# Физическая декомпозиция и контроль корректности программ

Курс «Разработка ПО систем управления»

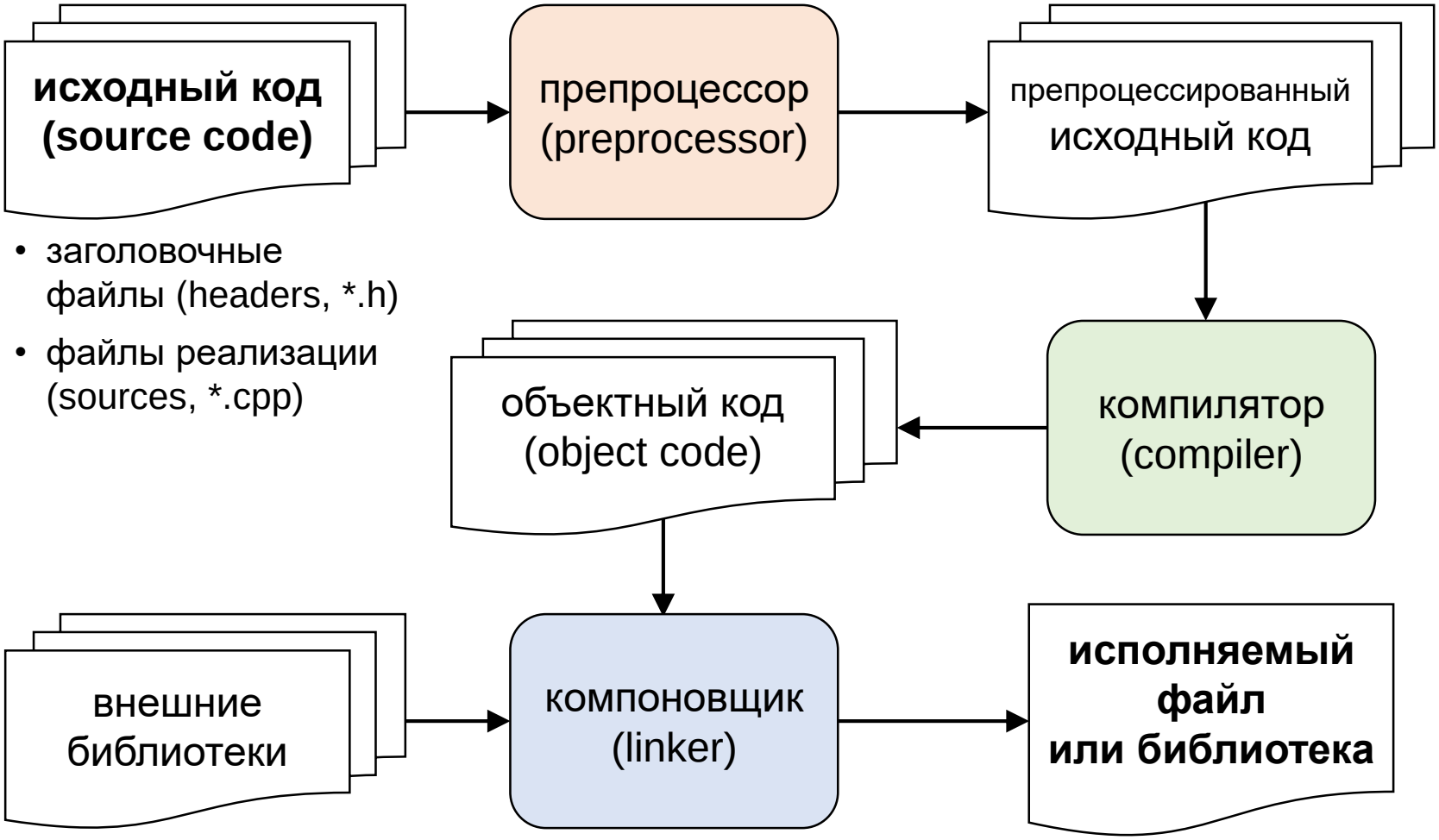
Кафедра управления и информатики НИУ «МЭИ»

Весна 2017 г.

# Декомпозиция

- **Физическая —** разделение кода по файлам.
  - Упрощение редактирования, навигации, контроля версий.
  - Ускорение сборки: пересобирать только измененные файлы.
- **Процедурная —** выделение в коде функций.
  - Упрощение восприятия кода.
  - Повторное использование.
  - Защита от ошибок: в компактной функции труднее запутаться.
- **Модульная —** выделение в программе подсистем и их интерфейсов.
  - Управление сложностью: не важно, как реализовано, — важно, как с этим работать обращаться.
  - Тестирование части программы в изоляции от других.

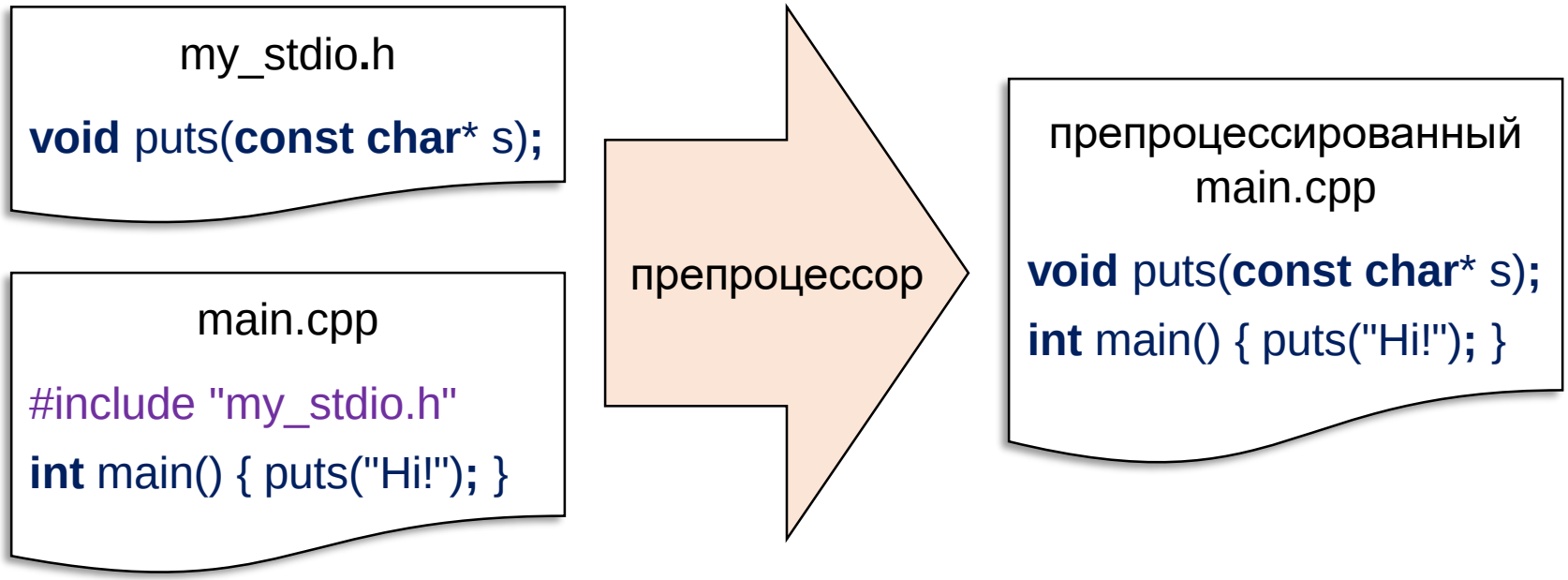
# Сборка программы (build)




- заголовочные файлы (headers, *.h)
- файлы реализации (sources, *.cpp)

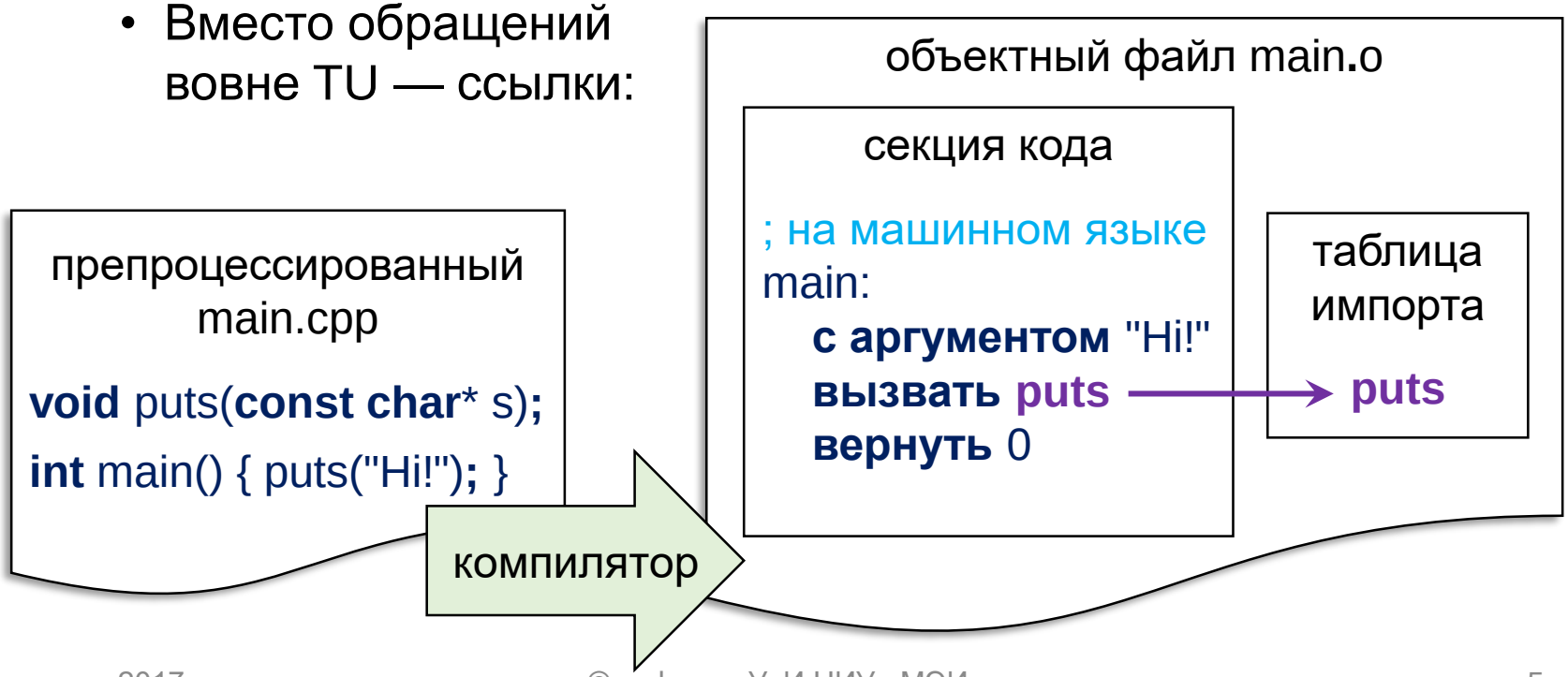
# Препроцессор

- Меняет код программы до компиляции как текст.
- Директивы препроцессора начинаются с #
- Пример: #include — подставляет текст из файла:



# Компилятор

- Обрабатывает файлы по отдельности.
  - Файл — единица трансляции (translation unit).
- Выдает объектный код (object code).
  - Очень близок к машинному коду.
  - Вместо обращений вовне TU — ссылки:



# Компоновщик (линкер)

- Собирает весь имеющийся код в исполняемый, разрешая ссылки в таблицах импорта объектных файлов.

# Сборка вручную

## Компиляция (и препроцессирование):

- Препроцессировать отдельным шагом можно.

- g++ –c –std=c++14 main.cpp –o main.o
- g++ –c –std=c++14 input.cpp –o input.o

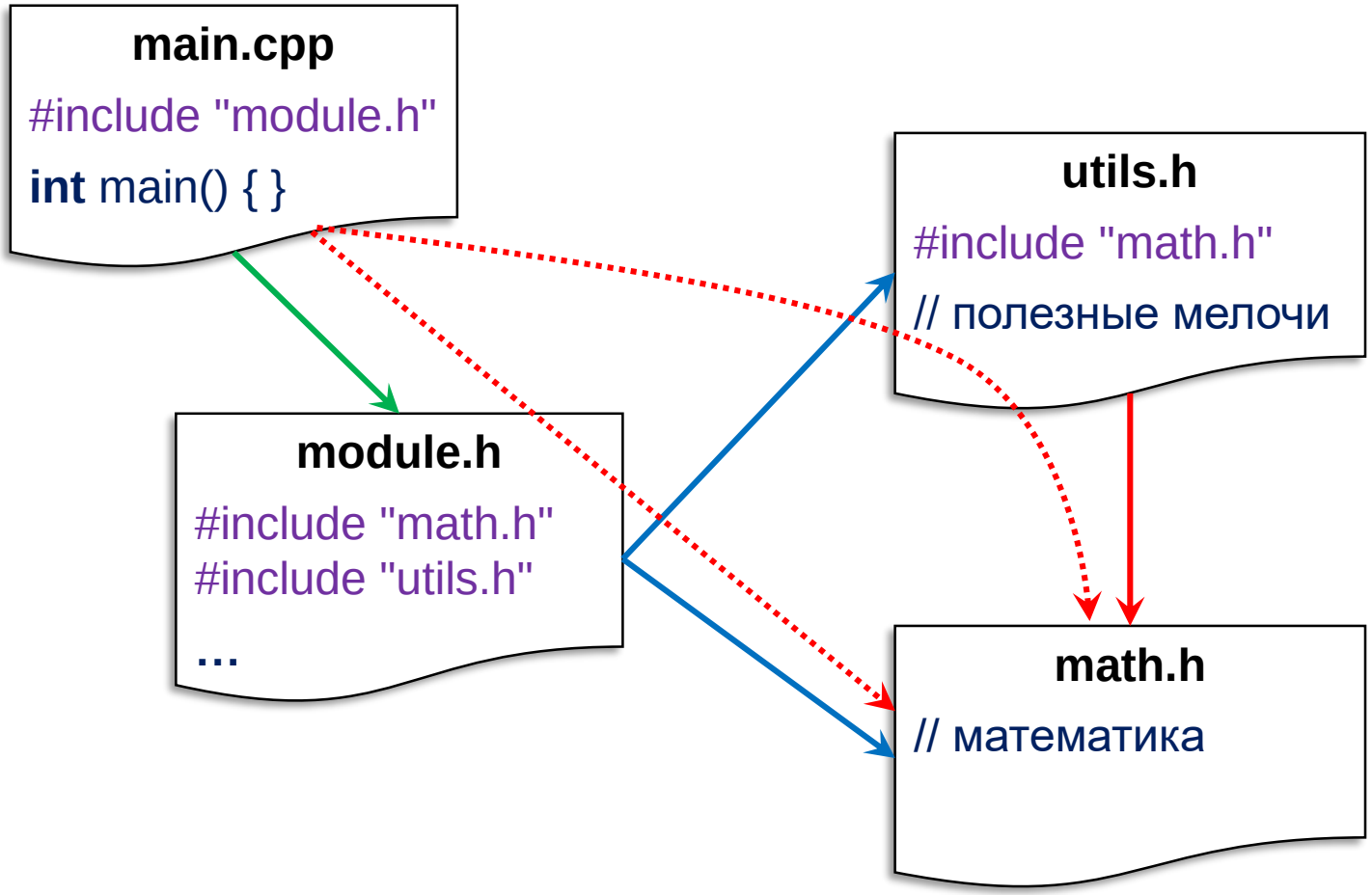
| g++ –c | –std=c++14 | input.cpp | –o input.o |
|---|---|---|---|
| вызов компилятора | флаги компиляции | вход: исходный код | выход: объектный код |

## Компоновка:

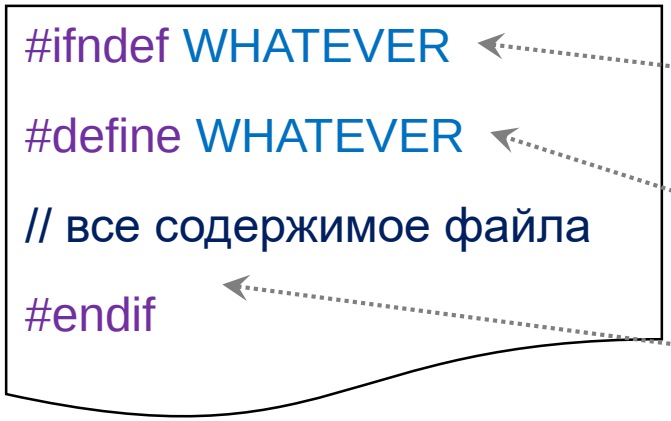
- g++ –static main.o input.o –o program.exe

| g++ | –static | main.o input.o | –o program.exe |
|---|---|---|---|
| вызов компоновщика | флаги компоновки | вход: объектный код | выход: исполняемый код |

# Проблема: повторное включение



# Решение: страж включения




Если препроцессору неизвестен символ WHATEVER…

…определить символ WHATEVER…

…и включить в текст программы все до #endif.

- **Гарантированно стандартный способ.**
- WHATEVER должно быть уникально (для единицы трансляции).

**«Нестандартный» способ:**

- Поддерживается всеми распространенными компиляторами.

```
#pragma once
// все содержимое файла
```

# 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 запросить дополнительный анализ:

     –Wall –Wextra –pedantic-errors  # большая часть предупреждений
  3. Можно трактовать как ошибки (-Werror)

# Обработка ошибок: код возврата

```
int convert_temperature(
    double temperature,
    char from, char to,
    double& result)
{
    if (from != 'K' && from != 'C')
        return 1;
    //…
}
```

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

Лишняя переменная — повод ошибиться.

(Аналог cout для сообщений об ошибках.)

```
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";
}
```

# 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.

# Защитное программирование

- Defensive programming:
  паранойя как подход к работе.
  - Проверять все входные параметры (контракт).
  - 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.

# assert() из <cassert>

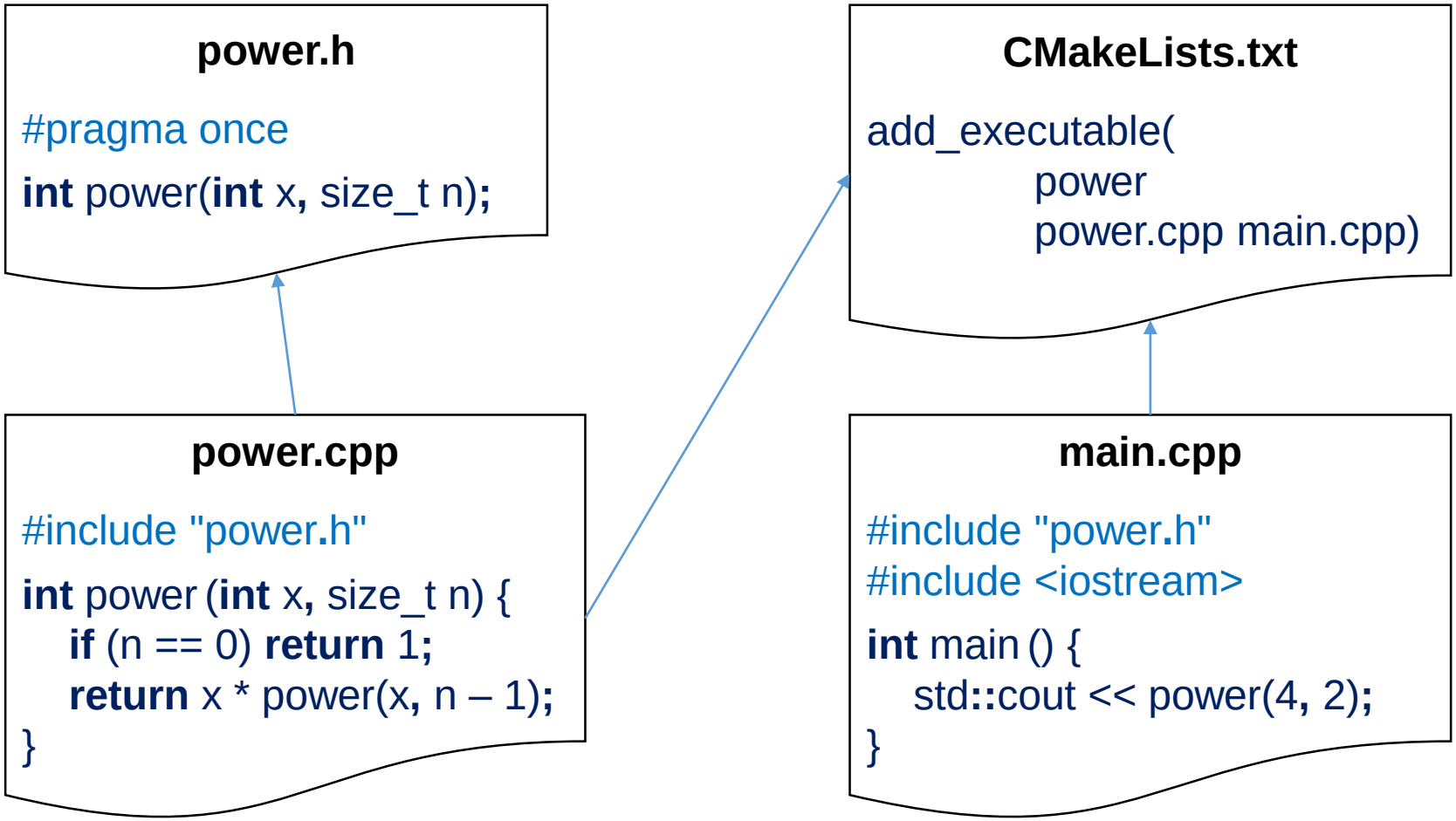
- Проверяет условие-аргумент.
- Если не выполняется:
  - печатает сообщение с этим условием;
  - завершает программу аварийно.
- Вне отладочной сборки ничего не делает.
  - CMAKE_BUILD_TYPE=Debug (по умолчанию в CLion).
  - Не влияет на конечную программу.

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

# 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.

# Unit testing: пример



# 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 );
}
```

# Тест в сборке

**CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.0)
set(CMAKE_EXE_LINKER_FLAGS –static)

add_executable(power power.cpp main.cpp)

# Тест использует модули исходной программы.

# Тест является отдельной программой.
    # Как и обычная программа, это исполняемый файл:
        add_executable(test_power power.cpp test_power.cpp)

    # Можно пометить программу как тест,
      чтобы позже запускать тесты пачками:
        enable_testing()
        add_test(NAME test_power COMMAND test_power)
```

# Запуск тестов через ctest

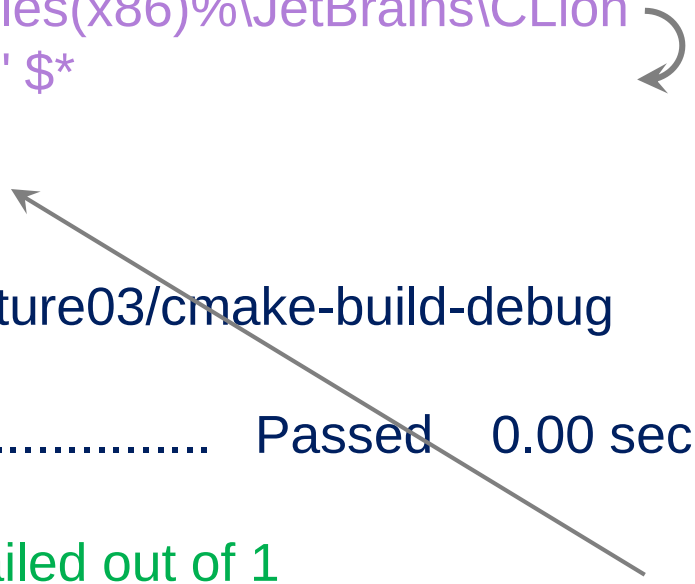
```
$ doskey ctest="%ProgramFiles(x86)%\JetBrains\CLion
   2017.1\bin\cmake\bin\ctest" $*

$ ctest  cmake-build-debug

Test project C:/cs-17-labs/lecture03/cmake-build-debug
    Start 1: test_power
1/1 Test #1: test_power .....................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.02 sec
```



Каталог
с программами-
тестами (*.exe).

# 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);
}
```

# 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
```

# 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.

# 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.