включить в программу заголовочный файл fstream.h.

Существует небольшое отличие между использованием предопределенных и файловых потоков. Файловый поток должен быть связан с файлом прежде, чем его можно будет использовать. С другой стороны, предопределенные потоки могут использоваться сразу после запуска программы, даже в конструкторах статических классов, которые выполняются раньше вызова функции main(). Можно позиционировать файловый поток в произвольную позицию в файле, в то время как для предопределенных потоков это обычно не имеет смысла.

Для создания файлового потока эти классы предусматривают следующие формы конструктора:

создать поток, не связывая его с файлом: ifstream(); ofstream(); fstream(); *создать поток, открыть файл и связать поток с*

файлом: ifstream(const char *name, ios::openmode mode = ios::in);
ofstream(const char* name, ios::openmode mode=ios::out | ios::trunc);
fstream(const char * name, ios::openmode mode = ios::in | ios::out);

Чтобы открыть файл для ввода или вывода, можно использовать вторую форму нужного конструктора fstream fs("FileName.txt");
или вначале создать поток с помощью первой формы конструктора, а затем открыть файл и связать поток с открытым файлом, вызвав функцию-член open(). Эта функция определена в каждом из классов потокового ввода-вывода и имеет следующие прототипы:

void ifstream::open(const char *name, ios::openmode mode = ios::in);
void ofstream::open
(const char * name, ios::openmode mode = ios::out | ios::trunc); void
fstream::open
(const char * name, ios::openmode mode = ios::in | ios::out);

Здесь name – имя файла, mode – режим открытия файла. Параметр mode является перечислением и может принимать значения, указанные в табл. 8.

Таблица 8

Режимы открытия и их назначение

Режим открытия	Назначение
ios::in	Открыть файл для чтения
ios::out	Открыть файл для записи
ios::ate	Начало вывода устанавливается в конец файла
ios::app	Окрыть файл для добавления в конец
ios::trunc	Усечь файл, то есть удалить его содержимое
ios::binary	Двоичный режим операций

Режимы открытия файла представляют собой битовые маски, поэтому можно задавать два или более режима, объединяя их побитовой операцией ИЛИ. В следующем фрагменте кода файл открывается для вывода с помощью функции open(): ofstream ofs; ofs.open ("FileName.txt");

Обратим внимание, что по умолчанию режим открытия файла соответствует типу файлового потока. У потока ввода или вывода флаг режима всегда установлен неявно. Например, для потока вывода в режиме добавления файла можно вместо оператора ofstream ofs ("FName.txt", ios::out | ios::app);
написать ofstream ofs ("FName.txt",
ios::app);

Между режимами открытия файла ios::ate и ios::app имеется небольшая разница.

Если файл открывается в режиме добавления, то вывод в файл будет осуществляться в позицию, начинающуюся с текущего конца файла, безотносительно к операциям позиционирования в файле. В режиме открытия ios::ate (от англ. *at end*) можно изменить позицию вывода в файл и осуществлять запись, начиная с нее. Для потоков вывода режим открытия эквивалентен ios::out | ios::trunc, то есть можно опустить режим усечения файла. Однако для потоков ввода-вывода его нужно указывать явно. Файлы, которые открываются для вывода, создаются, если они еще не существуют.

Если открытие файла завершилось неудачей, объект, соответствующий потоку, будет возвращать 0:

```
if(!ofs){ cout << "Файл не открыт\n";}
```

Проверить успешность открытия файла можно также с помощью функции-члена is_open(), имеющей следующий прототип: int is_open() const;

Функция возвращает 1, если поток удалось связать с открытым файлом.

Например:

```
if(!ofs.is_open()){ cout << "Файл не открыт\n"; return; }
```

Если при открытии файла не указан режим ios::binary, файл открывается в текстовом режиме. И после того, как файл успешно открыт, для выполнения операций ввода-вывода можно использовать операторы извлечения и вставки в поток. Для проверки, достигнут ли конец файла, можно использовать функцию ios::eof(), имеющую прототип int eof();

Завершив операции ввода-вывода, необходимо закрыть файл, вызвав функцию-член close() с прототипом void close():

```
ofs.close();
```

Закрытие файла происходит автоматически при выходе потокового объекта из области существования, когда вызывается деструктор потока.

Рассмотрим пример, демонстрирующий файловый ввод-вывод с использованием потоков:

```
#include <iostream>
#include
```

```
<fstream.h> void
main( ){ int m = 50;
// Открываем файл для вывода  ofstream
ofs("Test.txt");
if(!ofs){
cout << "Файл не открыт.\n";
return; } ofs <<
"Hello!\n" << m;
// Закрываем файл  ofs.close();
// открываем тот же файл для ввода ifstream file("Test.txt"); if(!file){
cout << "Файл не открыт.\n";
return; } char
str[80]; file >>
str >> m;
cout << str << " " << m << endl;
// Закрываем файл
file.close();
}
```

26.6. Неформатируемый ввод-вывод

Когда файл открывается в текстовом режиме, происходит следующее:

- *при вводе каждая пара символов* '\r' + '\n' *(возврат каретки + перевод строки) преобразуется в символ перевода строки ('\n')*;
- *при выводе каждый символ перевода строки ('\n') преобразуется в пару* '\r' + '\n' *(возврат каретки + перевод строки)*.

Это не всегда удобно. Если необходимо использовать файл вывода для последующего ввода в программу (возможно, другую), лишние байты информации ни к чему. С этой целью система ввода-вывода предоставляет возможность осуществления неформатируемого ввода-вывода, то есть записи и чтения двоичной информации (иногда говорят – сырых данных). Для осуществления ввода-вывода в двоичном режиме нужно включить флаг ios::binary в параметр open_mode, передаваемый конструктору потока или функции open(). Чтение двоичной информации из файла осуществляется функцией read(), которая имеет следующие прототипы:

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

Здесь параметр s задает буфер для считывания данных, а параметр n – число читаемых символов.

Запись двоичных данных осуществляет функция-член write():

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

Эта функция получает n символов из буфера, адрес которого задан параметром s, и вставляет их в поток вывода. Рассмотрим пример:

```
#include<iostream>
#include<fstream.h>
void main(){ int x =
255;
char str[80] = "Тестирование двоичного ввода-вывода.";
// Открываем файл для вывода в двоичном режиме ofstream
ofs("Test.txt");
if(!ofs){ cout << "Файл не открыт.\n"; return;
}
ofs.write((char*)&x, sizeof(int));
ofs.write((char*)&str, sizeof(str)); ofs.close();
// Открываем файл для вывода в двоичном режиме ifstream
ifs("Test.txt");
if(!ifs){ cout << "Файл не открыт.\n"; return;
}
ifs.read((char*)&x, sizeof(int));
ifs.read((char*) str, sizeof(str)); cout
<< x << '\n' << str << '\n';
}
```

26.7. Часто применяемые функции

Помимо уже описанных функций, библиотека ввода-вывода C++ содержит широкий набор различных функций. Здесь мы приведем лишь некоторые, наиболее часто употребляемые из них. Большинство этих функций используется для неформатируемого ввода-вывода.

Для извлечения символа из потока можно использовать функциючлен **get**() потока istream. Она имеет следующие прототипы:

```
int get();
```

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

Функция get() в первой форме возвращает код прочитанного символа или -1, если встретился конец файла ввода (ctrl/z). Приведем пример использования функции get ():

```
#include <iostream>
void main(){ char
ch;
cout << "Введите число. "
<< "Для завершения ввода нажмите <ENTER>:";
while(cin.get(ch)){ // Проверка на код клавиши <ENTER>  if(ch
== '\n') break;
}
return; }
```

Для вставки символа в поток вывода используется функция put() с прототипом ostreaiu& put(char ch). Функция get() может также использоваться для чтения строки символов. В этом случае используются ее варианты, в которых эта функция извлекает из входного потока символы в буфер str, пока не встретится символ-ограничитель *delim* (по умолчанию – перевод строки) или не будет извлечено (len - 1) символов, или не будет прочитан признак конца файла. Сам символ-ограничитель не извлекается из входного потока.

Ввиду того, что функция get() не извлекает из входного потока символограничитель, она используется редко. Гораздо чаще используется функция **getline**(), которая извлекает из входного потока символ-ограничитель, но не помещает его в буфер. Она имеет следующие прототипы:

```
istream& getline (char* str, int len, char delim); istream&
getline(char * str, int len); Здесь параметры имеют те же
назначения, что и в функции get().
```

Функция **gcount**() (с прототипом int gcount()const;) возвращает число символов, извлеченных из потока последней операцией неформатируемого ввода (то есть функцией get(), getline() или read()).

Рассмотрим пример, в котором используются две последние функции:

```
#include <iostream> void
main(){ char *name; int len =
100; int count = 0; name = new
char[len]; cout << "Введите
свое имя: "; cin.getline(name,
len); count = cin.gcount();
```

// Уменьшаем значение счетчика на 1, так как // getline() не помещает ограничитель в буфер. cout << "\nЧисло прочитанных символов: " << count - 1;} Для того, чтобы пропустить при вводе несколько символов, используется функция **ignore**():

istream & ignore(int n = l, int delim = EOF); Эта функция игнорирует вплоть до n символов во входном потоке. Пропуск символов прекращается, если она встречает символограничитель, которым по умолчанию является символ конца файла.

Символ-ограничитель извлекается из входного потока.

Функция **peek**(), имеющая прототип int peek(), позволяет «заглянуть» во входной поток и узнать следующий вводимый символ. При этом сам символ из потока не извлекается.

С помощью функции **putback**() (с прототипом istream &putback(char ch);) можно вернуть символ ch в поток ввода.

При выполнении вывода данные не сразу записываются в файл, а временно хранятся в связанном с потоком буфере, пока он не заполнится. Функция **flush**() позволяет вызвать принудительную запись в файл до заполнения буфера.

Она неявно используется манипулятором **endl**. Этот манипулятор вставляет в поток символ перевода строки и очищает буфер. Таким образом, оператор cout << endl;

эквивалентен следующим:

```
cout << '\n'; cout.flush();
```

Функция **rdbuf**() позволяет получить указатель на связанный с потоком буфер. Эта функция имеет прототип streambuf * rdbuf() const;

Наконец, функция **setbuf**() с прототипом void setbuf(char * buf, int n) позволяет связать с потоком другой буфер. Здесь buf – указатель на другой буфер длины n.

26.8. Файлы с произвольным доступом

Произвольный доступ в системе ввода-вывода реализуется с помощью функций seekg() и seekp(), используемых для

позиционирования, соответственно, входного и выходного потока. Каждая из них имеет по два прототипа:

```
istream& seekg(long pos); istream&
seekg(long pos, seek_dir dir);
ostream& seekp(long offset);
ostream& seekp(long offset, seek_dir
dir);
```

Здесь параметр pos задает абсолютную позицию в файле относительно начала файла. Параметр offset задает смещение в файле, а параметр dir – направление смещения, которое может принимать значения в соответствии с определением из класса ios: enum seek_dir { beg, cur, end };

Здесь константы перечисления определяют:
ios::beg – смещение от начала файла; ios::cur –
смещение относительно текущей позиции; ios::end
– смещение от конца файла.

С каждым потоком связан указатель позиционирования, который изменяет свое значение в результате операции ввода или вывода. Для выполнения операций произвольного доступа файл должен открываться в двоичном режиме.

Получить текущее значение позиции в потоке ввода или вывода можно с помощью функций tellg() и tellp() соответственно. Эти функции имеют следующие прототипы:

```
long tellg (); long
tellp();
```

Следующий пример демонстрирует возможность позиционирования потока ввода информации:

```
#include <iostream> #include
<fstream.h>
void main(int argc, char* argv[]){ int size = 0;
if(argc > 1){ const char *FileName = argv[1];
ofstream of;
of.open( FileName, ios::binary );
for(int i = 0; i<100; i++) of.put ((char)(i + 27)); of.close();  ifstream
file;
file.open(FileName, ios::in | ios::binary);
if(file){file.seekg(0, ios::end); size =
file.tellg();
```

```
if(size < 0){ cerr << FileName << " не найден. "; return;
}
cout << FileName << " size = " << size << endl;}
} else cout << "Вы не задали имя
файла.";}
```

Программа выводит на экран длину заданного файла.

26.9. Опрос и установка состояния потока

Класс ios поддерживает информацию о состоянии потока после каждой операции ввода-вывода. Текущее состояние потока хранится в объекте типа iostate, который объявлен следующим образом:

typedef int iostate;

Состояния потока являются элементами перечислимого типа io_state, который может иметь значения, представленные в табл. 9.

Таблица 9

Состояния потока и их значения

Состояние	Значение
Goodbit	Ошибок нет
Eofbit	Достигнут конец файла
Failbit	Ошибка форматирования или преобразования
Badbit	Серьезная ошибка

Для опроса и установки состояния потока можно использовать функции-члены класса ios. Имеется два способа получения информации о состоянии операции ввода-вывода. Во-первых, можно вызвать функцию rdstate(), имеющую прототип iostate rdstate() const.

Функция возвращает состояние операции ввода-вывода. Во-вторых, можно воспользоваться одной из следующих функций-членов: int good() const; int eof() const; int fail() const; int bad() const;

Каждая из этих функций возвращает 1, если установлен соответствующий бит состояния (точнее, функция fail() возвращает 1, если установлен бит failbit или badbit).

Если прочитано состояние, которое сигнализирует об ошибке, его можно сбросить с помощью функции clear():

void clear(iostate state = ios::goodbit);

Установить нужное состояние можно с помощью функции setsfate():

void setstate(iostate state);

Кроме перечисленных функций, класс ios содержит функцию приведения типа operator void*() const; (она возвращает NULL, если установлен бит badbit) и перегруженный оператор логического отрицания int operator !() const; (он возвращает 1, если установлен бит badbit). Это позволяет сравнивать выражения, в которые входит поток или его отрицание с нулем, то есть писать выражения вида while(!strm.eof()){

...

}

Следующий пример иллюстрирует получение информации о состоянии ввода-вывода.

```
#include <iostream> #include
<fstream> int main(int argc,
char* argv[]){ char c; if(argc >
1){  ifstream ifs(argv[1]);
if(!ifs){ cout << "Файл не открыт\n"; return 1; }
while (ifs.eof()){ifs.get(c); // Контроль
состояния потока  if(ifs.fail ()){cout <<
"Ошибка \n"; break;
} cout << c;
}
ifs.close();
}  return
0; }
```

В этом примере осуществляется ввод символов из файла, заданного в командной строке при запуске программы. Если при извлечении символов встречается ошибка, чтение прекращается и выводится сообщение об этом.

26.10. Переопределение операций извлечения и вставки

Одним из главных преимуществ потоков ввода-вывода является их расширяемость для новых типов данных. Можно реализовать операции извлечения и вставки для своих собственных типов данных. Чтобы избежать неожиданностей, ввод-вывод для определенных пользователем типов данных должен следовать тем же соглашениям, которые используются операциями извлечения и вставки для встроенных типов данных. Рассмотрим пример переопределения операций извлечения и вставки в поток для определенного пользователем типа данных, которым является следующий класс даты:

```
class Date { public:
Date(int d, int m, int y);
Date(const tm & t);
Date(); private: tm
tm_date;
};
```

Этот класс содержит член типа tm, который представляет собой структуру для хранения даты и времени, определенную в заголовочном файле time.h. Чтобы осуществить ввод-вывод пользовательского типа данных, какими являются объекты класса Date, нужно переопределить операции извлечения и вставки в поток для этого класса:

```
class Date {  tm
tm_date;
friend istream& operator >> (istreamfi &is, Date txt); friend
ostream& operator << (ostream& os, const Date& txt);
public:
Date(int d, int m, int y);
Date(tm t);
Date(); tm
tm_date;
friend istream& operator >> (istreamfi &is, Date txt); friend
ostream& operator << (ostream& os, const Date& txt); };
```

Реализуем операции извлечения и вставки для объектов класса Date.

Возвращаемым значением для операции извлечения (и вставки) должна являться ссылка на поток, чтобы несколько операций могли быть выполнены в одном выражении. Первым параметром должен быть поток, из которого будут извлекаться данные, вторым параметром – ссылка или указатель на объект определенного пользователем типа. Чтобы разрешить доступ к закрытым данным класса, операция извлечения должна быть объявлена как дружественная функция класса. Ниже приведена операция извлечения из потока для класса Date:

```
istream& operator >> (istream& is, Date& txt){
is >> txt.tm_date.tm_mday; is >>
txt.tm_date.tm_mon; is >>
txt.tm_date.tm_year; return is; }
```

Те же самые замечания верны и для операции вставки. Она может быть построена аналогично. Единственное отличие заключается в том, что в нее нужно передать константную ссылку на объект типа Date,

поскольку операция вставки не должна модифицировать выводимые объекты. Ниже приведена ее реализация для класса Date:

```
osiream& operator << (ostream& os, const Date& txt){
os << txt.tm_date.tm_mday <<'/'; os <<
txt.tm_date.tin_mon << '/'; os << txt.tm_date.tm_year;
return os; }
```

Следуя соглашениям о вводе-выводе для потоков, теперь можно осуществлять извлечение и вставку объектов класса Date следующим образом:

```
Date birthday(24, 10, 1985); cout
<< birthday << '\n';
```

или

```
Date date;
cout << "Пожалуйста, введите дату (день, месяц, год)\n";
cin >> date; cout << date << '\n';
```

Приведем теперь пример полностью:

```
#include <iostream>
#include <time.h>
class Date{
tm  tm_date;

public:
Date(int d, int m, int y){
tm_date.tm_mday = d; tm_date.tm_mon = m; tm_date.tm_year = y;
};
Date (tm t){tm_date = t;}; Date(){ tm_date.tm_mday = 01;
tm_date.tm_mon = 00; tm_date.tm_year = 00; } friend
ostream& operator << (ostream& os, const Date& txt); friend
istream& operator >> (istream& is, Date& txt);
};

istream& operator > >(istream& is, Date& txt){
is >> txt.tm_date.tm_mday;
is >> txt.tm_date.tm_mon ; is
>> txt.tm_date.tm_year ;
return is;
}
```

```
ostream& operator << (ostream& os,const Date& txt){
os << txt.tm_date.tm_mon << '/' ; os <<
txt.tm_date.tm_mday << '/' ; os << txt. tm_date.
tm_year ;
return os; } void
main(){ Date date;
cout << "Пожалуйста, введите дату (день, месяц, год)\n";
cin >> date; cout << date << '\n'; }
```

Приведем еще один пример, демонстрирующий переопределение операций извлечения и вставки в поток, на этот раз для структуры:

```
struct info{ char
*name; double val;
char *units; info(){ val
= 0; name = new
char[30]; units = new
char[30]; name[0] =
units[0] = 0;
} }; ostream& operator << (ostream& s, info &m){ // Вывод  info
в  s s << m.name <<"  "<< m.val <<"  "<< m.units << "\n"; return
s; }
```

Операция >> может быть переопределена следующим образом:

```
istream& operator >> (istream& s, info &m){ // Ввод в  info
s.width(30);  s >> m.name;
char c;
while((c = s.get())!= ' ' && c!= '\t' && c!= '\n'); s.putback(c);
s.width(30);   s
>> m.val;
s.width(30);   s
>> m.units;
return s;
}
```

Для считывания строки ввода, такой как «**Resistance 300 Ohm**», можно использовать следующую запись:

```
void main(){ info
m;
cout << "Введите наименование величины, ее значение \n";
```

```
cout << " и единицу измерения (и нажмите Enter.):\n"; cin
>> m;                  // Переопределенная операция >> cout
<< m;                  // Переопределенная операция <<
}
```

При выполнении этой программы диалог на экране монитора может выглядеть следующим образом:

```
Введите наименование величины, ее значение  и единицу измерения
(и нажмите Enter.):
Resistance 300 Ohm
Resistance 300 Ohm
```

26.11. Переадресация ввода-вывода

Можно переназначить имена cin или cout файловым потокам. Это позволяет легко проводить отладку ввода-вывода, переадресовывая ввод-вывод вместо файла на экран.

Следующий пример демонстрирует эту возможность:

```
#include <iostream>  #include
<fstream> int main(int argc,
char* argv[]){  char str[80];
// Создаем файловый поток:
ofstream  ofs;
cout << "Введите имя и фамилию:\n";
cin.getline(str, sizeof(str));
                        // Если в командной строке задан аргумент,
if(argc > 1){           // открываем файл с заданным именем.
ofs.open(argv[1]);

if(ofs)    // Если файл успешно открыт,  cout = ofs;   //
переадресовываем вывод.
}

cout << "Привет, " << str << '!' << endl; return
0;
}
```

Если при запуске программы в командной строке задано имя файла, то вывод осуществляется в этот файл, в противном случае – на экран терминала.