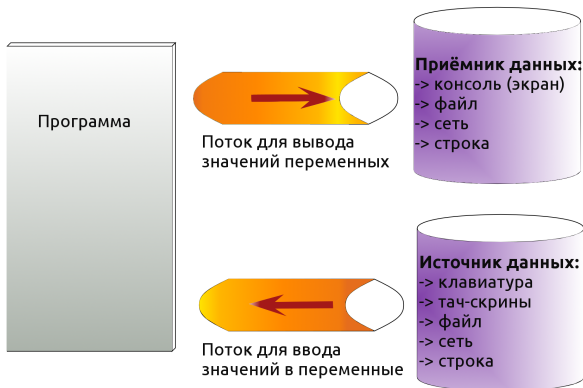


Лекция VI

30 ноября 2018

Ввод - вывод: как работает

Потоковый ввод/вывод



Данные в программе:

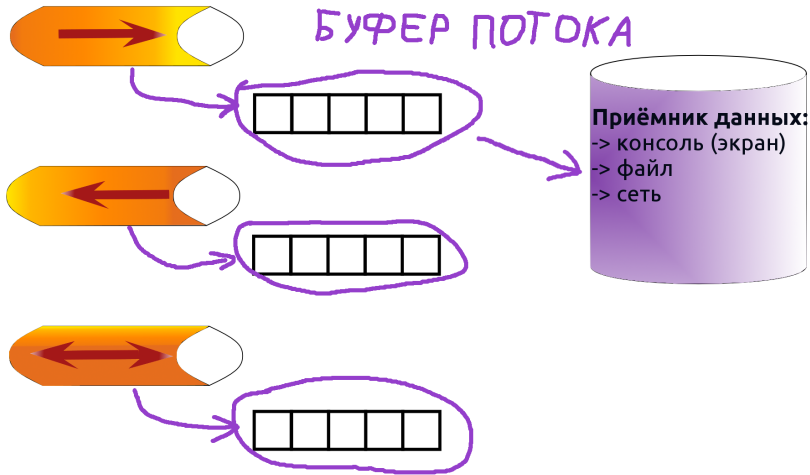
- * Текст: `char`, `string`
- * Числа: `int`, `double`, `size_t`

Данные снаружи:

- * Байты
 - > текст (ASCII, UTF8, ...)
 - > бинарные данные (raw bytes)

Концепция потока данных (stream)

- Логическая сущность, связывающаяся с одним приёмником или источником данных
- Берёт на себя обязанности по приёму/отправке байт извне
- Преобразует байты в значения требуемых типов при вводе; преобразует значения различных типов в байтовое представление при выводе
- Скрывает взаимодействие с реальным устройством ввода-вывода
- Имеет состояние: готов поток или нет выполнять операций ввода/вывода (корректное / ошибочное)
- Как правило, буферизован



В C++ стандартная библиотека предоставляет потоковый ввод/вывод для:

- 1 **текстового терминала** (консоль, командная строка) - осуществление операций вывода значений/ввода значений с экрана. Используются с помощью библиотеки **<iostream>**
- 2 **строки** - строки можно создавать или разбирать с помощью потоковых объектов. Подключаемая библиотека - **<sstream>**
- 3 **файлов** - запись/чтение файлов происходит с помощью библиотеки **<fstream>**

Ввод/вывод данных



Форматированный:

Для чтения: каждая группа символов (разделяемых пробелами) преобразуется в значение требуемого типа.

Для записи: значения конкретного типа преобразуется в текстовый вид самим потоком, и отправляется в файл.



Неформатированный:

читается/записывается строго определённое количество байт (указываемое в программе) и никаких форматных преобразований не происходит

При запуске каждой программы на C++ для работы с *текстовым терминалом*, ей предоставляется четыре потока (три для вывода, один - для ввода):

- **cout** - стандартный поток вывода, связанный с текстовым терминалом
- **cerr** - стандартный поток вывода для сообщений об ошибках. Перед записью в этот поток любого значения, буфер **cout** - сбрасывается
- **clog** - тоже поток для записи об ошибках, но при записи в него сброса буфера у **cout** не происходит
- **cin** - стандартный поток ввода, связанный с текстовым терминалом

Дополнительные характеристики потоков любых видов

- Есть индикатор текущего положения потока: количество прочитанных (ввод)/записанных (вывод) байт в текущий момент
- Такой индикатор позволяет в случае необходимости перемещаться по потоку (например, записывать значения в середину файла, а не в конец)
- Объекты потоков не являются копируемыми: то есть, нельзя применять *оператор присваивания* к переменным, представляющим собой поток ввода (или вывода)
- Все действия, что можно выполнять для чтения/записи различных значений - одинаковы для потоков всех видов (строковые, терминальные, файловые). Это означает, что операторы и методы для разбора информации - одинаковы для переменных разных типов

Далее идёт демонстрация работы с файловыми потоками ввода-вывода

Стандартная библиотека C++ определяет три основных класса для работы с файлами, определённых в библиотеке **<fstream>**:

- ❶ **ifstream** - для *Ввода (чтения)* информации из файла.
- ❷ **ofstream** - для *Вывода (записи)* информации в файл.
- ❸ **fstream** - для одновременных операций как *чтения*, так и *записи* из/в файл(а).

которые доступны после следующего include'a:

```
1 #include <fstream>
2
3 using namespace std;
```

Создание файлового потока для ввода: два конструктора

```
(1) ifstream stream_var;  
(2) ifstream stream_var{file_name,  
                        open_mode = ios_base::in};
```

- ❶ (1) - конструктор по умолчанию. Создаёт объект потока для ввода из файла, но не связывает его ни с каким реальным файлом
- ❷ (2) - конструктор с двумя аргументами: **file_name** - имя файла, с которым связывается поток; или C-строка или объект класса **string**. **open_mode** - специальные флаги для выбора режима работы с файлом, по умолчанию - чтение

Создание файлового потока для ввода

```
1 // Ни с каким файлом поток не связан ещё
2 ifstream config_file1;
3
4 // Следующий поток будет читать данные из ←
  config.dat
5 ifstream config_file2{"config.dat"};
6
7 // Для потока ввода можно указывать полный ←
  путь
8 string file_name = "C:\\test\\my_config.txt";
9 ifstream config_file3{file_name};
```

Режим открытия файла (**open_mode**) определяется набором флагов

Флаг	Для чего нужен
ios_base::ate	При открытии текущая позиция потока устанавливается в конец файла
ios_base::app	Операции вывода начинаются с конца файла, то есть происходит дозапись
ios_base::binary	Операции ввода/вывода происходят в двоичном режиме
ios_base::out	Открыть поток на запись
ios_base::in	Открыть поток на чтение
ios_base::trunc	При открытии файла удалить всё его содержимое

Флаги могут комбинироваться с помощью оператора побитового «или» - |

Пример изменения режима работы с файлом

```
1 // Открытие в двоичном режиме на чтение
2 // и перемещение в конец файла
3 auto my_mode = ios_base::in
4               | ios_base::binary
5               | ios_base::ate;
6 ifstream in_file{"my_file.txt", my_mode};
```

Отложенная связь потока с файлом: метод **open**

```
stream_var.open(file_name,  
                open_mode = ios_base::in);
```

- **file_name** - имя файла, с которым связывается поток: или C-строка или объект класса **string**
- **open_mode** - специальные флаги для выбора режима работы с файлом, по умолчанию - режим чтения данных
- Если функция **open** вызвана для потока, который уже открыт - поток переводится в ошибочное состояние

```
1 // Ни с каким файлом поток не связан ещё  
2 ifstream config_file1;  
3  
4 // А теперь — связан с first.log  
5 config_file1.open("first.log");
```


Проверка готовности файла, после создания потока

```
(1) bool stream_var.is_open();  
(2) if ( stream_var ) { ... };
```

- ❶ Метод **is_open** - возвращает значение **true**, если файл существует и с ним установлена связь; **false** - в противном случае

```
1 ifstream in_file1{"data_file.txt"};  
2 if ( in_file1.is_open() ) {  
3     // Ошибок нет, совершаем операции с файлом  
4 }
```

- ❷ Непосредственное использование переменной потока в условных выражениях (более общая форма)

```
1 ifstream in_file2{"another_file.txt"};  
2 if ( in_file2 ) {  
3     // Ошибок нет, совершаем операции с файлом  
4 }
```

Форматированный ввод значений нужного типа: оператор ввода

```
ifstream& stream_var.operator>>(Type& value);
```

- Преобразует группу символов в значение **value**. **Type** - известный к моменту вызова оператора тип данных: **int**, **double**, **size_t**, **string**, ...
- Оператор ввода возвращает ссылку на тот поток, из которого происходит чтение

Файловый ввод

Пример оператор ввода: пусть дан файл info.txt

45 678.905

1.2387E-3

Строка - просто строка

```
1 ifstream in_file{"info.txt"};
2 if ( in_file.is_open() ) {
3     int num1; double real1, real2;
4     in_file >> num1 >> real1;
5     in_file >> real2;
6
7     string word;
8     in_file >> word;
9
10    cout << "Целое: " << num1 << "; вещественные: "
11         << real1 << ", " << real2 << "\n";
12    cout << "Первое слово: " << word << "\n";
13 }
```

Методы для проверки на отсутствие ошибок при операциях ввода/вывода

- 1 Проверка достижения конца файла
`bool stream_var.eof();`
- 2 Проверка успешности операций чтения/записи (не был ли файл удалён, не пропал ли к нему доступ)
`bool stream_var.bad();`
- 3 Проверка успешности ввода данных (отсутствие логических ошибок)
`bool stream_var.fail();`
- 4 Полное отсутствие любых ошибок - участие переменной потока в условном выражении
`if (stream_var) { /*всё хорошо*/ }`

Если один из первых трёх методов возвращает **true**, то операции ввода/вывода просто не выполняются, сколько бы мы не обращались к объекту потока.

Пример оператор ввода: проверка успешности ввода для всех переменных

```
1 ifstream in_file{"info.txt"};
2 if ( in_file.is_open() ) {
3     int num1; double real1, real2;
4     in_file >> num1 >> real1;
5     in_file >> real2;
6
7     string word;
8     in_file >> word;
9
10    if ( in_file ) {
11        cout << "Целое: " << num1
12            << "; вещественные: " << real1
13            << ", " << real2 << "\n";
14        cout << "Первое слово: " << word << "\n";
15    }
16 }
```

Пример ввода всех чисел из файла. Пусть дан nums.txt:

```
45.657 6.88 10.56 5.456 8.9905 6.7  
7.8 14.5 5.616 8.8888 10.14 7.899
```

```
1 ifstream input_file{"nums.txt"};  
2 if ( input_file.is_open() ) {  
3     double num;  
4     // Прочитать все числа из файла  
5     // и напечатать их значения в консоли  
6     while ( input_file >> num; ) {  
7         cout << '\n' << num;  
8     }  
9  
10    if ( input_file.bad() ) {  
11        cerr << "Ошибка операций ввода/вывода";  
12    } else if ( input_file.fail() ) {  
13        cerr << "В файле есть нечисловые данные";  
14    }  
15 }
```

Посимвольное чтение файла - метод **get**

```
(1) int stream_var.get()
```

```
(2) ifstream& stream_var.get(char& symbol)
```

- 1 Читаем один символ из потока и возвращаем его код (несмотря на то, что возвращается значение типа **int**, фактически каждым символом считается один байт (**char**), то есть код многобайтового символа данный метод не вернёт
- 2 Читаем один символ из потока и помещаем его в переменную **symbol**

Чтение символов в C-строку

```
istream& stream_var.getline(char *str,  
                             size_t count, char delim = '\\n')
```

- Читаем как максимум **count - 1** символов и записываем их в переменную **str**. В **str** также добавляется символ окончания строки '\\0'. **delim** - символ разделитель, по умолчанию - символ переноса строки
- **Важно:** символ-разделитель извлекается из потока и не участвует в последующих операциях чтения данных.

Файлы. Чтение данных без форматирования

В файле:

Very important data

что-то делают

в нашем файле

```
1 ifstream input_file{"data_file.txt"};
2 if ( input_file.is_open() ) {
3     char str[8];
4     input_file.getline(str, 8);
5     // Печатаем "Very im"
6     cout << str << '\n';
7
8     char word[5];
9     input_file.getline(word, 5);
10    // Печатаем "port"
11    cout << word << '\n';
12 }
```

Файловый ввод

Посимвольное чтение файла

Что

говорить

говорить...

когда нечего

```
1 char sym;
2 ifstream in_file{"my_text.txt"};
3 if ( in_file.is_open() ) {
4     while ( in_file >> sym ) {
5         cout << sym;
6     }
7 }
8 // Разница: пропускаем пробелы и переносы или нет
9 ifstream in_again{"my_text.txt"};
10 if ( in_again.is_open() ) {
11     while ( in_file.get(sym) ) {
12         cout << sym;
13     }
14 }
```

Чтение символов в объект класса **string**

```
ifstream& getline(stream_var, string str,  
                  char delim = '\\n')
```

Читаем из файла и помещаем все символы в строку **str** до тех пор, пока не встретится разделитель

Very important data

что-то делают

в нашем файле

```
1 ifstream in_file{"data_file.txt"};  
2 if ( in_file.is_open() ) {  
3     string str;  
4  
5     while ( getline(in_file, str) ) {  
6         cout << "[" << str << "]" << "\\n";  
7     }  
8 }
```

Пример: есть файл text.txt:

Файл для посимвольного

чтения информации; #1 2# #3 #4 5

#...#

```
1 ifstream in_text{"text.txt"};
2 int symb, sharp_count = 0;
3
4 while ( in_text.get(symb) ) {
5     cout << symb;
6
7     if (symb == '#' ) { ++sharp_count; }
8 }
9
10 cout << "Найдено " << sharp_count << "решёток\n";
```

```
ifstream& stream_var.ignore(size_t count = 1,  
                             char delim = '\\n')
```

- Метод **ignore** пропускает заданное количество символов (байт) из файла и оставляет их необработанными (то есть не происходит сохранение или преобразование извлечённых символов). Пропуск прекращается или по достижении считывания **count** символов, или при встрече символа-разделителя **delim**.
- Оба параметра метода - **count** и **delim** имеют значения по умолчанию: **count** равен единице, а разделитель **delim** специальному символу, означающему конец файла
- Если пропуск символов прекращается при нахождении разделителя, то он тоже извлекается из потока и не участвует в дальнейших операциях чтения данных

Когда может быть полезен метод **ignore**?

Во многих программах для ввода начальных параметров более уместно использовать *конфигурационные файлы*, вместо ввода значений через консоль. Особенно это относится к вычислительным задачам: граничные условия при расчёте задач по вычислению различных интегралов или систем уравнений; количество частиц и параметры вроде температуры для задач термодинамики; размеры матриц в каких-нибудь вычислениях.

Конфигурационные файлы предпочтительней хотя бы тем, что при изменениях параметров быстрее и надёжнее поменять их в текстовом файле, чем каждый раз сосредотачиваться на консольном вводе.

Пример конфигурационного файла некой вычислительной задачи:

Максимальное число итераций: 260000

Количество строк матриц: 15

Количество столбцов матриц: 25

Количество слоёв: 5

Сила трения между слоями: -7.8

Что можно выделить из описания файла на предыдущем слайде?

- Есть повторяющаяся структура: описание параметра - двоеточие - значение
- Программе нужны значения комментариев-описания
нужны для человека
- Комментарии нужны для человека

Для написания универсального разбора и пригодится метод `ignore`

```
1 const size_t PASS_COUNT = 500;
2 size_t max_iter, cols, rows, layers_count;
3 double fric_force;
4
5 ifstream config_file{"my_config_file.dat"};
6 if ( config_file.is_open() ) {
7     // пропускаем символы до двоеточия
8     config_file.ignore(PASS_COUNT, ':');
9     // безопасно считываем первое значение
10    config_file >> max_iter;
11
12    config_file.ignore(PASS_COUNT, ':');
13    config_file >> cols;
14    // ... Остальные переменные – аналогично
15 }
```

Код со слайда **23** разбирает файл со слайда **21** со следующими особенностями:

- Разумно предположить, что более 500 символов в качестве описания параметра человеку будет просто лень набирать
- Перед вводом *каждого* числового значения ищется символ двоеточия
- После нахождения - считываем числовое значение в нужную переменную

```
void stream_var.clear()
```

- Метод **clear** позволяет вернуть поток в состояние, пригодное для новых попыток считывания данных.

Выше приводились три метода, которые проверяют, не случилось ли каких либо ошибок при операциях между объектом потока и файлом - **eof()**, **bad()** **fail()**. Если хотя бы один из этих методов возвращает **true**, то дальнейшие операции получения данных невозможны. Метод **clear** возвращает поток в такое состояние, что все методы проверки состояния начинают возвращать значение **false**. Это позволяет попробовать считать какие-нибудь данные снова.

Пусть дан файл, нужно считать все числа. Каждое число отделено пробелом, но могут попадаться и нечисловые символы

45.657 6.88 6.7 7.856 14.5 asd 5.616fs 8.88 sdsf

```
1 #include <limits>
2
3 using ios_lims = numerical_limits<streamsize>;
4 const size_t MAX_IGNORE = ios_lims::max();
5
6 ifstream input_file{"data_file.txt"};
7 if ( input_file.is_open() ) {
8     double num;
9     while ( input_file >> num ) {
10         if ( input_file.fail() ) {
11             input_file.clear();
12             input_file.ignore(MAX_IGNORE, ' ');
13         }
14     }
15 }
```

- Запись данных в файл осуществляется с помощью класса **ofstream** (поточковый класс, осуществляющий только операции вывода). Объекты этого класса связываются с файлом либо через конструктор, либо через функцию **open**, аналогично **ifstream**.
- При связи потока с несуществующим файлом, последний создаётся. Но стандартная библиотека ввода/вывода C++ не умеет создавать директории, если они не существуют
- Также объекты **ofstream** имеют методы **is_open()**, **bad()** и **fail()** - для проверки состояния потока.
- По умолчанию файл всегда создаётся заново, то есть если он существовал, то содержимое будет стёрто. Для предотвращения нужно пользоваться флагом **ios_base::app**

Аналогично операциям с консольным выводом через **cout**, запись данных происходит с помощью оператора:

`ofstream& stream_var.operator<<(Type& value)`

```
1 ofstream out_file{"D:\\test\\sq_of_nums.txt"};
2
3 if ( out_file.is_open() ) {
4     // Вывод квадратов чисел от 1 до 100
5     for (size_t i = 1; i <= 100; ++i) {
6         out_file << i << " * " << i
7             << " = " << i * i << '\n';
8     }
9 }
```

1 * 1 = 1

2 * 2 = 4

3 * 3 = 9

...

Пример записи в файл

```
1 #include <iomanip>
2
3 ofstream out("some_results.txt");
4 if ( out.is_open() ) {
5     int i_num = 858;
6     double r_num = 14.8326372364277;
7     string str = "Как всё просто";
8
9     out << boolalpha << showpos;
10    out << "Целое число: " << setfill('#')
11        << setw(8) << i_num << setfill(' ');
12
13    out << "\n Десятичное: " << r_num << "\n{";
14    out << str << "}}\n" << true << endl;
15 }
```

В файле "some_results.txt" окажется текст:

```
Целое число: +#####858  
Десятичное: +14.8326  
{{Как всё просто}}  
true
```

Что за **endl**

Использование **endl** - специального значения, означающего переноса строки - приводит к сбросу буфера потока: то есть к реальной записи накопившихся символов в файл.

Справка по форматированному выводу значений -
<https://github.com/posgen/OmsuMaterials/wiki/Format-output>

Файловый вывод

Запись символов как значений типа **char** в файл.

```
ostream& stream_var.put(char symbol)
```

Пример: посимвольный вывод всех строчных букв английского алфавита в файл

```
1 ofstream out_file{"alphabet.txt"};
2
3 if ( out_file.is_open() ) {
4     for (char sym = 'a'; sym <= 'z'; ++sym) {
5         // Метод put Возвращает ссылку
6         // на тот объект, для которого
7         // он был вызван, что позволяет
8         // строить цепочки, подобные следующей
9         out_file.put(sym).put( '\n' );
10    }
11 }
```

Справка по файловому вводу-выводу доступна в интернете, а также здесь:

<https://github.com/posgen/OmsuMaterials/wiki/File-input-output>

Методы: **flush**, **tellp**, **seekp**, **tellg**, **seekg**.

Работа со строковыми потоками

Строковые потоки ввода/вывода

Для строковых потоков также определено три основных класса для работы с ними:

- 1 **istream** - для *ввода (чтения)* значений из какой-нибудь строки.
- 2 **ostream** - для *вывода (записи)* значений в строку (фактически - форматированное или неформатированное создание строки).
- 3 **stringstream** - для одновременных операций как *чтения*, так и *записи* из/в строку.

которые доступны после следующего include'a:

```
1 #include <sstream>
2
3 using namespace std;
```

Сами **операторы и методы** для работы с объектами указанных классов аналогичны приведённым выше для файловых потоков

Создание потока для ввода:

```
1 istringstream input;
```

Но созданный таким образом поток не содержит никаких данных, поэтому операции ввода значений из него приведут к ошибке. Обычно, потоку ввода надо предоставить какую-нибудь строку для разбора. Это можно сделать при использовании метода **str**:

```
2 string my_str = "Строка для разбора";  
3 input.str(my_str);
```

Альтернативный способ - передать потоку строку в момент создания его объекта (ака переменная):

```
1 istringstream input_nums{"1234.56 -0.567 4.555 ←  
    0.334"};
```

Строковые потоки ввода

Как только поток связан с какой-нибудь строкой - можно начинать операции ввода. Как пример: разбор входящей строки на слова с помощью потока:

```
1 string str;
2 cout << "Введите строку: ";
3 getline(cin, str);
4
5 // Создали строковый поток Ввода, который
6 // будет разбирать символы из str
7 istream input{str};
8
9 cout << "Введённые слова:\n";
10 string word;
11 while (input >> word) {
12     cout << '{' << word << '}' << endl;
13 }
```

Пример: разбор входящей строки на действительные числа с помощью потока:

```
1 istringstream input_nums{"1234.56 -0.567 4.555 ↵  
    0.334"};  
2  
3 double numb;  
4 while (input_nums >> numb) {  
5     cout << "Получено число: " << numb < endl;  
6 }
```

Общий подход к разбору строковых или файловых потоков заключается в следующем: выполнить **ожидаемые** операции ввода, а затем - проверить, были ли все из них успешны. Для примера, рассмотрим следующую задачу: пусть есть файл, в котором в каждой строке записаны по 10 чисел. Пусть это будут значения некоторой физической величины в определённые моменты времени. Но некоторые строки могут содержать меньше значений, чем 10. Задача состоит в том, чтобы посчитать средние значения в каждой точке **только** по наборам данных из 10 чисел. Файл, для примера, может выглядеть так:

3	-4	2	6	7	-55	7	8	11	
6	5	1	9	8					
2	3	3	11	4	-23	3			
-1	-1	5	12	5	-5	-7	6	14	

Строковые потоки ввода

Целые числа использованы лишь для наглядности. Вторая и третья строка - не подходят для усреднения.

```
1  const size_t TENTH = 10;
2
3  ifstream in_data{"nums.txt"};
4  if (in_data) { // Если файл найден
5      string data_line;
6      double avereges[TENTH] = { 0.0 };
7      stringstream input;
8      size_t lines_count = 0;
9      // Начинаем читать из файла строку за строкой
10     while (getline(in_data, data_line) {
11         double arr[TENTH];
12         // Каждую прочитанную строку — передаём
13         // в строковый поток ...
14         input.str(data_line);
15         //... и пытаемся получить именно 10 чисел
16         for (size_t i = 0; i < TENTH; i++) {
17             input >> arr[i];
18         }
```

Продолжаем...

```
18 // Проверяем, что 10 чисел введены без проблем
19 if ( input ) {
20     // всё ок — добавляем к средним
21     for ( size_t i = 0; i < TENTH; i++) {
22         averages[i] += arr[i];
23     }
24     lines_count++;
25 } else {
26     input.clear();
27 }
28 }
29 if (lines_count > 0) {
30     // Усредняем и показываем результат
31     for ( size_t i = 0; i < TENTH; i++) {
32         averages[i] /= lines_count;
33         cout << "Среднее в " << (i + 1)
34             << "точке: " << averages[i] << endl;
35     }
36 }
37 }
```

Строковые потоки вывода

Строковые потоки для вывода - помогают в некоторых ситуациях создавать строки с нестандартно отформатированными значениями. Создание такого потока делается так:

```
1 ostringstream output;
```

Созданный таким образом поток по умолчанию содержит внутри себя пустую строку, к которой будет добавляться другая текстовая информация. Также можно сразу задать содержимое строки, которую нужно дополнить чем-нибудь. Это можно сделать при использовании метода **str**:

```
2 string my_str = "Самый лучший заголовок\n";
```

```
3 output.str(my_str);
```

Альтернативный способ - передать потоку строку в момент создания его объекта (ака переменная):

```
1 ostringstream output_nums{"1234.56 -0.567"};
```

Строковые потоки вывода

Работа со строковыми потоками вывода - на примере: нужно создать строку, в которой действительные числа будут в *научной нотации*, с **пятью** знаками после запятой и выведенные в столбик. Плюс шапку перед колонкой чисел добавим.

```
1 //Второй аргумент нужен для дозаписи в поток
2 ostream output{"Числа по формату:\n",
3               ios_base::ate};
4 double nums[] = {1.4566723, -567.2334575,
5                  556.623322, 8.90335435,
6                  0.47437, 1000983.1283713831};
7 output << scientific << setprecision(6);
8 for (const double& elem : nums) {
9     output << setw(12) << elem << "\n";
10 }
11 string final_str = output.str();
12 cout << "Полученная строка:\n"
13      << final_str << endl;
```