

# **Структурирование программы и её взаимодействие с пользователем**

Курс «Разработка ПО систем управления»

Кафедра управления и информатики НИУ «МЭИ» Весенний  
семестр 2018 г.

# Определение функции

Тип возвращаемого значения.

Имя функции.

**double** area (  
    **double** width,  
    **double** height )

Параметры и их типы.

- Тип указывается каждому!

Возврат значения  
и выход из функции.

тело  
функции

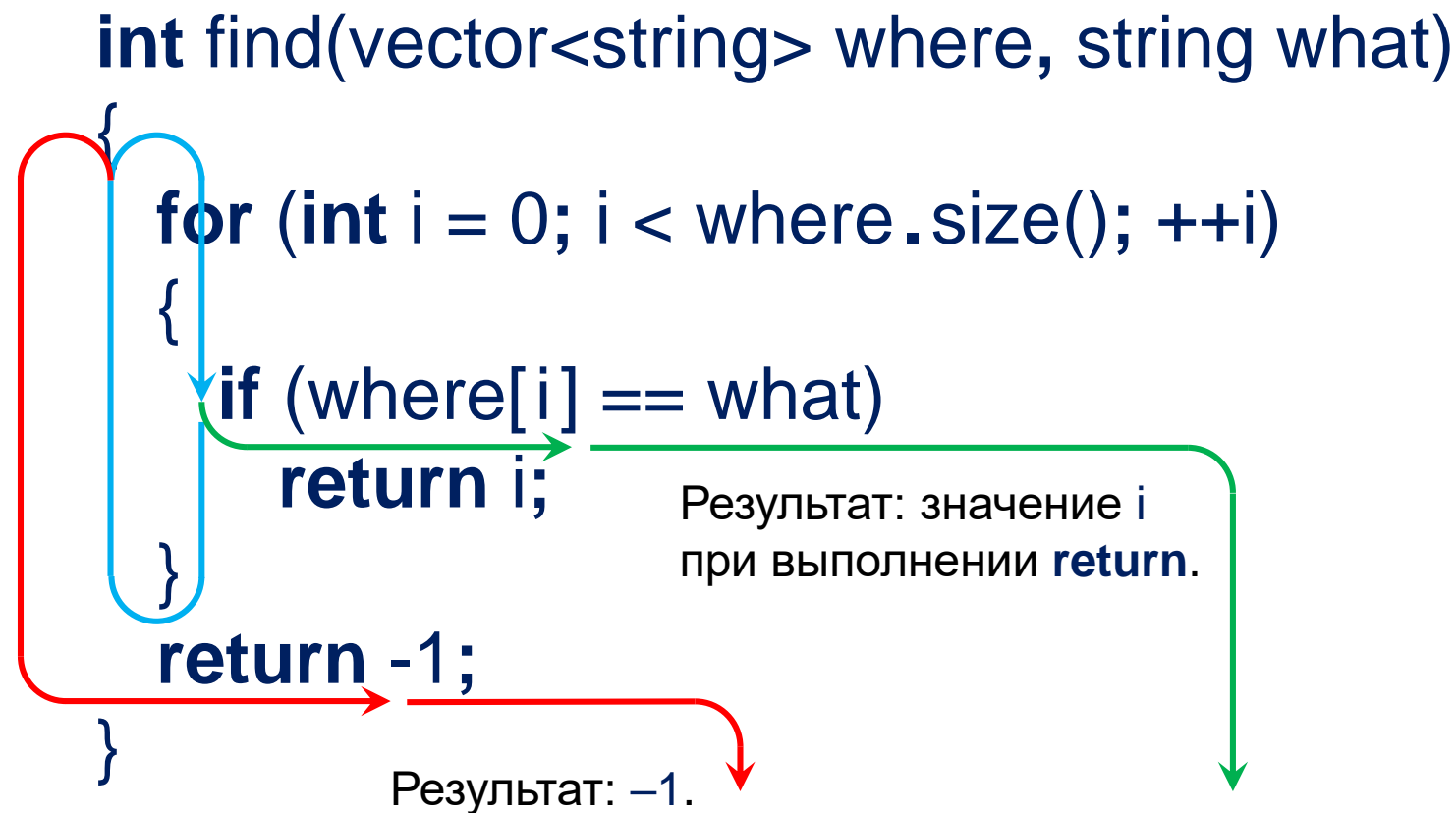
{  
}  
}

**return** width \* height;

**double** S = area ( 4, 5 ); // S == 20

area ( 3, 2 );                      // 6 (игнорируется)

# Пример функции на C++



# Оператор **return**

- Оператор **return** *X*:
  - указывает, что возвращаемое значение — *X*;
  - производит выход из функции.
- Не-**void** функции обязаны вернуть значение.
  - Иначе — не ошибка, но *опасное* предупреждение!

# Выходные переменные

Функция ничего  
не возвращает  
(как процедура).



```
void solve_quadric_equation(  
    double a, double b, double c,  
    double& x1, double& x2)  
{  
    double const D = b*b - 4*a*c;  
    x1 = (-b + sqrt(D)) / (2*a);  
    x2 = (-b - sqrt(D)) / (2*a);  
}
```

& — амперсанд

```
double x1, x2;  
solve_quadric_equation(1, 3, 2, x1, x2);  
// x1 == -1, x2 == -2
```

# Передача по ссылке

- Удобна для возврата нескольких значений.
- Проблема — читаемость:

```
double a = 1, b = 3, c = 2, x1 = 0, x2 = 0;  
solve_quadric_equation(a, b, c, x1, x2);  
// Какие переменные изменились?
```

# Параметр-ссылка

- Проблема — обязательность всех аргументов:

```
void get_statistics(  
    vector<double> samples,  
    double & mean, double & variance)  
{  
    // Расчет мат. ожидания и дисперсии.  
}
```

```
vector<double> data { 1, 2, 3, 4, 5 };  
double mean;  
double variance;  
get_statistics(data, mean, variance);
```

Не нужна!



# Ссылка как тип данных

- Ссылка — новое имя ячейки памяти.

```
double x = 1;
```

```
double y = 3;
```

```
double& z = x;
```

```
z = 2;
```

```
// x == 2, y == 3, z == 2
```

```
z = y;
```

```
// x == 3, y == 3, z == 3
```

```
y = 4;
```

```
// x == 3, y == 4, z == 3
```

Амперсанд перед именем переменной.

Инициализация:

- «привязка» к значению (переменной);
- обязательна
  - иначе — «новое имя» для чего?

Действия над ссылкой  
равнозначны действиям  
над привязанной переменной.

Привязку изменить нельзя.



# Применение ссылок

- Сокращение кода:
  - **double&** middle = data[data.size() / 2];  
middle = 42;  
// data[data.size() / 2] == 42
  - **double&** x = change\_a\_or\_b ? a : b;  
x += 2;
- Неизменяемые ссылки:
  - **const double&** middle = data[data.size() / 2];  
~~middle = 42;~~
  - Неизменяемость всегда относится к значению.

# Неизменяемые параметры

Будет создана копия значения **a** и помещена в **x**.

↓  
**void f(int x) {**  
    **x = 42;**  
    **// x == 42**  
**}** ← Копия разрушается.

**void f(const int x) {**  
    **// ...**  
**}** ↑ Копию нельзя изменить  
    • и обычно не нужно.

**int a = 0;**  
**f(a);**  
**// a == 0**  
Действия над копией не влияют на аргумент.

- А если **x** — вектор или строка?
  - Большого размера?
- Зачем вообще копия?
  - Нужна независимость **x** и **a**.
  - Обычно нужна неизменяемость.

# Передача без копирования

- Передача по ссылке:

```
void function ( vector<int>& data ) { ... }
```

- Нет копирования.
- Аргумент и data связаны.

- Передача по неизменяемой ссылке:

```
void function ( const vector<int>& data ) { ... }
```

- Копирования нет.
- Случайно изменить data нельзя.
  - Изменять параметры — плохая практика!
- Имеет смысл использовать по умолчанию.
  - Кроме **int**, **double**, ... (пользы нет, вреда — тоже).

# Рекурсия

- Вызов функцией самой себя.
- Для случаев, когда



путь(от Новокосино до Авиастроительной) =

«Новокосино — Новогиреево» + путь(от Новогиреево до Авиастроительной)

# Рекурсивный вызов

`power(2, 3); // 23 = 8`

$$f(a, n) = a^n = \begin{cases} a \cdot a^{n-1}, & n > 0 \\ 1, & n = 0 \end{cases} =$$

$$= \begin{cases} a \cdot f(a, n-1), & n > 0 \\ 1, & n = 0 \end{cases}$$

условие окончания

```
double power(2, 3) {  
    if (3 == 0)  
        return 1;  
    return 2 * power(2, 3 - 1);  
}
```

```
double power(2, 2) {  
    if (2 == 0)  
        return 1;  
    return 2 * power(2, 2 - 1);  
}
```

```
double power(2, 1) {  
    if (1 == 0)  
        return 1;  
    return 2 * power(2, 1 - 1);  
}
```

```
double power(double a, int n) {  
    if (n == 0)  
        return 1;  
    return a * power(a, n - 1);  
}
```

```
double power(2, 0) {  
    if (0 == 0)  
        return 1;  
    return 2 * power(2, 0 - 1);  
}
```

# Рекурсия (продолжение)

- ❑ Вызов функции расходует часть ограниченной области памяти — стека.
  - Этот расход возвращается по выходе из функции.
  - Глубокая рекурсия сильно расходует стек.
    - Бесконечная рекурсия невозможна.
    - Ошибка: «Stack overflow» («переполнение стека»).



## ❑ Прямая рекурсия



В каком порядке  
описывать функции?

## Косвенная рекурсия

```
bool is_even ( unsigned int n ) {  
    return n == 0 || is_odd ( n - 1 );  
}  
bool is_odd ( unsigned int n ) {  
    return n != 0 && is_even ( n - 1 );  
}
```

# Объявление и определение

```
double get_mean ( const vector<double>& xs );
```

← **Объявление** функции (прототип).

```
int main() {  
    vector<double> data { 1, 2, 3, 4, 5 };  
    cout << "Mean is " << get_mean(data);  
}
```

← Благодаря объявлению, компилятор уже «знает», что такая функция есть.

```
double get_mean ( const vector<double>& xs ) {  
    double mean = 0;  
    for (const double& x : xs) {  
        mean += x;  
    }  
    return mean / xs.size();  
}
```

← **Определение** функции.

← Копия значения в векторе не нужна, менять его не нужно.

Функции можно перегружать - определять несколько функций с одинаковым именем, но различными параметрами (порядком следования, количеством, типом)

```
double get_mean ( const vector<int>& xs );
```

# Какими должны быть функции?

```
int square(int x)
{
    return x * x;
}
```

- ✓ Одна задача;
- ✓ ничего лишнего;
- ✓ полезна широко.

```
int square(int& x, int& count)
{
    cout << "Enter element #" << count << ": ";
    cin >> x;
    count++;
    return x * x;
}
```

Задачи:

- 1) Ввод и вывод,
  - а если не нужны?
- 2) подсчет;
  - зачем?
- 3) возведение в квадрат.

## 1) Повторно используемыми (reusable).

- Решать одну задачу.
- Не иметь побочных эффектов:
  - зависеть только от входных данных (не от ввода, времени и т. п.);
  - выдавать результат только возвращаемым и выходными значениями.



# Какими должны быть функции?

2) Могут обозначать логику работы программы.

«Как съесть слона? — По кусочкам!»

Расчет корреляции  $\vec{x}$  и  $\vec{y}$ :

1) ввести  $N$ ,  $\vec{x}$ ,  $\vec{y}$ ;

2) вычислить  $m_x$  и  $m_y$ ;

3) вычислить  $s_x$  и  $s_y$ ;

4)  $S = \sum_{i=0}^{N-1} (x_i - m_x)(y_i - m_y)$ ;

5)  $cov(x, y) = S / (N-1)$ ; Цикл?

6)  $r_{xy} = \frac{cov(x, y)}{s_x s_y}$ .

`vector < double > input (  
 unsigned int how_many )`

`double get_mean (  
 const vector<double> & data)`

`double get_stdev (  
 const vector<double> & data,  
 double mean)`

# Декомпозиция

**unsigned int** N;

- 1) **cin** >> N;  
vector < **double** > x = input ( N );  
vector < **double** > y = input ( N );
- 2) **double** m\_x = get\_mean ( x );  
**double** m\_y = get\_mean ( y );
- 3) **double** s\_x = get\_stdev ( x, m\_x );  
**double** s\_y = get\_stdev ( y, m\_y );
- 4) **double** sum = 0;  
**for** ( **unsigned int** i = 0; i < N; ++i ) {  
    sum += ( x[i] - m\_x ) \* ( y[i] - m\_y );  
}
- 5) **double** covariance = sum / ( N - 1 );
- 6) **double** correlation = covariance / ( s\_x \* s\_y );

Расчет корреляции  $\vec{x}$  и  $\vec{y}$ :

- 1) ввести  $N$ ,  $\vec{x}$ ,  $\vec{y}$ ;
- 2) вычислить  $m_x$  и  $m_y$ ;
- 3) вычислить  $s_x$  и  $s_y$ ;
- 4)  $S = \sum_{i=0}^{N-1} (x_i - m_x)(y_i - m_y)$ ;
- 5)  $cov(x, y) = S / (N - 1)$ ;
- 6)  $r_{xy} = \frac{cov(x, y)}{s_x s_y}$ .

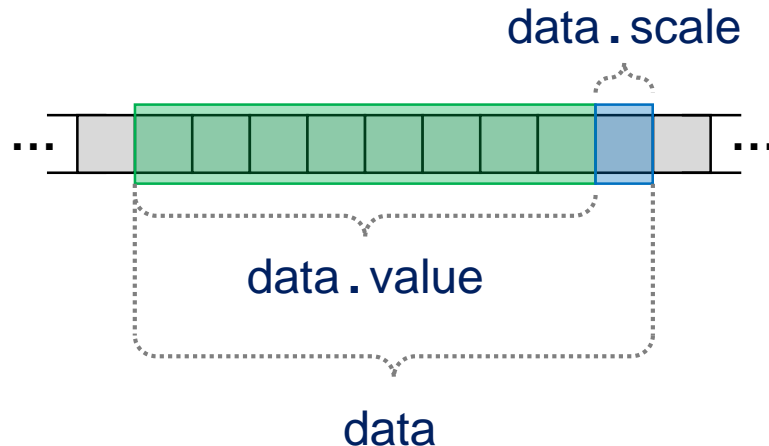
# Структуры

- Хранят вместе несколько именованных значений разных типов.

- **struct** Temperature

```
{  
    double value;  
    char scale;  
};
```

- Temperature data;  
data.value = 273.15;  
data.scale = 'K';  
cin >> data.value >> data.scale;  
cout << data.value - 273.15 << 'C';



# Перегрузка операторов

```
Temperature boiling { 100, 'C' };
```

```
if (data > boiling) { ... }
```

```
Temperature mean { 0, 'C' };
```

```
mean = mean + data;
```

<, +, = - отдельные операторы!

Особое имя функции.

Типы результата и параметров должны быть точно такими.

```
bool operator > (  
    const Temperature& lhs,  
    const Temperature& rhs)  
{  
    return lhs.value > rhs.value;  
}
```

```
Temperature operator + (  
    const Temperature& lhs,  
    const Temperature& rhs)  
{  
    return {  
        lhs.value + rhs.value,  
        lhs.scale  
    };  
}
```

Предполагается одинаковая шкала для краткости!

«Left-Hand Side» и «Right-Hand Side»

# Вывод пользовательских типов

сокращенная  
форма записи

`cout << x;` → `operator<< (cout, x);`

`cout << x << y;` → `operator<< (operator<< (cout, x), y);`

`1 + 2 + 3` → `(1 + 2) + 3`

тип `cout`\* → `ostream& operator << (`  
`ostream& output, const Temperature& data)`  
`{`  
`output << data.value << data.scale;`  
`return output;`  
`}`

`cout << data; // 237.15K`

\* На самом деле, тип любого стандартного потока вывода.

# Ввод пользовательских типов

```
istream& operator>> (
    istream& input, Temperature& data)
{
    input >> data.value >> data.scale;
    if (data.scale != 'K' || data.value < 0) {
        input.setstate (ios_base::failbit);
    }
    return input;
}
```

тип `cin`\*

Ссылка не **const**, так как `data` изменяется.

Здесь можно выполнить преобразования и проверку ввода.

Как сообщить об ошибке?..

А чтобы это работало?

- Temperature data;  
**while** (cin >> data) { ... }
- **if** (! (cin >> data)) {  
    cout << "Incorrect temperature input!";  
}

Окончание ввода `cin` отследит сам.

# Файловый ввод и вывод

Или:

```
ifstream input;  
input.open ( "file.txt" );
```

- Ввод:

```
ifstream input ( "file.txt" );  
input >> temperature;
```

- Вывод:

```
ofstream output ( "file.txt", режим );  
output << "Result: " << result << '\n';
```

- Заккрытие файла — автоматически или `.close()`.

- Работа с файлами подобна работе с `cin` и `cout`.

- Ничего или `ios::out`.
- `ios_base::ate` — дописывать в конец,
  - Append (Output) в Pascal;
- `ios_base::trunc` — очистить файл перед записью,
  - Truncate (Output) в Pascal.

# Форматный вывод

**double** value = 12.34567;

■ cout << setprecision (2);

Действует на **cout** все время после установки.

■ cout << value

// 12

...значащих цифр

<< fixed

<< value // 12.35

...цифр после запятой

<< scientific

<< value // 1.23e+01

...цифр после запятой  
в мантиссе

<< defaultfloat << value; // 12

Действуют только  
на следующее  
выводимое  
значение.



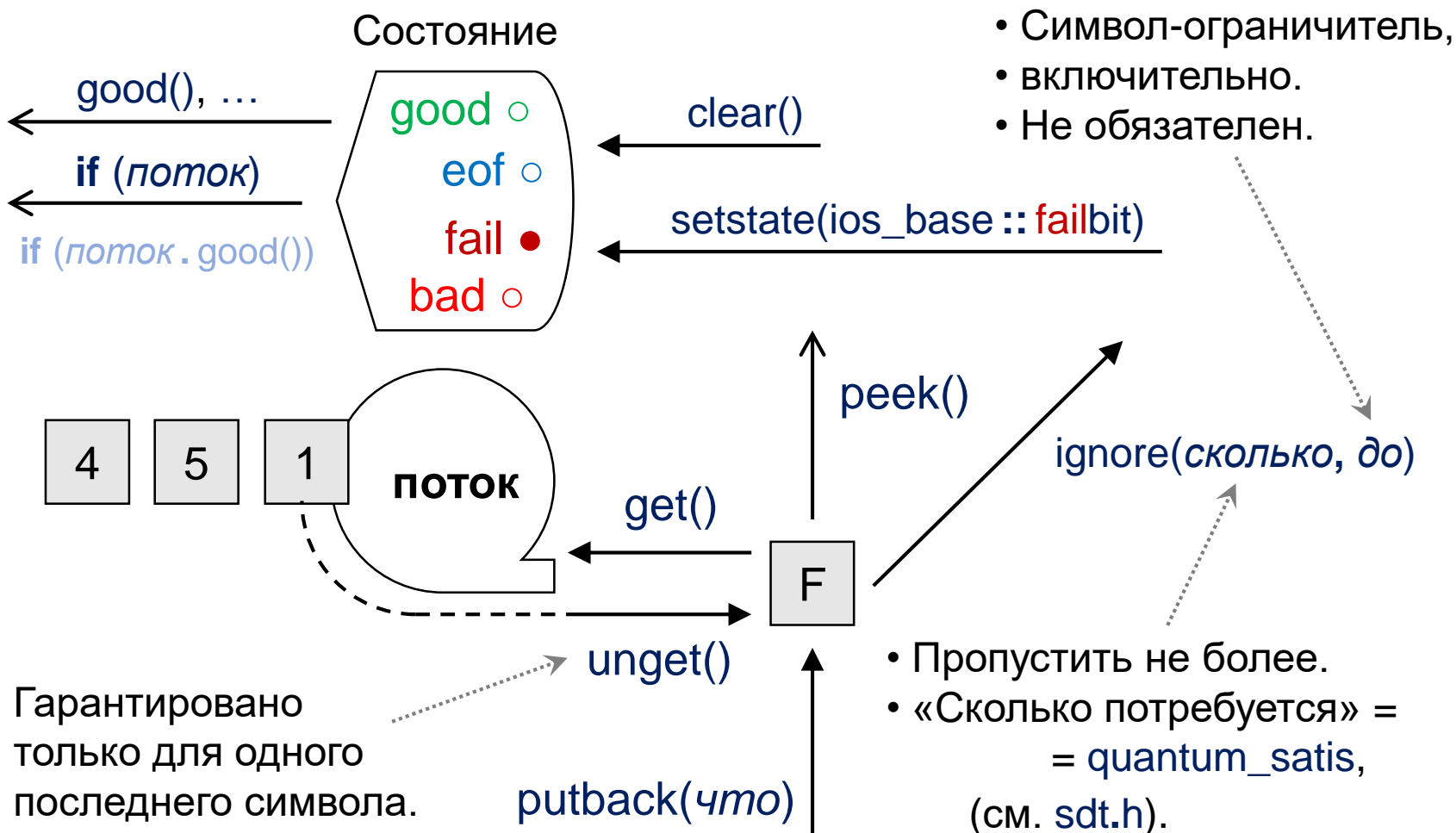
# Форматный вывод

## (продолжение)

- `cout << setw(15) << setfill('.') << left << "Code:"`  
`<< setw(4) << setfill('0') << right << 12;`
- `Code:.....0012`
- `cout << endl;`
  - `cout << '\n';`  
`cout.flush();`

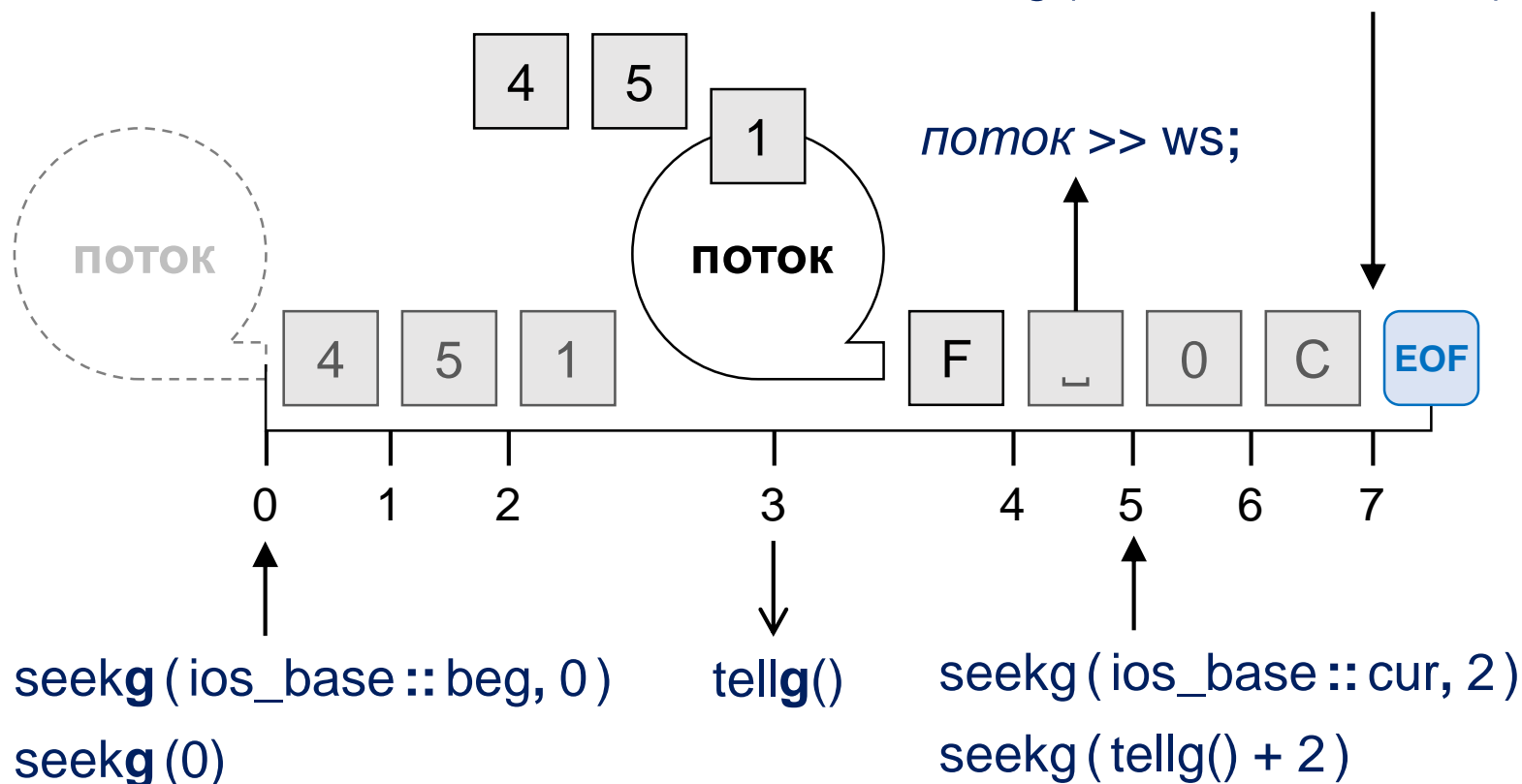
Форматный вывод в C++ устроен сложнее, чем в C, но на самом деле он гибче, т. к. параметры форматирования легко менять во время работы.

# ПОТОКОВЫЙ ВВОД



# Перемещение в потоке

Для вывода: `tellp()`, `seekp()`.



# Потоки в памяти, или «как превратить строку в число?»

- **string** input;  
getline(cin, input);  
stringstream source(input);

Из строки вычитываются слова,  
разделенные пробелами.

size\_t count = 0;  
**for** (**string** word; source >> word; ++count);  
cout << "Word count: " << count << '\n';

- **int** parse (**string** text) {  
stringstream stream ( text );  
**int** result;  
stream >> result;  
**return** result;  
}

Тело цикла пустое:  
++count и есть подсчет слов.

- stringstream Чтение и запись.
- istream Only чтение.
- ostream Only запись.

# Функции **printf()** и **scanf()**

- Функция **printf()** позволяет выводить информацию на экран при программировании в консольном режиме.
- Функция **printf()** не работает со **string** напрямую:  

```
string hello = "hello";  
printf( "%s", hello.c_str() );
```
- Подробная справка по форматной строке: [[cppref](#)].
- Для ввода информации с клавиатуры можно использовать функцию **scanf()**:
- **float** value;  

```
scanf( "%f", &value );
```

# Работа с файлами в С

Тип-указатель (не разыменовывается), обозначающий открытый файл.

`FILE* file = fopen ( "name.ext", "r" );`

***Считать из file в number:***

`fscanf ( file, "%d", &number );`

Режим доступа [[cppref](#)]:

"r" — чтение текста;

"w" — перезапись файла;

"a" — дозапись в конец.

***Строка из 120 символов '\0' (Это специальный символ «конец строки»):***

`string line(120, '\0');`

***Считать из file определенное количество символов (в данном случае = длине строки) и записать в line:***

`fgets (line.data(), line.size(), file );`

# Работа с файлами в С (продолжение)

**Записать в *file* указанную последовательность символов в заданном формате:**

```
fprintf ( file, "%4.2f", 3.14 );
```

**Записать в *file* строку *line*:**

```
fputs ( line.c_str(), file );
```

**Перемещение на 0 символов от конца, то есть к концу файла. См. [\[cppref\]](#).**

```
fseek ( file, 0, SEEK_END );
```

***ftell()* - значение указателя текущего положения потока.**

```
long position = ftell ( file );
```

```
fclose ( file );
```

# Параметры командной строки

Приглашение (prompt).

\$ g++ -o program main.cpp



Имя программы  
(исполняемого файла).

Аргументы командной строки.

- Программа может получить аргументы, с которыми вызвана.
- И имя, под которым вызвана (как аргумент № 0).
- Аргументы с дефисами в начале иногда называют **опциями (options)**.



# Разбор параметров командной строки

```
int main (int argc, char* argv[])
{
    for (int i = 0; i < argc; i++) {
        cout << "argv[" << i << "] = " << argv[i] << endl;
    }
}
```

\$ **./program** -o option 123 --long-option=value -ab

argv[0] = **./program**


argv[1] = -o

argv[2] = option

argv[3] = 123

argv[4] = --long-option=value

argv[5] = -ab




Для многих программ  
это эквивалентно -a -b,  
но это их внутренняя  
логика!

# Что значит `char* argv[]` ?

- `argv[]` значит, что `argv` — это массив.
  - ...массив значений аргументов (**argument values**).
  - Количество элементов — `argc` (**argument count**).
  - Тип каждого элемента — `char *`
- `char` — это СИМВОЛ.
- `char*` — указатель на СИМВОЛ:
  - Указатель содержит адрес памяти, т. е. место, где хранится что-либо (здесь: символ).
  - Строка — цепочка символов, имеем указатель на первый символ в ней.
- **Итого:** массив мест, где начинаются строки-аргументы.

# Пример разбора параметров командной строки

Неправильно сравнивать  
адрес данных `argv[1]`  
с адресом константы `"-h"`.  
Тип `string` сравнит значения.



```
int main (int argc, char* argv[])  
{  
    if (argc > 1 && string(argv[1]) == "-h") {  
        cout << "Программа вычисляет оценки " <<  
            "математического ожидания и дисперсии."  
    }  
    // ...  
}
```

\$ `./program -h`

Программа вычисляет оценки математического ожидания и дисперсии.

\$ `./program`

Введите количество чисел:

# О командной строке

- На практике командную строку разбирают с помощью библиотек (`getopt`, `Boost.ProgramOptions`).
- С ключом `-h`, `--help`, `-?`, `--usage` принято отображать краткую справку.
- Если типичный запуск требует много опций, имеет смысл сделать текстовый файл настроек.
- В Windows вместо `-f` принято `/F` (`/` вместо `-`). От этого отказываются. Не нужно так делать.

# Литература к лекции

- *Programming Principles and Practices Using C++:*
  - глава 4, раздел 4.5 — функции;
  - глава 6, раздел 6.5 — декомпозиция;
  - глава 8 (пункт 8.5.8 — опционально);
  - пункт 9.4.1 — структуры, раздел 9.5 — перечисления;
  - упражнения к главам 4 и 8.
- *C++ Primer:*
  - глава 2, раздел 2.3 — указатели и ссылки;
  - глава 6 — функции;
  - раздел 19.3 — перечисления;
  - упражнения.