# IV

# Текст и C++

# Кодировка в ОС

**Кодировка** - специальная таблица, связывающая каждый символ с соответствующим ему целым числом

1. **ASCII**[1] – 8-битная кодировка:
   - 256 символов, представленных неотрицательными целыми кодами: от $0$ до $255$ (ссылка). Изначально поддерживала только 128 символов
   - каждый символ занимает один байт
   - Включает в себя все цифры, буквы английского алфавита в нижнем/верхнем регистрах и некоторые другие символы (тильда, процент, '@', решётка и т.п.)
   - Коды букв (отдельно группы в верхнем и нижнем регистрах) и цифр - идут последовательно
   - В C++ в диапазон типа **char** гарантированно входят первые 128 символов

[1] American Standard Code for Information Interchange, стандартная американская кодировка для обмена информацией

# Кодировка в ОС

2. Региональные кодировки: расширенные версии ASCII: **CP866**, **CP1251**, **KOI8-R**, **Latin-1**.
   - 128 положительных кодов совпадали с кодами ASCII
   - Добавляли локальный алфавит: *й, ы, ъ, ä, ö, Õ*
   - В C++ позволяли обходиться типом **char**
   - Не обладали переносимостью

# Кодировка в ОС

2. Региональные кодировки: расширенные версии ASCII: **CP866**, **CP1251**, **KOI8-R**, **Latin-1**.
   - 128 положительных кодов совпадали с кодами ASCII
   - Добавляли локальный алфавит: *й, ы, ъ, ä, ö, Õ*
   - В C++ позволяли обходиться типом **char**
   - Не обладали переносимостью
3. Унифицированный набор символов (юникод), многобайтовые кодировки: **UTF-16**, **UTF-32** (unicode transformation format)
   - руны (runes, code points) – текстовые символы
   - базовая единица, «code unit» – 16- ил 32-бита
   - **wchar_t**, **char16_t**, **char32_t**

# Кодировка в ОС

2. Региональные кодировки: расширенные версии ASCII: **CP866**, **CP1251**, **KOI8-R**, **Latin-1**.
   - 128 положительных кодов совпадали с кодами ASCII
   - Добавляли локальный алфавит: *й, ы, ъ, ä, ö, Õ*
   - В C++ позволяли обходиться типом **char**
   - Не обладали переносимостью
3. Унифицированный набор символов (юникод), многобайтовые кодировки: **UTF-16**, **UTF-32** (unicode transformation format)
   - руны (runes, code points) – текстовые символы
   - базовая единица, «code unit» – 16- ил 32-бита
   - **wchar_t**, **char16_t**, **char32_t**
4. 8-битный юникод – **UTF-8**
   - базовая единица – 8-бит
   - руна представлена от 1 до 4 единицами
   - абсолютная победа в эпоху цифрового мира

# C++ и текст

- это не лучший инструмент;
- какой диапазон могут хранить базовые строки (**char***, **char**[], они же - C-строки) – определяется ОС;
- в ОС Windows юникод представлен кодировкой **UTF-16**, в других – **UTF-8**;
- стандратная библиотека не предоставляет никаких средств для работы с рунами, только с базовыми единицами кодировки;

# C++ и текст

- это не лучший инструмент;
- какой диапазон могут хранить базовые строки (**char***, **char**[], они же - C-строки) – определяется ОС;
- в ОС Windows юникод представлен кодировкой **UTF-16**, в других – **UTF-8**;
- стандратная библиотека не предоставляет никаких средств для работы с рунами, только с базовыми единицами кодировки;

```
1 const char* str =
2   "this: \xE7\x82\xA4, \xE2\x9B\x94, \xF0\x9F\xA6↩
      \x81";
3 std::cout << str << std::endl;
4
5 for (char unit : str) {
6   std::cout << unit << std::endl;
7 }
```

- стоит выбирать строки на основе **char**.

# C++: работа с текстом

Для работы с текстом C++ в своей стандартной библиотеке предоставляет тип – **string**. Это специальный класс, который:

- подключается через заголовочный файл:

```
1 #include <string>
```

- представляет собой динамическую строку из символов типа **char**, скрывая детали работы с памятью;
- даёт возможность копирования содержимого разных переменных через оператор присваивания;
- даёт возможность использовать стандартные операторы сравнения, например: $>$, $<$, $==$, $!=$;
- использует оператор $+$ для объединения строк.

**Как обычно, для стандартной библиотеки C++**

тип определён внутри пространства имён **std** и его полное именование **std::string** (без использования **using**).

# C++: конструкторы string

Способы создать строковые объекты:

- пустые строки – конструктор по умолчанию

```
1 std::string s1, s2{};
```

- из базовых строк – +2 конструктора

```
1 const char* base_str = "try to be better";
2
3 std::string s3{"new world"}, s4{base_str, 9},
4             s5{base_str + 10, 6};
```

- фиксированная длина + символ-заполнитель – +1

```
1 std::string s6(10, '@');
```

- из другого строкового объекта – +3

```
1 std::string s7{s3}, s8{s4, 4}, s9{s4, 4, 2};
```

- альтеранативный синтаксис для **s3** и **s7**

```
1 std::string s3 = "new world", s7 = s3;
```

# C++: конструкторы string

Способы создать строковые объекты:

- пустые строки – конструктор по умолчанию

```
1 std::string s1, s2{};
```

- из базовых строк – +2 конструктора

```
1 const char* base_str = "try to be better";
2
3 std::string s3{"new world"}, s4{base_str, 9},
4             s5{base_str + 10, 6};
```

- фиксированная длина + символ-заполнитель – +1

```
1 std::string s6(10, '@');
```

- из другого строкового объекта – +3

```
1 std::string s7{s3}, s8{s4, 4}, s9{s4, 4, 2};
```

- альтеранативный синтаксис для **s3** и **s7**

```
1 std::string s3 = "new world", s7 = s3;
```

# C++: конструкторы string

Способы создать строковые объекты:

- пустые строки – конструктор по умолчанию

```
1 std::string s1, s2{};
```

- из базовых строк – +2 конструктора

```
1 const char* base_str = "try to be better";
2
3 std::string s3{"new world"}, s4{base_str, 9},
4             s5{base_str + 10, 6};
```

- фиксированная длина + символ-заполнитель – +1

```
1 std::string s6(10, '@');
```

- из другого строкового объекта – +3

```
1 std::string s7{s3}, s8{s4, 4}, s9{s4, 4, 2};
```

- альтеранативный синтаксис для **s3** и **s7**

```
1 std::string s3 = "new world", s7 = s3;
```

# C++: конструкторы string

Способы создать строковые объекты:

- пустые строки – конструктор по умолчанию

```
1 std::string s1, s2{};
```

- из базовых строк – +2 конструктора

```
1 const char* base_str = "try to be better";
2
3 std::string s3{"new world"}, s4{base_str, 9},
4             s5{base_str + 10, 6};
```

- фиксированная длина + символ-заполнитель – +1

```
1 std::string s6(10, '@');
```

- из другого строкового объекта – +3

```
1 std::string s7{s3}, s8{s4, 4}, s9{s4, 4, 2};
```

- альтеранативный синтаксис для **s3** и **s7**

```
1 std::string s3 = "new world", s7 = s3;
```

# C++: конструкторы string

Способы создать строковые объекты:

- пустые строки – конструктор по умолчанию

```
1 std::string s1, s2{};
```

- из базовых строк – +2 конструктора

```
1 const char* base_str = "try to be better";
2
3 std::string s3{"new world"}, s4{base_str, 9},
4             s5{base_str + 10, 6};
```

- фиксированная длина + символ-заполнитель – +1

```
1 std::string s6(10, '@');
```

- из другого строкового объекта – +3

```
1 std::string s7{s3}, s8{s4, 4}, s9{s4, 4, 2};
```

- альтеранативный синтаксис для **s3** и **s7**

```
1 std::string s3 = "new world", s7 = s3;
```

Инициализация списком, три шага к неоднозначности:

1. начало неплохое

```
1 std::string s10 = {'f', 'i', 'n', 'e'};
```

# C++: конструкторы string

Инициализация списком, три шага к неоднозначности:

1. начало неплохое

```
1 std::string s10 = {'f', 'i', 'n', 'e'};
```

2. явный вызов конструктора

```
1 std::string s10{{'f', 'i', 'n', 'e'}};
```

# C++: конструкторы string

Инициализация списком, три шага к неоднозначности:

1. начало неплохое

```
1 std::string s10 = {'f', 'i', 'n', 'e'};
```

2. явный вызов конструктора

```
1 std::string s10{{'f', 'i', 'n', 'e'}};
```

3. упрощение от авторов C++

```
1 std::string s10{'f', 'i', 'n', 'e'};
```

# C++: конструкторы string

Инициализация списком, три шага к неоднозначности:

1. начало неплохое

```
1 std::string s10 = {'f', 'i', 'n', 'e'};
```

2. явный вызов констурктора

```
1 std::string s10{{'f', 'i', 'n', 'e'}};
```

3. упрощение от авторов C++

```
1 std::string s10{'f', 'i', 'n', 'e'};
```

Какой конструктор вызывается далее?

```
1 std::string s11{50, 'i'};
```

# C++: конструкторы string

Инициализация списком, три шага к неоднозначности:

1. начало неплохое

```
1 std::string s10 = {'f', 'i', 'n', 'e'};
```

2. явный вызов конструктора

```
1 std::string s10{{'f', 'i', 'n', 'e'}};
```

3. упрощение от авторов C++

```
1 std::string s10{'f', 'i', 'n', 'e'};
```

Какой конструктор вызывается далее?

```
1 std::string s11{50, 'i'};
```

**Правило вызова корректного конструктора**

Если для типа определён конструктор, определяющий инициализацию списком, то он будет выбран всегда при использовании явных фигурных скобок, если перечисляемые значения приводятся к ожидаемому типу.

Вывод содержимого строкового объекта в текстовый терминал

```
1  std::string text =
2      "Everything\n"
3      "that has\n"
4      "begining\n"
5      "has end";
6
7  text += std::string{"."};
8
9  std::cout << "I remember that:\n"
10           << text << std::endl;
```

Плюс демонстрация *контактенации* строк (строка **7** в примере).

Сравнение строк: для полного сравнения строк используются привычные операторы сравнения: $>$, $>=$, $<$, $<=$, $==$, $!=$

```
1  std::string s1 = "France",
2              s2 = "Russia";
3
4  if ( s1 < s2 ) {
5    std::cout << "No doubts!\n";
6  }
7
8  bool is_equals = s1 == s2;
9  std::cout << std::boolalpha
10           << "Is s1 equals to s2?\n"
11           << "Answer is " << is_equals
12           << std::endl;
```

# C++: работа с текстом

Сравнение строк: как операторы сравнения $>$, $>=$, $<$, $<=$, $==$, $!=$ работают.

Строка s1 считается **меньше**, чем s2 в двух случаях (s1 $<$ s2)

1. первый несовпадающий символ в s1 имеет **целочисленный код** меньше, чем таковой в s2
2. все символы из s1 совпадают с начальными символами в s2, но вторая строка имеет **большую длину**

Строка s1 считается **больше**, чем s2 в двух случаях (s1 $>$ s2)

1. первый несовпадающий символ в s1 имеет **целочисленный код** больше, чем таковой в s2
2. начальные символы из s1 совпадают со всеми символами в s2, но первая строка имеет **большую длину**

Иначе - строки считаются **равными**. Терминологически — *лексикографическое* сравнение строк.

Таблица 1: Методы касающиеся размера строкового объекта и внутреннего буфера

| | |
|---|---|
| size_t size()<br>size_t length() | Число символов строки (значений **char**) |
| bool empty() | Проверить, пустой ли строковый объект |
| size_t max_size() | Узнать максимальный размер строки, которую можно хранить в объекте |
| void reserve(size_t how) | Выделить место под **how** символов |
| void shrink_to_fit() | Привести размер внутреннего буфера к актуальному количеству символов. Минимальная размерность буфера – 15 символов |

# C++: работа с текстом

```
1 string s1 = "whatever somewhere", s2;
2 if ( s2.empty() ) {
3   s2 = s1 + " somehow";
4 }
5
6 cout << "s1 len: " << s1.length() << '\n';
7 cout << "s2 len: " << s2.size() << '\n';
```

Таблица 2: Методы доступа к символам строки

| | |
|---|---|
| char& operator[size_t index] | Получение символа строки по индексу |
| char& at(size_t index) | Получение символа строки по индексу с проверкой границ. Если индекс вне диапазона – огибка времени выполнения |
| char& front() | Вернуть первый символ строки. Неопределённое поведение, если строка пуста. |
| char& back() | Вернуть последний символ строки. Неопределённое поведение, если строка пуста. |
| const char* c_str() | Вернуть неизменяемую C-строку. |

```
1  string s1 = "You'd like a holiday";
2
3  cout << "1st char: " << s1.front() << '\n';
4  cout << "Last  char: " << s1.back() << '\n';
5
6  cout << "char at 8: " << s1[7] << '\n';
7  cout << "char at 12" << s1.at(11) << '\n';
8  cout << s1.operator[](1) << endl;
9
10 // for-range example
11 for (const char& symb : s1) {
12   cout << symb << ",";
13 }
14 cout << endl;
```

Таблица 3: Методы поиска в строковых объектах

| | |
|---|---|
| size_t find(...) | Найти первое вхождение подстроки в исходном объекте |
| size_t rfind(...) | Найти последнее вхождение подстроки в исходном объекте |
| size_t find_first_of(...) | Найти первое вхождение любого символа из группы в исходном объекте |
| size_t find_first_not_of(...) | Найти первое вхождение любого символа **не** из группы в исходном объекте |

| | |
|---|---|
| size_t rfind_first_of(...) | Найти последнее вхождение любого символа из группы в исходном объекте |
| size_t rfind_first_not_of(...) | Найти последнее вхождение любого символа **не** из группы в исходном объекте |

| | |
|---|---|
| size_t rfind_first_of(...) | Найти последнее вхождение любого символа из группы в исходном объекте |
| size_t rfind_first_not_of(...) | Найти последнее вхождение любого символа **не** из группы в исходном объекте |

Тип **string** при поиске и других операциях со сторокой (вставка, замена) всегда оперирует позициями символов в строке (а не указателями в сравнении с языком C). В нём определено *статическое поле* **string::npos**, которое в зависимости от контекста может быть:

- индикатором того, что поиск не нашёл нужного фрагмента;
- индикатором конца строки.

# C++: поиск в тексте

Поиск в строке – на примере **find**

```
size_t str.find(frag, size_t pos = 0)
size_t str.find(base_str, size_t pos,
                size_t count)
```

- **frag** - текстовый фрагмент, который ищется в строке **str**
- возвращает позицию первого символа из **frag** в строке **str**, если **frag** присутствует в **str**;
- **frag** может быть переменной типа **string**, базовой строкой, и переменной типа **char**;
- вторая форма определена только для C-строк, третий параметр определяет длину фрагмента из base_str для поиска;
- поиск начинается с позиции, определяемой вторым аргументом;
- если фрагмента не было найдено, возвращается значение **string::npos**

# C++: поиск в тексте

Пример: поиск всех вхождений фрагмента в строку

```
1  string text = "Да, были люди в наше время,\n"
2                "Не то, что нынешнее племя:\n"
3                "Богатыри — не вы!\n"
4                "Плохая им досталась доля:\n"
5                "Немногие вернулись с поля...\n"
6                "Не будь на то господня воля,\n"
7                "Не отдали б Москвы!\n";
8  cout << "Ищем все запятые в тексте\n" << text;
9  cout << "\n--------------------------------\n";
10 const string to_found = "Не";
11
12 size_t found_pos = text.find(to_found);
13 while ( found_pos != string::npos ) {
14   size_t place = found_pos + to_found.length();
15   cout << to_found << "найдена на " << place
16        << " месте\n");
17   found_pos = text.find(to_found, place);
18 }
```

# C++: остальные операции с текстом

Частичное сравнение строк - метод **compare**

```
(1) int str.compare(other_str)
(2) int str.compare(size_t pos, size_t len,
                    other_str)
(3) int str.compare(size_t pos, size_t len,
      other_str, size_t o_pos, size_t o_len)
```

, где **str** - переменная типа **string**, которую сравниваем. А **other_str** - с которой сравниваем.

1. полное сравнение строк **str** и **other_str**
2. сравнение фрагмента внутри **str**, начиная с символа на позиции **pos** и длиной **len**, со строкой **other_str**
3. сравнение фрагментов из **str** и **other_str**. Позиция и длина для второго задаются аргументами **o_pos** и **o_len**

Метод **compare** возвращает **нуль**, если строки илиих фрагменты равны; **число больше нуля**, если str $>$ other_str; **число меньше нуля**, если str $<$ other_str (сравнение - лексикографическое).

# C++: остальные операции с текстом

```
1  using namespace std::string_literals;
2  string s1 = "два отличия найдите",
3         s2 = "найдите кота";
4  const size_t nlen = "найдите"s.length();
5  if (s1.compare(s2) < 0) {
6    cout << "Вторая строка больше первой\n";
7  }
8  size_t end = string::npos,
9         pos1 = s1.length() - nlen, pos2 = 0;
10 if (s1.compare(pos1, end, s2, pos2, nlen) == 0) {
11   cout
12     << "слово 'найдите' есть в обеих строках\n";
13 }
```

Строка **4** демонстрирует *строковый литерал*: добавление суффикса **s** к С-строке превращает её в объект типа **string**. Для их доступа нужно «подгрузить» специальное пространостово имён (строка **1**).

# C++: остальные операции с текстом

Частичное сравнение с C-строками

```
(4) int str.compare(base_str)
(5) int str.compare(size_t pos, size_t len,
                    base_str)
(6) int str.compare(size_t pos, size_t len,
                    base_str, size_t base_len)
```

, где **str** - переменная типа **string**, которую сравниваем. А **base_str** - «базовая» строка.

4. полное сравнение строк **str** и **base_str**
5. сравнение фрагмента внутри **str**, начиная с символа на позиции **pos** и длиной **len**, со строкой **base_str**
6. сравнение фрагментов из **str** и **base_str**. Для базовой строки можно указать только длину фрагмента для сравнения через аргумент **base_len**

Частичное сравнение с C-строками

```
1  string s1 = "два отличия найдите",
2  size_t nlen = "найдите"s.length();
3
4  size_t end = string::npos,
5         pos1 = s1.size() - nlen;
6
7  char base_str[] = "найдите что-нибудь";
8
9  if (s1.compare(pos1, end, base_str, nlen) == 0) {
10   cout << "Строки равны только по "
11           "слову 'найдите' на "
12           "соответствующих позициях\n");
13 }
```

# C++: остальные операции с текстом

Вставка на указанную позицию - метод **insert**

```
(1) str.insert(size_t pos, other_str)
(2) str.insert(size_t pos, other_str,
                size_t o_pos, size_t o_len)
(3) str.insert(size_t pos, base_str)
(4) str.insert(size_t pos, base_str, size_t base_len)
(5) str.insert(size_t pos, size_t count, char sym)
```

1. Вставляет строку **other_str** в **str** сразу **перед** номером символа, заданного аргументом **pos**
2. Вставляет фрагмент из **other_str**, длиной **o_len** и начиная с символа **o_pos** в **str**
3. Вставляет строку **base_str** в **str** перед символом с номером **pos**.
4. Вставляет фрагмент строки **base_str**, длиною **base_len**, в **str** перед символом с номером **pos**.
5. Вставляет в строку **str** символ **sym** в количестве **count** штук перед символом с номером **pos**.

# C++: остальные операции с текстом

Вставка на указанную позицию - метод **insert**

```
1 string s1 = "Что дела?";
2 size_t w_len = strlen("Что");
3
4 s1.insert(w_len + 1, "за ");
5
6 // Напечатает "Что за дела?"
7 cout << s1 << "\n";
```

# C++: остальные операции с текстом

Выделение фрагмента строки - метод **substr**

```
string str.substr(size_t start,
                  size_t len = string::npos)
```

**start** - переменная типа **size_t**, указывающая позицию первого символа подстроки. **len** - количество символов для извлечения.

```
1  string s1 = "Phase transitions are "
2              "great part of physics";
3  size_t len1 = "Phase transitions are "s.length(),
4         len2 = "great"s.length();
5
6  string s2 = s1.substr(len1, len2);
7  // Печатаем: "great"
8  cout << s2 << "\n";
9
10 string s3 = s1.substr(len1 + len2 + 1);
11 // Печатаем: "part of physics"
12 cout << s3 << "\n";
```

# C++: остальные операции с текстом

Замена части текста в строке - метод **replace**

```
(1) str.replace(size_t pos, size_t len, other_str)
(2) str.replace(size_t pos, size_t len, other_str,
                size_t o_pos, size_t o_len)
(3) str.replace(size_t pos, size_t len, base_str)
(4) str.replace(size_t pos, size_t len, base_str,
                size_t base_len)
(5) str.replace(size_t pos, size_t len, size_t count,
                char sym)
```

1. В строке **str** символы в количестве **len** штук (сколько символов из исходной строки удаляем), начиная с номера **pos**, заменяются на строку **other_str**
2. Аналогично, но замена происходит на фрагмент из **other_str**, длиной **o_len** и начиная с символа **o_pos**
3. Замена нужного количества символов на строку **base_str**
4. Замена нужного количества символов на **base_len** символов из строки **base_str**
5. Замена происходит на символ **sym** в количестве **count**

Замена части текста в строке

```
1 string str = "сегодня будет абракадабра!";
2 size_t w_len = "абракадабра!".length();
3
4 str.replace(str.size() - w_len,
5               w_len, "всё хорошо");
6 cout << "И у нас " << str << "\n";
```

# C++: остальные операции с текстом

Для ввода строк (не отдельных «слов») используется функция **getline** из заголовочного файла **<string>**

```
getline(input_stream, string& str_to_fill,
        char separator = '\n');
```

- Функция считывает все введённые символы в строку, которую указали в аргументе **str_to_fill**
- input_stream – объект потока для чтения. Для консольного ввода это всегда **std::cin** (определён в **<iostream>**)
- Возможно задать разделитель **separator** - символ, **после которого** текст считан не будет. Сам разделитель в строку **str_to_fill** не помещается и не учавствует в дальнейших операциях ввода.

Пример ввода двух строк с терминала

```
1 string s1, s2;
2
3 cout << "Введите первую строку: ";
4 getline(cin, s1);
5 cout << "Введите вторую строку: ";
6 getline(cin, s2, '*');
7
8 cout << "Первая строка: [[" << s1 << "]]\n";
9 cout << "Вторая строка: [[" << s2 << "]]\n";
```

В строку **s1** попадут все символы до первого переноса строки, в **s2** - все, предшествующие символу «*».
Более того, во втором случае ввод не закончится, пока среди символов не появится «звёздочка».

# C++: остальные операции с текстом

Из-за особенностей форматированного ввода, после каждой операции чтения >> из потока ввода, в нём остаётся символ переноса строки (**\n**)

```
1 double num;
2 cin >> num;
3
4 string s1;
5 getline(cin, s1);
6
7 cout << boolalpha << s1.empty() << endl;
```

В результате строка **s1** всегда остаётся пустой в данном фрагменте.

Первый способ исправить ситуацию – метод **ignore** для потока.

```
1 #include <limits>
2
3 double num;
4 cin >> num;
5 cin.ignore(
6   numeric_limit<streamsize>::max(), '\n');
7
8 string s1;
9 getline(cin, s1);
10
11 cout << "{{" << s1 << "}}" << endl;
```

Второй – вспомнить приём из языка C

```
1 #include <limits>
2
3 double num;
4 cin >> num;
5 while (cin.get() != '\n');
6
7 string s1;
8 getline(cin, s1);
9
10 cout << "{{" << s1 << "}}" << endl;
```

Цикл в **4** строке также удалит все символы из потока до первого переноса строки.

# C++: остальные операции с текстом

Преобразование строк в числа возможно с помощью функций из библиотеки <**string**>

```
(1) int    std::stoi(  str )
(2) size_t std::stoul( str )
(3) double std::stod(  str )
```

Данные три функции пытаются преобразовать переданную им строку в соответствующее числовое значение (целое со знаком, целое без знака, действительное число). Если преобразование не удалось - строка содержала некорректное число - то происходит ошибка времени выполнения.

Преобразование строк в числа, пример

```
1  string s_double = "456.7788",
2         s_int    = "-7485",
3         s_size_t = "313377317135";
4
5  double d_num  = stod(  s_double );
6  int    i_num  = stoi(  s_int );
7  size_t sz_num = stoul( s_size_t );
8
9  cout << d_num << " " << i_num << " "
10      << sz_num << endl;
```

# C++: остальные операции с текстом

Обратное преобразование числа в строку можно сделать с помощью функции **to_string** (<**string**>)

```
string std::to_string( number_value );
```

Данная функция принимает число (целое, действительное) и возращает его представление в виде строки. Для действительных чисел число знаков после запятой **ограничено шестью**.

```
1  string s1;
2  double real_num = 456.3247899;
3  int integer     = -899;
4  size_t natural  = 13340089;
5
6  s1 += to_string(real_num) + "##";
7  s1 += to_string(integer) + "##";
8  s1 += to_string(natural);
9
10 cout << s1 << endl;
```

# C++ и текст: резюме

- Работа с текстом становится гораздо удобнее с библиотекой <**string**>
- **string** в качестве символов использует тип **char**
- **string** позволяет свести многие операции со строкой к манипуляции позициями фрагментов строк (вместо указателей для C-строк)
- В функции/методы строки лучше всего передавать в виде ссылки (константной/неконстантной) во избежаниия ненужного копирования
- Больше полезных методов для типа **string** можно ▸ найти тут

# Потоковый ввод-вывод и C++

# C++: потоковый ввод/вывод

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. **строк** – 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

# C++: потоковый ввод/вывод

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)
- Объекты потоков сами освобождают внутренние ресурсы при выходе из области видимости

Далее идёт демонстрация работы ввода-вывода на примере файловых потоков

# C++: файловый ввод/вывод

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

1. **ifstream** - для *ввода (чтения)* информации из файла.
2. **ofstream** - для *вывода (записи)* информации в файл.
3. **fstream** - для одновременных операций как *чтения*, так и *записи* из/в файл(а).

которые доступны через подключение заголовочного файла:

```
1 #include <fstream>
```

И про **std** не забываем.

# C++: файловый ввод

Создание файлового потока для ввода: два конструктора

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

1. (1) - конструктор по умолчанию. Создаёт объект потока для ввода из файла, но не связывает его ни с каким реальным файлом
2. (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 // Поток читает данные из config.dat
5 ifstream config_file2{"config.dat"};
6
7 // Вместо имени файла может быть полный путь
8 string file_name = "C:\test\my_config.txt";
9 ifstream config_file3{file_name};
```

Режим открытия файла (**open_mode**) определяется набором битовых 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 открытиии 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};
```

# C++: файловый ввод

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
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 ) { ... };
```

1. Метод **is_open** - возращает 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 }
```

2. 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 }
```

Использование объекта в контексте, требующим логического значения, возможно, если перегружен *оператор приведения* к типу **bool**

```
1  class SomeType
2  {
3  public:
4    explicit operator bool()
5    { return true; }
6  };
7
8  SomeType var;
9  if (var) {
10   std::cout << "var is true" << std::endl;
11 }
12
13 // compile error now,
14 // but OK without explicit
15 // bool value = var;
```

# C++: файловый ввод

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 возращает ссылку на тот поток, из которого происходит чтение

# C++: 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 ) {
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 }
```

# C++: файловый ввод/вывод

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. Проверка, что всё хорошо (в том числе, не достигнут ещё конец файла)

   ```
   bool stream_var.good();
   ```

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 ) {
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     // Работаем с переменными
12   }
13 }
```

# C++: файловый ввод

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 ) {
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 }
```

# C++: файловый ввод

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**

# C++: файловый ввод

Чтение символов в C-строку

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

**getline**:

- 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 учавствует в последующих операциях чтения данных.

**read**:

- Читаем как **count** символов и записываем их в переменную **str**. Никаких символов конца строки не добавляется, символ переноса строки считывается как отдельный элемент.

# C++: файловый ввод, 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    // Печатает "Very im"
6    cout << str << '\n';
7
8    char word[5];
9    input_file.getline(word, 5);
10   // Печатает "port"
11   cout << word<< '\n';
```

# C++: файловый ввод

## Посимвольное чтение файла

```
Что
   говорить                говорить...
        когда нечего
```

```
1  char sym;
2  ifstream in_file{"my_text.txt"};
3  if ( in_file ) {
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 ) {
11   while ( in_file.get(sym) ) ) {
12     cout << sym;
13   }
14 }
```

# C++: файловый ввод

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 }
```

# C++: файловый ввод

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"↩
       ;
```

# C++: файловый ввод

```
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 учавствует 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.

# C++: файловый ввод

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
```

# C++: файловый ввод

Что можно выделить из описания файла на предыдущем слайде?

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

# C++: файловый ввод

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 }
```

# C++: файловый ввод

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 попробывать считать какие-нибудь данные снова.

# C++: файловый ввод

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 }
```

# C++: файловый вывод

- Запись данных в файл осуществляется с помощью класса **ofstream** (потоковый 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**

# C++: файловый вывод

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 }
```

Относительно интересный манипулятор – quoted

```
#include <iomanip>

string s1 = "simple string";

cout << quoted(s1) << endl;
cout << quoted(s1, '@', '!') << endl;

string s2;
cin >> quoted(s2);
cout << "s2: " << s2 << endl;
```

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

# C++: 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 возращает 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 }
```

# C++: файловый ввод/вывод

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**, **write** .

# Работа со строковыми потоками

# C++: строковые потоки ввода/вывода

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

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

которые доступны после следующего include'а:

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

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

# C++: строковые потоки ввода

Создание потока для ввода:

```
1 istringstream input;
```

Но созданный такми 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"};
```

# C++: 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  istringstream input{str};
8
9  cout << "Введённые слова:\n";
10 string word;
11 while (input >> word) {
12     cout << '{' << word << '}' << endl;
13 }
```

# C++: строковые потоки ввода

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 }
```

# C++: строковые потоки ввода

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
```

# C++: 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    istringstream 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 << "Среденее в " << (i + 1)
34             << "точке: " << averages[i] << endl;
35      }
36    }
37  }
```

# C++: 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 такми образом поток по умолчанию содержит внутри себя пустую строку, к которой будет добавляться другая текстовая информация. Также можно сразу задать содежимое строки, которую нужно дополнить чемн-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"};
```

# C++: 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  ostringstream 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;
```