

Севастопольский государственный университет
Кафедра «Информационные системы»

Курс лекций по дисциплине
**«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ»**
(ООП)

Лектора: Пелипас Всеволод Олегович
Сметанина Татьяна Ивановна

Лекция 10

ПОТОКОВЫЕ КЛАССЫ.
ОШИБКИ ПОТОКОВ ВВОДА-ВЫВОДА.
ФАЙЛОВЫЙ ВВОД-ВЫВОД.
СТРОКОВЫЕ ПОТОКИ.

Потоки

Система ввода-вывода языка C++ основывается на концепции потока. **Поток** в C++ - это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. В качестве синонимов вывода данных используются термины **извлечение, прием, получение**, а синонимов ввода - **вставка, помещение, заполнение**.

В языке C++ вся передаваемая/принимаемая информация рассматривается как последовательность символов, поскольку любое двоичное представление может быть рассмотрено как неотображаемая последовательность байт.

Перенос данных от источника к приемнику чаще всего осуществляется через временное хранение в некоторой области памяти, называемой **буфером**.

Система ввода-вывода C++ состоит из четырех основных библиотек: **iostream, fstream, strstream** и **ostream**.

Потоки

Библиотека ***iostream*** обеспечивает ввод-вывод, связанный со стандартными потоками. Библиотека ***fstream*** поддерживает работу с файлами, ***stringstream*** - со строками символов, а ***ostream*** обеспечивает работу с консолью. Все эти библиотеки позволяют выполнять форматный ввод-вывод с контролем типов как для предопределенных, так и для определяемых пользователем типов данных с помощью перегруженных операций и других объектно-ориентированных методов.

Для поддержки потоков библиотека C++ содержит иерархию классов, построенную на основе двух базовых классов — ***ios*** и ***streambuf***.

Класс ***streambuf*** обеспечивает буферизацию потоков и их взаимодействие с физическими устройствами.

Стандартные потоки ввода-вывода

Класс **ios** предназначен для форматного ввода-вывода через класс **streambuf**. Из него производятся три класса **istream**, **ostream** и **iostream** соответственно для ввода, вывода и одновременного ввода-вывода.

В C++ predeterminedены четыре стандартных потока : **cin**, **cout**, **cerr** и **clog**. Эти потоки автоматически открываются при запуске программы, содержащей файл **iosrteam.h**:

cin – стандартный ввод (обычно с клавиатуры);

cout – стандартный вывод (обычно на экран);

cerr – вывод сообщений об ошибках на экран;

clog – буферизированный вывод сообщений об ошибках на экран.

Неформатный ввод-вывод

Компоненты-функции **put()** и **write()** класса **ostream** обеспечивают неформатный вывод данных в стандартные потоки.

ostream & put (char);

Функция **put()** позволяет вывод двоичных данных или отдельного символа, получаемого в качестве аргумента, в указанный поток, например:

```
int ch = 'a';    cout.put(ch);
```

выводит символ 'a' в поток **cout**.

Большие по размерам объекты выводятся с помощью функции **write()**:

ostream & write (char *p, int n);

где **p** указывает на выводимые данные; **n** - размер в байтах.

Их использование ограничено простыми примерами ввода-вывода, когда предъявляются жесткие требования к быстродействию и размеру абсолютного кода программы.

Неформатный ввод-вывод

Для неформатного ввода данных из стандартного потока в C++ служит компонента-функция **get()** класса **istream**:

istream & get(char *str, int max, int term = '\n');

Функция **get()** считывает символы из входного потока в массив **str** до тех пор, пока не будет прочитано **max-1** символов, либо пока не встретится символ, заданный терминатором (ограничителем) **term**. К прочитанным данным автоматически добавляется символ конца строки **'\0'**. По умолчанию значением терминатора является символ новой строки **'\n'**, который в **str** не считывается и из **istream** не удаляется.

Для корректной работы программы массив **str** должен иметь размер не менее **max** символов, например:

```
char s[81]; cin.get(s,81); // ввод строки с терминала
```

Функция **get()** в классе **istream** перегружена и имеет несколько реализаций, например,

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

позволяет читать символы в некоторый буфер потока **buf**.

Неформатный ввод-вывод

Для неформатного ввода блока данных служит функция **read()**, определенная в классе **istream**:

istream & read(char *p, int n);

где **p** указывает на вводимые данные, а **n** - размер данных в байтах

Пример: вводит и выводит непреобразованное значение последовательности символов:

```
void main(){  
char s[80]={0};  
cout << "Введите пять символов\n";  
cin.read(s, sizeof(s));  
cout << "Было введено: ";  
cout.write(s, sizeof(s)); }
```

Стандартные потоки **cin** и **cout** осуществляют последовательный символьный ввод-вывод. Функции же **read()** и **write()** наиболее эффективны при вводе-выводе двоичных блоков данных, поэтому их часто используют для работы с бинарными файлами.

Неформатный ввод-вывод

В библиотеке **iostream** имеются еще несколько полезных функций неформатного ввода-вывода:

istream & getline(char *str, int max, int term = '\n');

подобна функции **get()** за исключением того, что ограничивающий символ **term** извлекается, но не копируется в **str** (удобна для чтения строк);

int peek(); - возвращает следующий символ без извлечения из потока (просмотр вперед);

int gcount(); - возвращает число символов последнего извлечения;

istream & putback(char ch); - возвращает обратно во входной поток символ **ch**;

istream & ignore(int n = 1, int term = EOF); - пропускает **n** символов во входном потоке, останавливается, когда встретится **term**;

ostream & flush(); - разгружает поток (т.е. выводит содержимое) на связанное с ним устройство.

Форматный ввод-вывод

В языке C++ форматный вывод в поток выполняется с помощью перегруженной операции сдвига "<<" (оператором вставки или помещения); форматный ввод - ">>" (оператором извлечения или просто извлечением).

ostream& ostream::operator<<(Тип Op)

istream& istream::operator>>(Тип Op)

где типы: **char, signed и unsigned short, int, long, float, double, long double, char *(строка), void *(адрес).**

Первый параметр – поток, второй – данные указанных типов, результат – ссылка на тот же поток, что позволяет строить выражения для ввода и вывода. По умолчанию извлечение опускает пробельные символы ('\v', '\t', '\n' и пробел), а затем считывает символы, соответствующие типу объекта ввода. Это позволяет объединить в одном операторе несколько операций ввода, например:

```
int k;
```

```
float y;
```

```
cin >> k >> y;
```

Форматирование ввода и вывода

определяется форматизирующими флагами, представляющими собой биты числа типа **long int**, определенного в классе **ios** следующим образом: **enum**

```
{ skipws = 0x0001, // пропустить пробелы при вводе
  left  = 0x0002, // выполнять по левой гр. при выводе
  right = 0x0004, // выполнять по правой гр. при выводе
  interval=0x0008, // дополнить пробелами при выводе
  dec   = 0x0010, // преобразовать в десятичную с/с
  oct   = 0x0020, // преобразовать в восьмиричную с/с
  hex   = 0x0040, // преобразовать в шестнадцатир. с/с
  showbase=0x0080, // показывать основание с/с при выводе
  showpoint=0x0100, // показывать дес. точку при выводе
  uppercase=0x0200, // вывод шестн. цифр в верхнем р-ре
  showpos = 0x0400, // выводить + перед полож. числами
  scientific=0x0800, // вывод в формате с плав. точкой
  fixed   = 0x1000, // вывод в формате с фикс. точкой
  unitbuf = 0x2000, // стереть все потоки после вставки
  stdio   = 0x4000}; //стереть после вставки stdin, stdout
```

Форматирование ввода и вывода

Флаги вместе с другими управляющими полями объявлены в классе **ios**:

```
class ios
```

```
{private:
```

```
long x_flags;      // флаги
```

```
int x_width;      // ширина поля вывода
```

```
int x_precision; // число цифр дробной части
```

```
int x_fill; ...}   // символ-заполнитель при выводе
```

Для работы с полями используют специальные методы:

```
long setf(long flags);
```

функция возвращает предыдущую установку флагов и изменяет флаги, определенные в **flags**.

```
long unsetf(long flags);
```

функция возвращает предыдущую установку и сбрасывает флаги, определенные в **flags**.

```
long flags(void);
```

```
long flags(long bits);
```

функция определяет значения флагов без их изменения.

Форматирование ввода и вывода

Следующие три функции библиотеки **iostream** позволяют устанавливать ширину поля, заполняющий символ и число цифр, показываемых после запятой:

int width(int len);

char fill(char ch);

int precision(int num);

где **len** - ширина поля, **ch** - символ заполнения, **num** - число цифр после запятой в отображении числа с плавающей точкой. Функции возвращают предыдущие значения соответствующих параметров.

По умолчанию ширина поля равна нулю, т.е. будет выведено минимальное число символов, которыми может быть представлено выводимое значение. В избыточных позициях выводится символ заполнителя, заданный функцией **fill()**, по умолчанию — это пробел.

Если указанная ширина не достаточна для представления выводимого значения, то она будет проигнорирована и вывод выполняется как для нулевой ширины. Вызов функций **width()**, **fill()** и **precision()** без аргументов возвращает предыдущее значение соответствующих параметров без их изменения.

Пример

```
#include <iostream.h>
main() {
    cout.setf(ios::hex);
    cout.setf(ios::scientific);
    cout << 365 << " " << 365.78 << "\n";
    cout.precision(2);
    cout.width(10);
    cout << 365 << " " << 365.78 << "\n";
    cout.fill('*');          // cout.width(10);
    cout << 365 << " " << 365.78 << "\n";
}
```

```
16d 3.6578e+02
      16d 3.66e+02
16d 3.66e+02
```

```
16d 3.6578e+02
      16d 3.66e+02
*****16d 3.66e+02
```

Форматирование с помощью манипуляторов

Манипуляторы - это специальные функции, которые принимают в качестве аргументов ссылку на поток и возвращают ссылку на тот же поток.

Они могут объединяться в цепочку вместе с операторами ввода-вывода. Сами манипуляторы никаких действий по вводу или выводу не выполняют, однако, осуществляют "побочный" эффект, воздействуя на флаги формата и другие параметры ввода-вывода.

Таблица манипуляторов

Манипулятор	Операторы	Действие
dec	<<,>>	десятичное представление чисел
oct	<<,>>	восьмеричное представление чисел
hex	<<,>>	шестнадцатеричное представление чисел
ws	>>	извлечение пробельных символов
endl	<<	вставка символа новой строки '\0' и разгрузка потока
ends	<<	вставка в строку символа '\n' конца строки
flush	<<	очистка потока ostream
setbase(int n)	<<,>>	установка системы счисления с основанием n, где n - одно из {0,8,10,16}; ноль означает десятичную систему счисления при выводе и правила C для литералов целых чисел при вводе
setiosflags(long f)	<<,>>	установка флагов, указанных в f
resetiosflags(long f)	<<,>>	сброс флагов, указанных в f
setfill(int ch)	<<,>>	установка символа-заполнителя в ch
setprecision(int n)	<<,>>	установка числа цифр после запятой в представлении чисел с плавающей запятой
setw(int l)	<<,>>	установка ширины поля в l

Форматирование с помощью манипуляторов

```
cout << setw(10) << setprecision(2) << setfill('*') << 365.766;
```



```
*****365.77
```

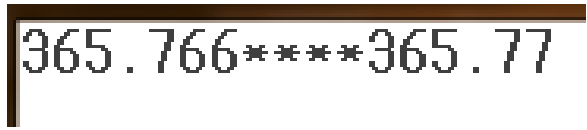
Создание собственных манипуляторов:

```
ostream & <имя_манипулятора>(ostream & stream)
{ // необходимый код
  return stream;
}
```

Пример:

```
#include <iomanip.h>
ostream & my_manip(ostream & stream){
stream << setw(10) << setprecision(2) << setfill('*');
return stream;
}
```

```
main() {
cout << 365.766 << my_manip << 365.766 << endl;
}
```



```
365.766*****365.77
```

Форматирование с помощью манипуляторов

Заказные манипуляторы являются полезными в двух случаях.

Во-первых, когда осуществляется вывод на устройство, которое не выполняет применяемые манипуляторы, например, плоттер. В этом случае создание собственных манипуляторов позволяет выполнять необходимые преобразования во время вывода.

Во-вторых, когда часто повторяется последовательность одних и тех же манипуляторов.

Определение собственных манипуляторов пользователя для ввода имеет следующую структуру:

```
istream & <имя_манипулятора>(istream & stream){  
// необходимый код  
return stream;  
}
```

Форматирование с помощью манипуляторов

Пример:

```
istream & manip_in_hex(istream & stream){  
    cout << "Введите число, используя шестнадцатеричный  
формат";  
    cin >> hex;  
    return stream;  
}
```

```
main(void){  
    int i;  
    cin >> manip_in_hex >> i;  
    cout << i << endl;  
}
```

```
Введите число, используя шестнадцатеричный формат  
F  
15
```

Ошибки потоков ввода-вывода

С каждым открытым потоком в C++ связывается перечислимая переменная **io_state**, определяющая биты состояния потока. Она объявлена в классе **ios** и может рассматриваться как целая величина:

public:

enum io_state{

goodbit = 0x00, // если бит не установлен, то ошибок нет

eofbit = 0x01, // обнаружен конец файла

failbit = 0x02, // сбой в последней операции ввода-вывода

badbit = 0x04, // попытка недопустимой операции

hardfail = 0x80 // в потоке невозстанавливаемая ошибка

};

В случае установки **eofbit** игнорируются попытки выполнить операции извлечения; бит **failbit** может быть сброшен и продолжено использование потока; после сброса **badbit** не всегда можно восстановить работоспособность потока; перед сбросом **hardfail** требуется установить причину, вызвавшую ошибку.

Ошибки потоков ввода-вывода

После того как для некоторого потока возникло состояние ошибки, все попытки ввода-вывода будут игнорироваться до тех пор, пока не будет устранена причина, вызвавшая ошибку, а биты ошибки не сброшены с помощью функции **clear()**, например:

in.clear(0); // очистка всех бит ошибок

in.clear(ios::eofbit); // очистка бита **eofbit**

Хорошим стилем программирования считается проверка состояния ошибки в наиболее ответственных точках программы. Это можно выполнить с помощью следующих функций, возвращающих ненулевые значения, если:

good() - не было ошибки;

eof() - обнаружен конец файла;

fail() - был установлен один из битов **failbit**, **badbit**, **hardfail**;

bad() - был установлен бит **badbit** или **hardfail**.

Пример

double d; cin>>d;

/* 1) 2 - преобразуется к d=2.0

2) 2А - d=2.0 - значение сформировано, но **<А>** еще осталось в буфере потока и ждет приема **char**

3) 3,3 вместо **3.3** - **d=3.0**, но запятая и вторая **3** осталась в буфере ввода и ждет ввода

4) вводим ААА - поток испорчен, вырабатывается флаг ошибки, пользоваться **d** нельзя! Значение **d** не изменилось!!! Весь последующий ввод (**cin>>**) будет проигнорирован! ***/**

if(!cin) { **/*** Сюда попадем, если только поток ввода Ваш ввод никаким образом проинтерпретировать не может (случай 4). Для того, чтобы программу можно было выполнять дальше необходимо:

1) сбросить ошибки */****

cin.clear(); **//** иначе весь последующий ввод будет проигнорирован
} else { **/*используем d*/**

cin.ignore(MAXINT, '\n'); **// 2) очистить буфер ввода**

cin>>d;

Ошибки потоков ввода-вывода

Текущее состояние ошибки можно получить с помощью функции **rdstate()**, которая возвращает номер бита ошибки, например :

```
cout << "Состояние потока cin: " << cin.rdstate() << endl;  
cout << "Состояние потока cout: " << cout.rdstate() << endl;  
cout << "Состояние потока cerr: " << cerr.rdstate() << endl;  
cout << "Состояние потока clog: " << clog.rdstate() << endl;
```

Пример:

```
double d;  
cin>>d;  
cout << "Состояние потока cin: " << cin.rdstate() << endl;
```

```
7.0
```

```
Состояние потока cin: 0
```

```
aaa
```

```
Состояние потока cin: 2
```

Ошибки потоков ввода-вывода

Кроме того, в классе **ios** имеются перегруженные операции

int operator !();

operator void *();

Операция **void *()** определяет преобразование потока в указатель, который будет равен нулю в случае установления бит **failbit**, **badbit** или **hardfail** и ненулевому значению в противном случае.

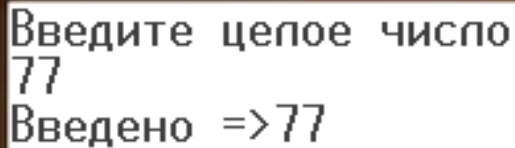
Операция **!"** наоборот возвращает ненулевое значение, если установлен один из бит **failbit**, **badbit** или **hardfail**, и возвращает нулевое значение в противном случае.

Это позволяет рассматривать в логических выражениях в качестве переменной непосредственно сами потоки ввода-вывода, например:

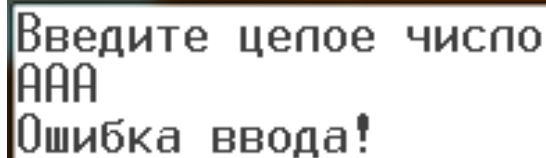
Ошибки потоков ввода-вывода

Пример:

```
int main(){
    int x;
    if (!cin) return -1; // ошибка ввода!
    cout << "Введите целое число\n";
    if (cin >> x)
        cout << "Введено" << x << endl; // все в порядке!
    else{
        cout << "Ошибка ввода!\n"; // ошибка ввода!
        return -1;
    }
    return 0;
}
```



Введите целое число
77
Введено =>77



Введите целое число
AAA
Ошибка ввода!

Достоинства и недостатки

Основным **преимуществом** потоков по сравнению с функциями ввода/вывода, унаследованными из библиотеки C, является **контроль типов**, а также расширяемость, то есть **возможность работать с типами, определенными пользователем** (для этого требуется переопределить операции потоков).

К **недостаткам** потоков можно отнести **снижение быстродействия** программы, которое в зависимости от реализации компилятора может быть весьма значительным.

Файловый ввод-вывод

Под **файлом** обычно подразумевается именованная информация на внешнем носителе, например, на жестком или гибком магнитном диске. Устройства такие, как дисплей, клавиатуру и принтер рассматривают как частные случаи файлов.

По способу доступа файлы можно разделить на

- **последовательные**, чтение и запись в которых производятся последовательно от начала, байт за байтом, и
- файлы с **произвольным** доступом, допускающие чтение и запись в указанную позицию.

Стандартная библиотека содержит три класса для работы с файлами: **ifstream** — класс входных файловых потоков;

ofstream — класс выходных файловых потоков;

fstream — класс двунаправленных файловых потоков.

Эти классы являются производными от классов **istream**, **ostream** и **iostream** соответственно, поэтому они наследуют перегруженные операции $<<$ и $>>$, флаги форматирования, манипуляторы, методы, состояние потоков и т. д.

Файловый ввод-вывод

При работе с файлами в программе должны присутствовать следующие операции:

- создание потока;
- открытие потока и связывание его с файлом;
- обмен (ввод/вывод);
- уничтожение потока;
- закрытие файла.

Каждый класс файловых потоков содержит конструкторы, с помощью которых можно создавать объекты этих классов различными способами:

- *конструкторы без параметров* создают объект соответствующего класса, не связывая его с файлом:

ifstream();

ofstream();

fstream();

Файловый ввод-вывод

- *конструкторы с параметрами* создают объект соответствующего класса, открывают файл с указанным именем и связывают файл с объектом:

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

Пример:

```
#include <fstream.h>  
int main(){ //...  
ifstream f1("1.txt", ios::in);  
ofstream f2("2.txt", ios::out | ios::trunc);  
fstream f3("3.txt", ios::in | ios::out);
```

Вторым параметром конструктора является режим открытия файла. Если установленное по умолчанию значение не устраивает программиста, можно указать другое, составив его из битовых масок, определенных в классе **ios**:

Файловый ввод-вывод

```
enum open_mode{  
in      = 0x01 // Открыть для чтения  
out     = 0x02 // Открыть для записи  
ate     = 0x04 // Установить указатель на конец файла  
app     = 0x08 // Открыть для добавления в конец  
trunc   = 0x10 // Если файл существует, удалить  
nocreate= 0x20      // Если файл не существует, выдать ошибку  
noreplace= 0x40     // Если файл существует, выдать ошибку  
binary  = 0x80      // Открыть в двоичном режиме  
};
```

Открыть файл в программе можно с использованием либо конструкторов, либо метода **open**, имеющего такие же параметры, как и в соответствующем конструкторе, например:

```
ifstream f1 ("input.txt", ios::in|ios::nocreate);  
if(!f1){ cout << "Невозможно открыть файл для чтения";  
        return 1; }
```

Файловый ввод-вывод

//...

ofstream f2;

f2.open("output1.txt"); // Использование метода **open**

if (!f2){

cout << "Невозможно открыть файл для записи";

return 1;}

void open (const char* filename, int mode);

где **mode** - режим ввода/вывода:

in - открыть поток для ввода;

out - открыть поток для вывода;

ate - установить указатель потока на конец файла (отсчет позиции с конца),

app - открыть поток для добавления,

trunc - удалить содержимое файла, если он уже существует,

binary - открыть в двоичном режиме.

Файловый ввод-вывод

Примеры:

fstream f; // объявление переменной-потока без открытия файла

а)

f.open("simple.txt", ios::in); // открыть поток для ввода

б)

f.open("simple.txt", ios::out|ios::trunc); /* открыть поток для вывода и стереть файл с указанным именем, если он существует */

в)

f.open("simple.txt", ios::in|ios::out|ios::binary); /* открыть двоичный файл для ввода и вывода */

Файловый ввод-вывод

Закрытие файла

```
void close();
```

Пример :

```
fstream f;  
f.open("simple.txt", ios:: in);  
/*Обработка компонентов файла*/  
f.close();
```

Чтение и запись выполняются либо с помощью операций чтения и извлечения, аналогичных потоковым классам, либо с помощью методов классов.

Пример файлового ввода-вывода

Опишем программу, которая выводит на экран содержимое файла:

```
int main(){
char text[81], buf[81];
cout << "Введите имя файла:";
cin >> text;
ifstream f(text, ios::in|ios::nocreate); // для чтения
if (!f){ cout << "Ошибка открытия файла"; return 1; }
while (!f.eof()){
    f.getline(buf, 80);
    cout << buf << endl;
}
return 0;
}
```

Для закрытия потока определен метод **close()**, но поскольку он неявно выполняется деструктором, явный вызов необходим только тогда, когда требуется закрыть поток раньше конца его области видимости.

Пример использования write()

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
int main(){
    const int MAX = 80;
    char buff[MAX+1] = "Hello World!";
    int len = strlen (buff) + 1;
    fstream f;
    f.open("CALC.DAT", ios::out|ios::binary);
    f.write((const char*) &len, sizeof(len));
    f.write((const char*) buff, len);
    f.close();
}
```

Пример использования read()

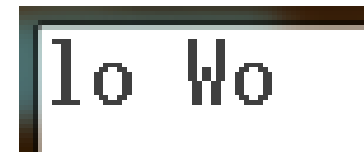
```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main(){
    const int MAX = 80;
    char buff [MAX+1];
    int len;
    fstream f;
    f.open("CALC.DAT", ios::in|ios::binary);
    f.read((char *) &len, sizeof(len));
    f.read((char *) buff, len);
    cout << len << ' ' << buff << endl;
    f.close();
}
```



```
13 Hello World!
```

Пример прямого доступа

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
using namespace std;
int main() {
    char bbb[80]={0}, buff[80] = "Hello World!";
    fstream f,f1;
    f.open("CALC.DAT", ios::out | ios::binary);
    f.write(buff,strlen(buff));
    f.close();
    f1.open("CALC.DAT", ios::in|ios::binary);
    f1.seekg(3); // переставить указатель чтения через 3 байта на 4
    f1.read(bbb, 5);
    cout << bbb << endl;
    f1.close();
}
```



lo Wo

Строковые потоки

Строковые потоки позволяют считывать и записывать информацию из областей оперативной памяти так же, как из файла, с консоли или на дисплей. В стандартной библиотеке определено три класса строковых потоков:

istream — входные строковые потоки;

ostream — выходные строковые потоки;

stringstream — двунаправленные строковые потоки.

Эти классы определяются в заголовочном файле `<sstream>` и являются производными от классов **istream**, **ostream** и **iostream** соответственно, поэтому они наследуют перегруженные операции `<<` и `>>`, флаги форматирования, манипуляторы, методы, состояние потоков и т. д. Участки памяти, с которыми выполняются операции чтения и извлечения, по стандарту определяются как строки C++ (класс **string**). Строковые потоки создаются и связываются с этими участками памяти с помощью конструкторов.

Строковые потоки

Строковые потоки являются некоторым аналогом функций **sscanf** и **sprintf** библиотеки **C** и могут применяться для преобразования данных, когда они заносятся в некоторый участок памяти, а затем считываются в величины требуемых типов. Эти потоки могут применяться также для обмена информацией между модулями программы.

В строковых потоках описан метод **str**, возвращающий копию строки или устанавливающий ее значение:

```
string str() const;
```

```
void str(const string & s);
```

Проверять строковый поток на переполнение не требуется, поскольку размер строки изменяется динамически.

В приведенном ниже примере строковый поток используется для формирования сообщения, включающего текущее время и передаваемый в качестве параметра номер:

Строковые потоки

Пример:

```
#include <sstream>
```

```
#include <iostream>
```

```
#include <time.h>
```

```
using namespace std;
```

```
string me(int i) {
```

```
    stringstream os; // поток строкового вывода
```

```
    time_t t;
```

```
    time(&t); // записать текущее время в t
```

```
    os << " time: " << ctime(&t) << " number: " << i << endl;
```

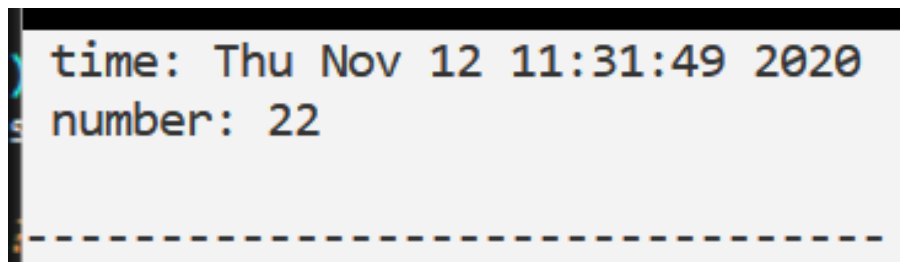
```
    return os.str(); //вернуть выведенную в поток информацию как строку
```

```
}
```

```
int main(){
```

```
    cout<<me(22); return 0;
```

```
}
```



```
time: Thu Nov 12 11:31:49 2020
number: 22
```