Взаимодействие программы с пользователем и контроль корректности

Курс «Разработка ПО систем управления»

Кафедра управления и интеллектуальных технологий

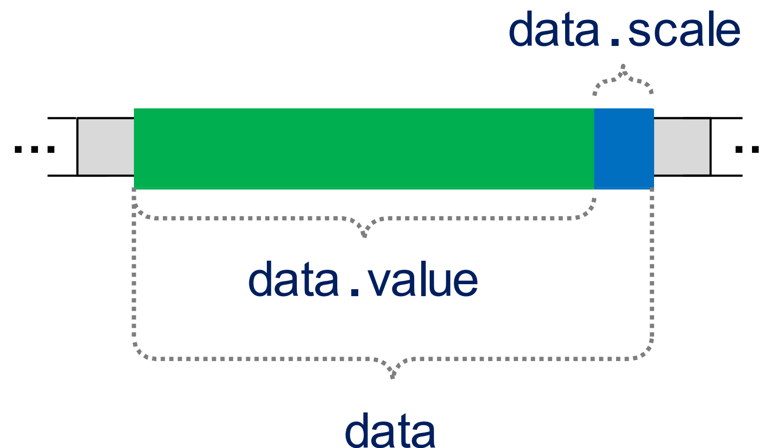
НИУ «МЭИ»

Весенний семестр 2021 г.

© Кафедра УИТ НИУ «МЭИ»

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!

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

Предполагается одинаковая шкала для краткости!

Особое имя функции.

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

Типы результата и параметров должны быть точно такими.

«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

тип cin*

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

```
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;
}
```

Здесь можно выполнить преобразования и проверку ввода.

Как сообщить об ошибке?..

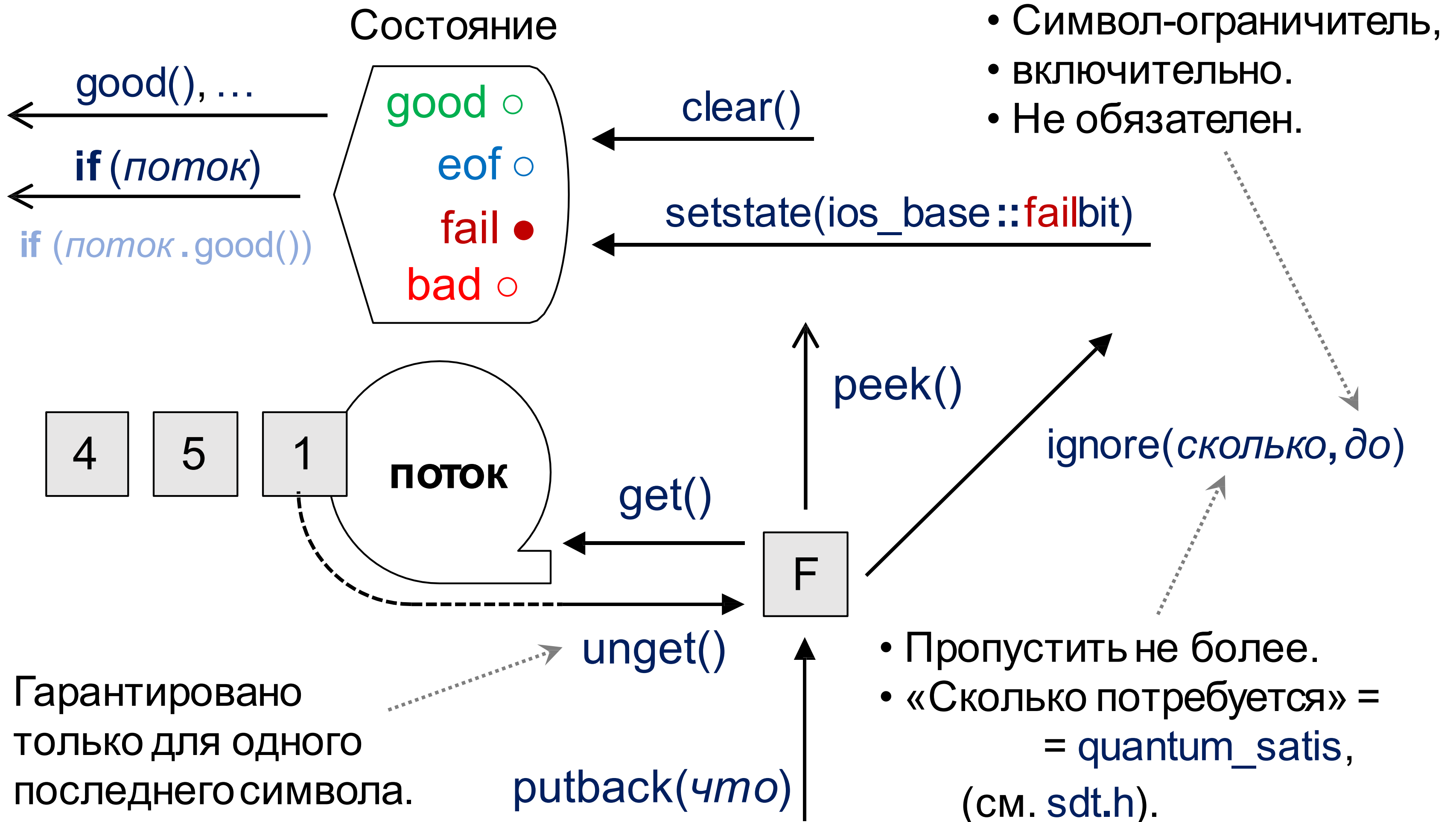
- ```
  Temperature data;
  while (cin >> data) { … }
  ```
- ```
  if (!(cin >> data)) {
      cout << "Incorrect temperature input!";
  }
  ```

А чтобы это работало?

Окончание ввода cin отследит сам.

© Кафедра УИТ НИУ «МЭИ»

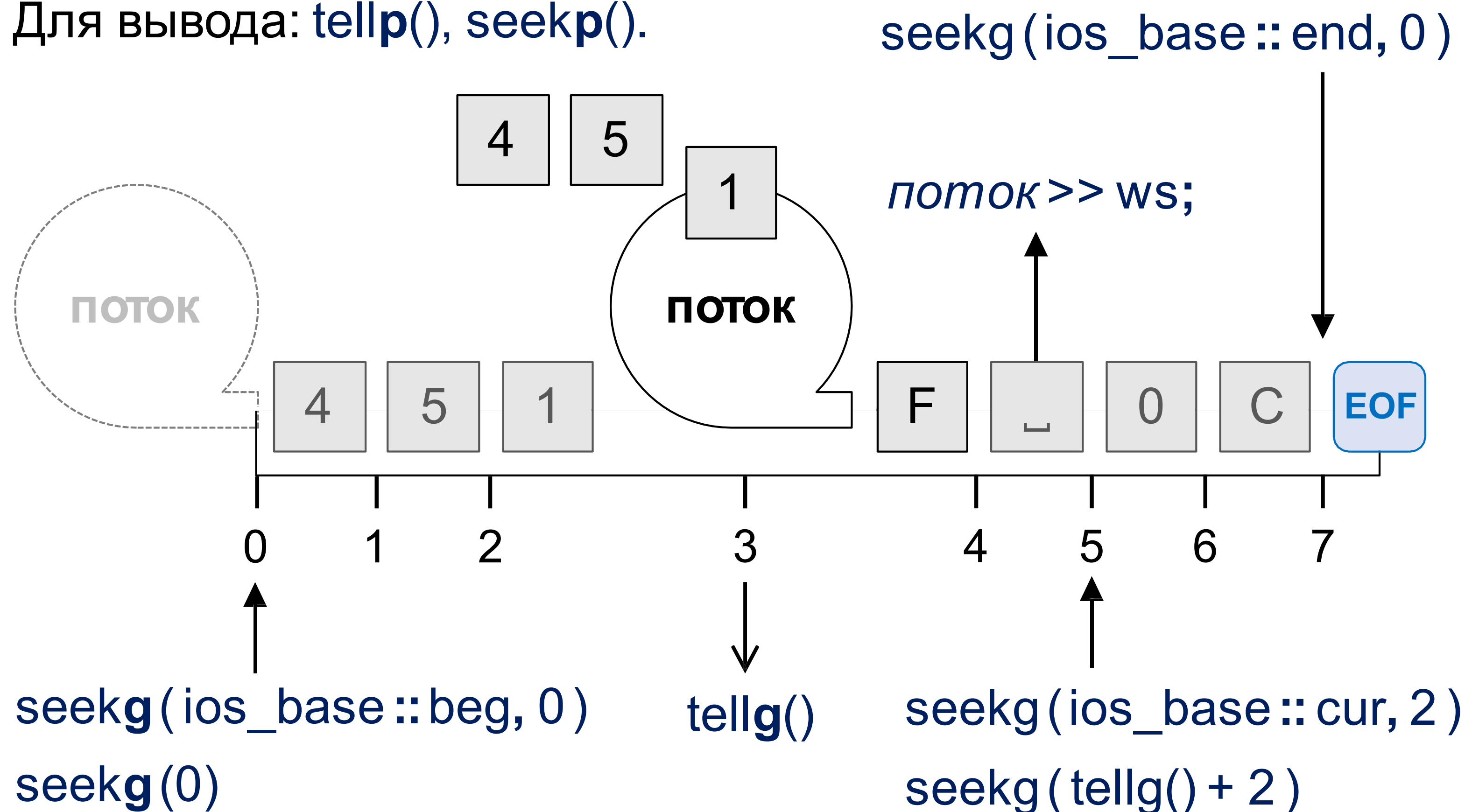
Потоковый ввод



© Кафедра УИТ НИУ «МЭИ»

Перемещение в потоке

Для вывода: tell**p**(), seek**p**().



Форматный вывод

```
#include <iomanip>
double value = 12.34567;
```

- cout << setprecision(2);
- cout << value // 12
  << fixed << value // 12.35
  << scientific << value // 1.23e+01
  << defaultfloat << value; // 12

Действует на cout все время после установки.

…значащих цифр

…цифр после запятой

…цифр после запятой в мантиссе

© Кафедра УИТ НИУ «МЭИ»

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.

Файловый ввод и вывод

- #include <fstream>

Или:

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

- Ввод:

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

- Вывод:

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

- Ничего или ios::out.
- ios_base::app — дописывать в конец,
- ios_base::ate — переместить указатель в конец,
- ios_base::trunc — очистить файл перед записью,

- Закрытие файла — автоматически или .close().
- Работа с файлами подобна работе с cin и cout.

© Кафедра УИТ НИУ «МЭИ»

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

```
string input;
getline(cin, input);

stringstream source(input);

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;
}
```

- stringstream Чтение и запись.
    - istringstream Только чтение.
    - ostringstream Только запись.

Параметры командной строки

Приглашение (prompt).

$ g++ -o program main.cpp

Имя программы (исполняемого файла).

Аргументы командной строки.

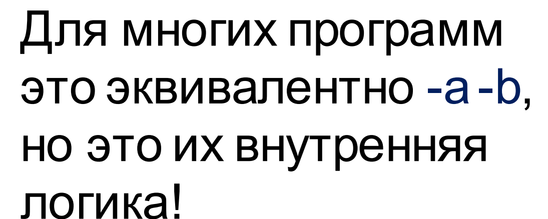
- Программа может получить аргументы, с которыми вызвана.
- И имя, под которым вызвана (как аргумент № 0).
- Аргументы с дефисами в начале иногда называют **опциями (options).**

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
```

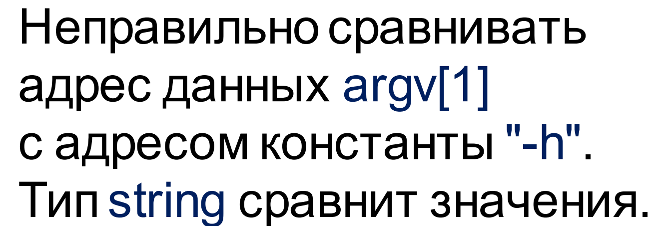


Что значит char* argv[] ?

- argv [] значит, что argv — это массив.
  - …массив значений аргументов (**arg**ument **v**alues).
  - Количество элементов — argc (**arg**ument **c**ount).
  - 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.

© Кафедра УИТ НИУ «МЭИ»

Пример разбора параметров командной строки




```
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.
- В Windows вместо -f принято /F (/ вместо -). От этого отказываются. Не нужно так делать.

Контроль корректности программ

- Статический анализ кода
- Обработка ошибок в штатном режиме
- Защитное программирование
- Модульное тестирование

Статический анализ кода

Суть: поиск потенциальных ошибок без запуска программы.

Есть специальные инструменты,

но первый из них — сам компилятор.

Предупреждения компилятора:

1. Никогда не следует игнорировать.
2. Можно трактовать как ошибки.

© Кафедра УИТ НИУ «МЭИ»

Обработка ошибок: код возврата

```
int convert_temperature (
    double temperature,
    char from, char to,
    double& result )
{
    if ( from != 'K' && from != 'C' )
        return 1;
    //…
}
double kelvins;
switch ( convert_temperature ( celsius, 'C', 'K' , kelvins ) ) {
    case 0:  cout << kelvins << "K\n"; break;
    case 1:  cerr  << "Неизвестная исходная шкала!\n"; break;
    default: cerr  << "Неизвестная ошибка!\n";
}
```

Код	Ошибка
0	Нет ошибки.
1	Неизвестная шкала from.
2	Неизвестная шкала to.
3	temperature < 0 °K

Лишняя переменная — повод ошибиться.

Обработка ошибок: доступ к последней ошибке

```
int last_error = 0;
int get_last_error() { return last_error; }
double convert_temperature(
    double temperature, char from, char to)
{
    if (from != 'K' && from != 'C') {
        last_error = 1;
        return 0.0;
    }
    // …
}
double kelvins = convert_temperature(celsius, 'C', 'K');
switch (get_last_error()) { … }
```

Глобальная переменная. Объявлена вне функций, доступна в любой из них.

Возвращаемое при ошибке значение не имеет смысла. Использовать его

1) возможно, но это некорректно.
2) Проверка кода нужна, но необязательна.

© Кафедра УИТ НИУ «МЭИ»

Защитное программирование

- Defensive programming:

  паранойя как подход к работе.

  Проверять все входные параметры (контрактное программирование)

  Проверять предположения (assumptions)

  о состоянии программы в разных точках.

  Цель: узнать об ошибке как можно ближе к месту её возникновения.

- Fail-fast (ранний выход):

  обнаруживать ошибки как можно раньше;

  при обнаружении — завершаться.

- Стандарты и практики безопасного кодирования.

Unit testing (модульное тестирование)

Код, который проверяет,
что другой код работает правильно.

Обычно отдельная программа, использующая часть основной.

Позволяет проверить, что после изменений
код по-прежнему работает.

Локализует проблему вплоть до проверяемой функции.

Тесты нужно писать в дополнение к коду.

Прохождение тестов не гарантирует, что ошибок нет.

Непрохождение говорит, что они есть.

© Кафедра УИТ НИУ «МЭИ»

assert() из <cassert>

Проверяет условие-аргумент.

Если не выполняется:

печатает сообщение с этим условием;

завершает программу аварийно.

Вне отладочной сборки ничего не делает.

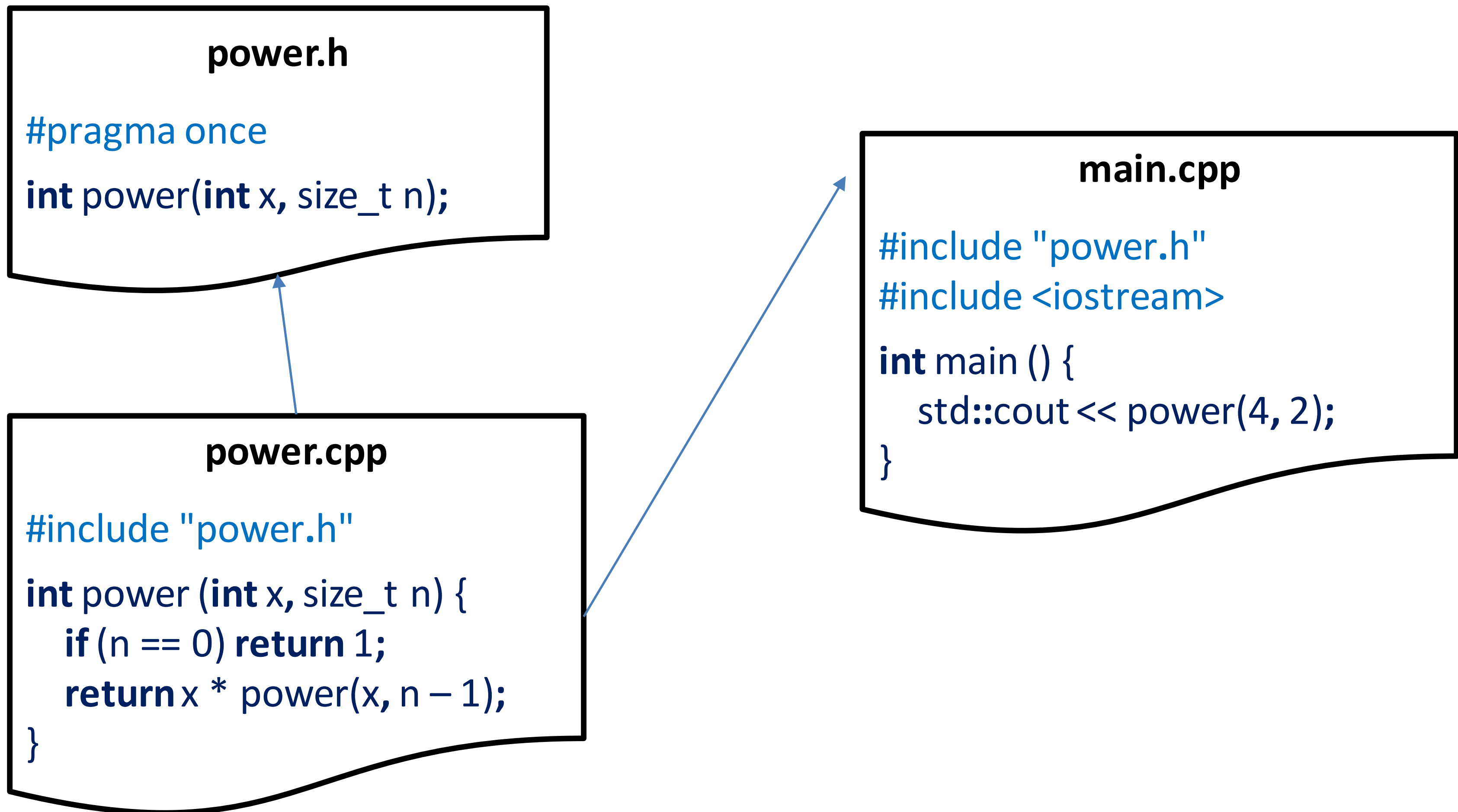
Не влияет на конечную программу.

`#define NDEBUG` (объявить до объявления макроса assert)

```
double square_root(double x) {
   assert(x >= 0);
   // ...
}
```


© Кафедра УИТ НИУ «МЭИ»

Unit testing: пример



© Кафедра УИТ НИУ «МЭИ»

Программа-тест

```
#include "power.h"
#include <cassert>
int main()
{
    assert( power(0, 0) == 1 );       // Возведение в степень нуля.
    assert( power(0, 1) == 0 );
    assert( power(2, 0) == 1 );       // Типичные случаи.
    assert( power(2, 1) == 2 );
    assert( power(2, 4) == 16 );
    assert( power(-1, 0) == 1 );      // Отрицательное основание.
    assert( power(-1, 2) == 1 );
    assert( power(-1, 3) == -1 );
}
```

Польза от модульных тестов (1)

1. Оптимизируем программу:

$$// \ a^n = \begin{cases} 1, & n = 0 \\ \left(a^{\frac{n}{2}}\right)^2, & n \text{ четно} \\ a \cdot a^{n-1}, & n \text{ нечетно} \end{cases}$$

```
int power(int x, size_t n) {
    if (n == 0)
        return 1;
    if (n % 2 == 1)
        return power(x, n / 2) * power(n, n / 2);
    else
        return x * power(x, n – 1);
}
```


© Кафедра УИТ НИУ «МЭИ»

Польза от модульных тестов (2)

2. Прогоним тест (вывод сокращен):

```
1: Assertion failed!
1:
1: Program: C:\cs-17-labs\lecture03\cmake-build-debug\test_power.exe
1: File: C:\cs-17-labs\lecture03\test_power.cpp, Line 12
1:
1: Expression: power(-1, 2) == 1
1/1 Test #1: test_power ......................***Failed   15.59 sec

0% tests passed, 1 tests failed out of 1

Total Test time (real) =  15.62 sec

The following tests FAILED:
	1 - test_power (Failed)
Errors while running CTest
```


© Кафедра УИТ НИУ «МЭИ»

Принципы модульного тестирования

Рассмотренный пример сильно упрощен.
Модульное тестирование шире, чем рассмотрено здесь.

Код должен быть **тестируемым.**

- Функции должны быть независимыми друг от друга.
- Желательны чистые функции.

Тесты должны быть:

1. **Исчерпывающими** — проверять все возможные пути выполнения (execution paths).
   - Покрытие тестами (coverage) — доля кода, который тестируется.
   - Но: тест проверяет *утверждение о результате* работы кода.

2. **Изолированными** — проверять только выбранный фрагмент или случай (тест-пример мог бы стать тремя);
   - вариант: одна проверка (assertion) на тест.

© Кафедра УИТ НИУ «МЭИ»

Литература к лекции

- Более подробное описание процесса сборки с примерами команд (http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html).

- Опции компилятора GCC для предупреждений (https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html)

- *Programming Principles and Practices Using C++:*

  глава 5 — обработка ошибок;
  глава 26 — тестирование.
  пункт 9.4.1 — структуры, раздел 9.5 — перечисления;

- *C++ Primer:*
  - глава 2, раздел 2.3 — указатели и ссылки;
  - глава 6 — функции;
  - раздел 19.3 — перечисления;