Структурирование и декомпозиция программы

Курс «Разработка ПО систем управления»

Кафедра управления и интеллектуальных технологий НИУ «МЭИ»

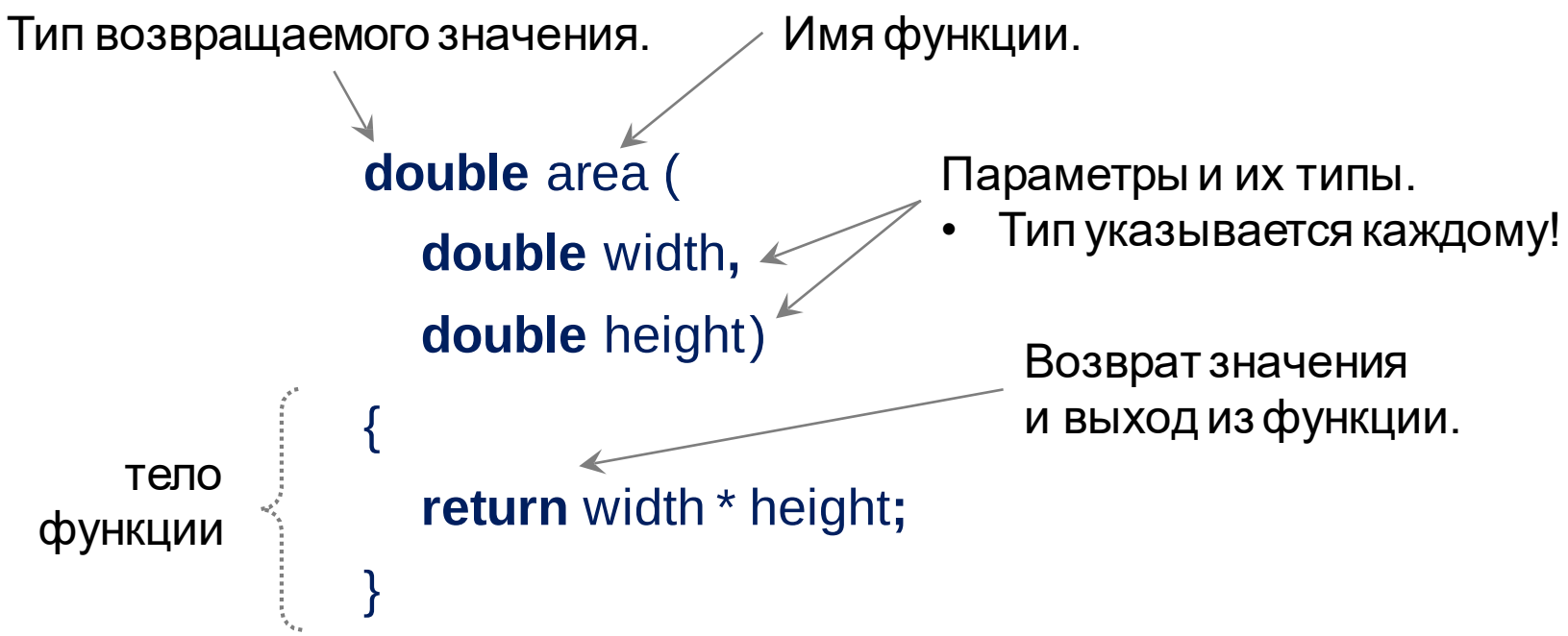
Весенний семестр 2021 г.

Декомпозиция

- **Процедурная —** выделение в коде функций.
  - Упрощение восприятия кода.
  - Повторное использование.
  - Защита от ошибок: в компактной функции труднее запутаться.
- **Физическая —** разделение кода по файлам.
  - Упрощение редактирования, навигации, контроля версий.
  - Ускорение сборки: пересобирать только измененные файлы.

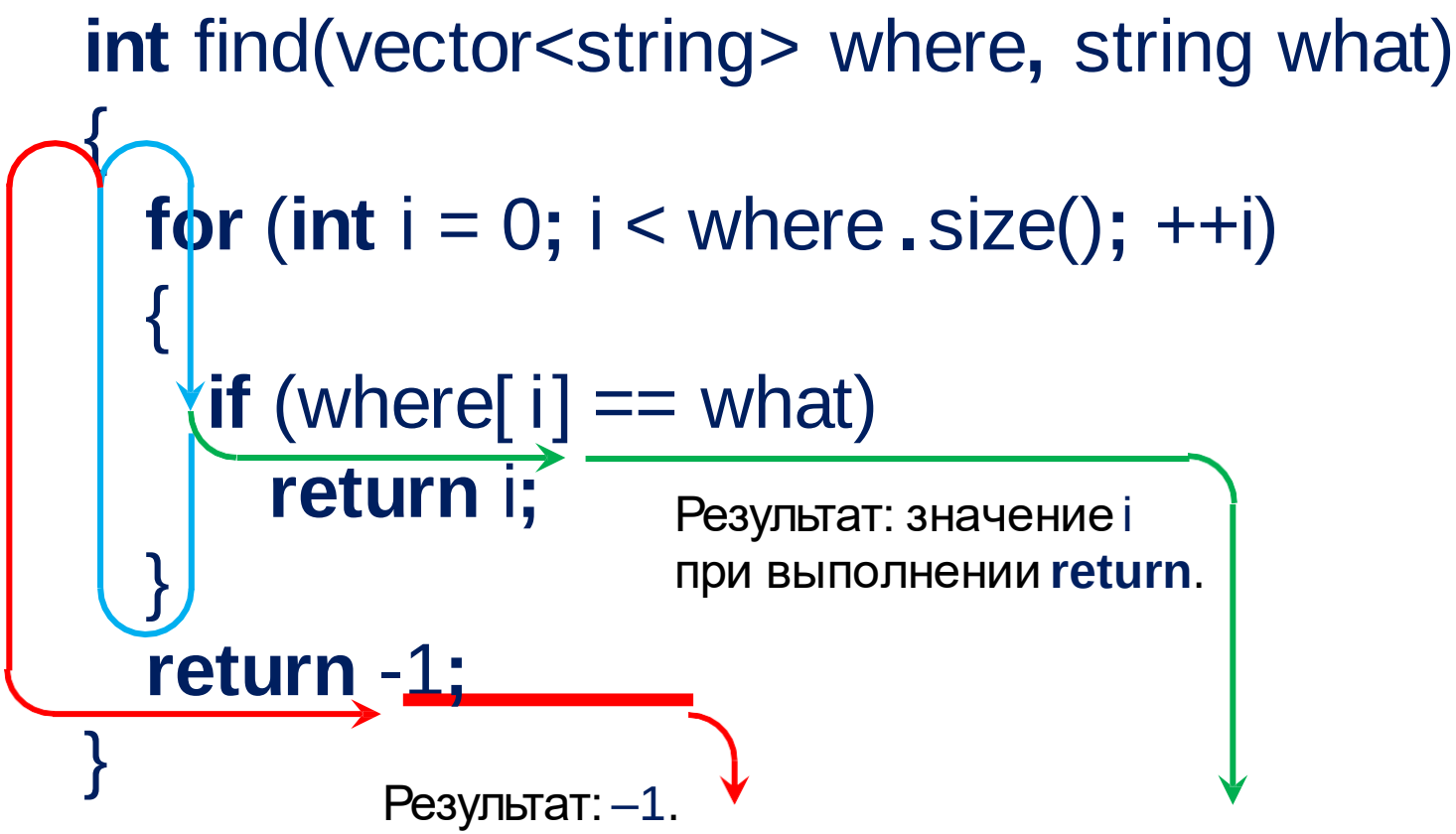
- **Модульная —** выделение в программе подсистем и их интерфейсов.
  - Управление сложностью: не важно, как реализовано, — важно, как с этим работать обращаться.
  - Тестирование части программы в изоляции от других.

Определение функции




```
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).

& — амперсанд

```
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);
}

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);
```

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;~~

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).

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) $S = \Sigma_{i=0}^{N-1}(x_i - m_x)(y_i - m_y)$;
5) $cov(x, y) = {}^{S}/_{N-1}$;
6) $r_{xy} = \frac{cov(x,y)}{s_x s_y}$.

Цикл?

```
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) $S = \Sigma_{i=0}^{N-1}(x_i - m_x)(y_i - m_y)$;
5) $cov(x, y) = {}^{S}/_{N-1}$;
6) $r_{xy} = \frac{cov(x,y)}{s_x s_y}$.

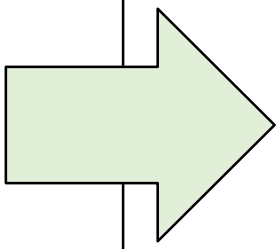
Физическая декомпозиция

- **Физическая декомпозиция —** разделение кода по файлам.
  - Упрощение редактирования, навигации, контроля версий.
  - Ускорение сборки: пересобирать только измененные файлы

Заголовочные файлы и файлы реализации

main.cpp

```
void func1 () {
…
}

void func2 () {
…
}


int main() {
func1 ();
func2 ();
}
```



funcs.h

```
#ifndef__FUNCS_H__
#define__FUNCS_H__
void func1 ();
void func2 ();
#endif //__FUNCS_H__
```

funcs.cpp

```
#include “funcs.h”
void func1 () {
…
}

void func2 () {
…
}
```

main.cpp

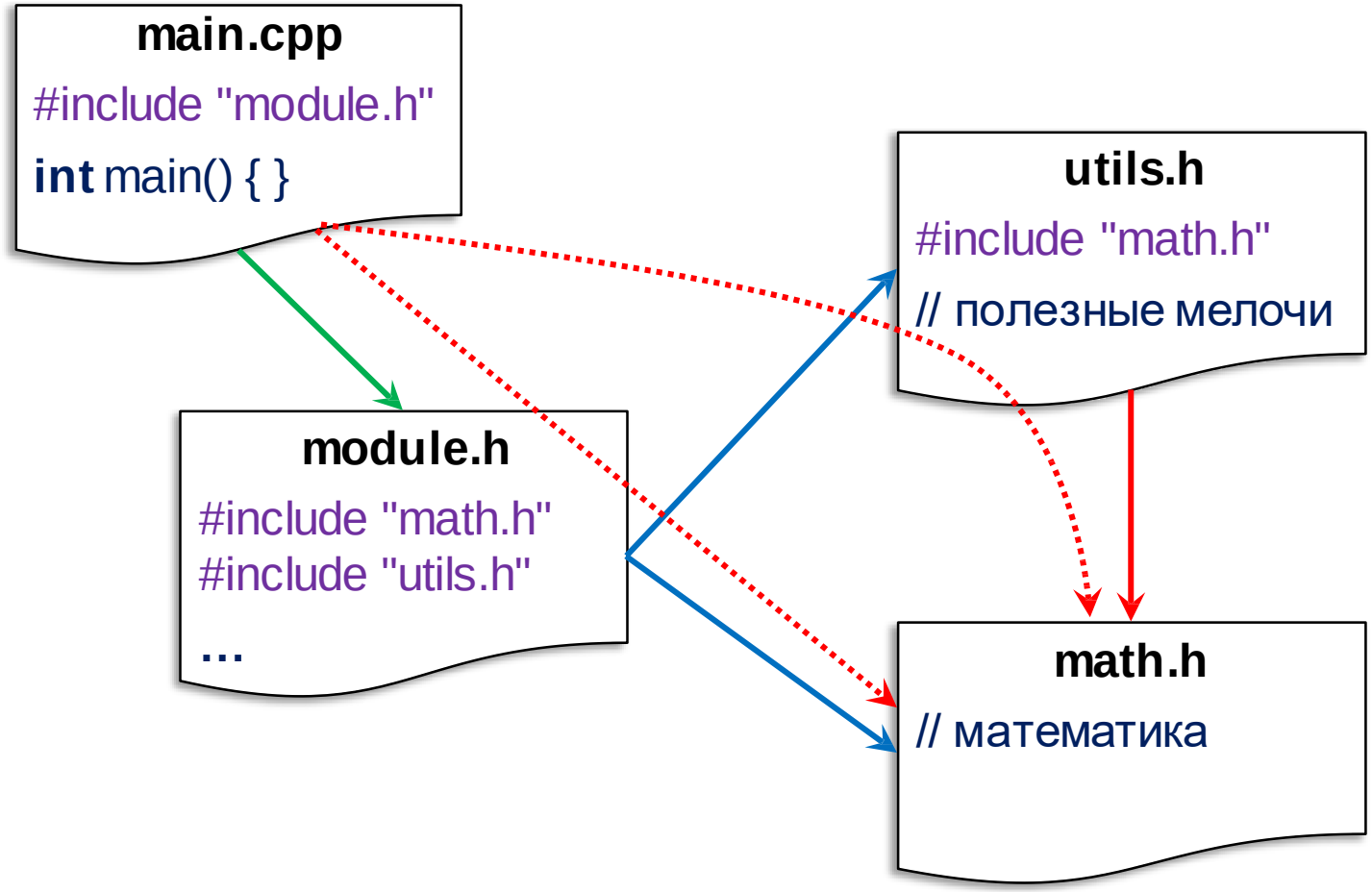
```
#include “funcs.h”

int main() {
func1 ();
func2 ();
}
```

Заголовочные файлы и файлы реализации

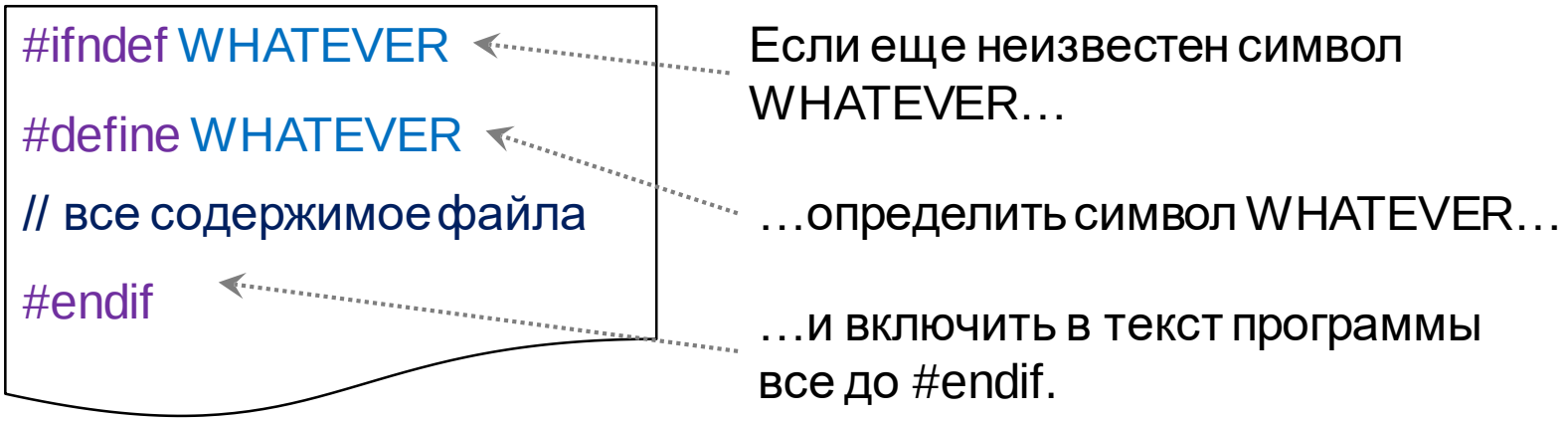
- **Заголовочные файлы (headers, *.h)** – содержат объявления функций, структуры, типы данных и т.д., используемые в другом модуле. В общем случае могут содержать любые конструкции языка программирования, но на практике исполняемый код в заголовочные файлы не помещают.

- **Файлы реализации (файлы исходников, sources, *.cpp)** – могут содержать как определения, так и объявления функций. Объявления, сделанные в файле реализации, будут действовать только для этой единицы компиляции. Должны содержать директиву включения соответствующего заголовочного файла.

Проблема: повторное включение



Решение: страж включения



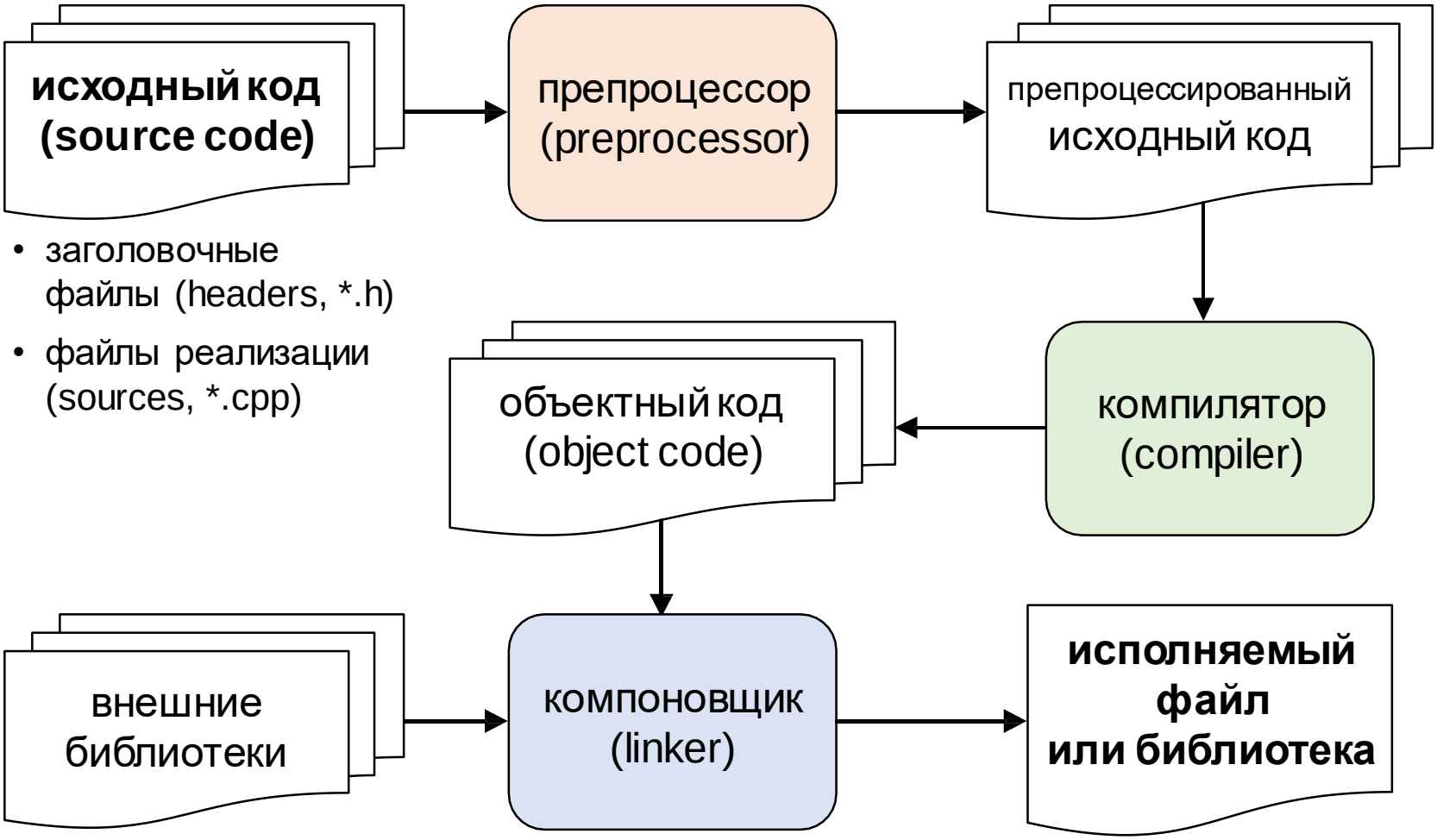
- **Гарантированно стандартный способ.**
- WHATEVER должно быть уникально (для единицы трансляции).

«Нестандартный» способ:

- Поддерживается всеми распространенными компиляторами.

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

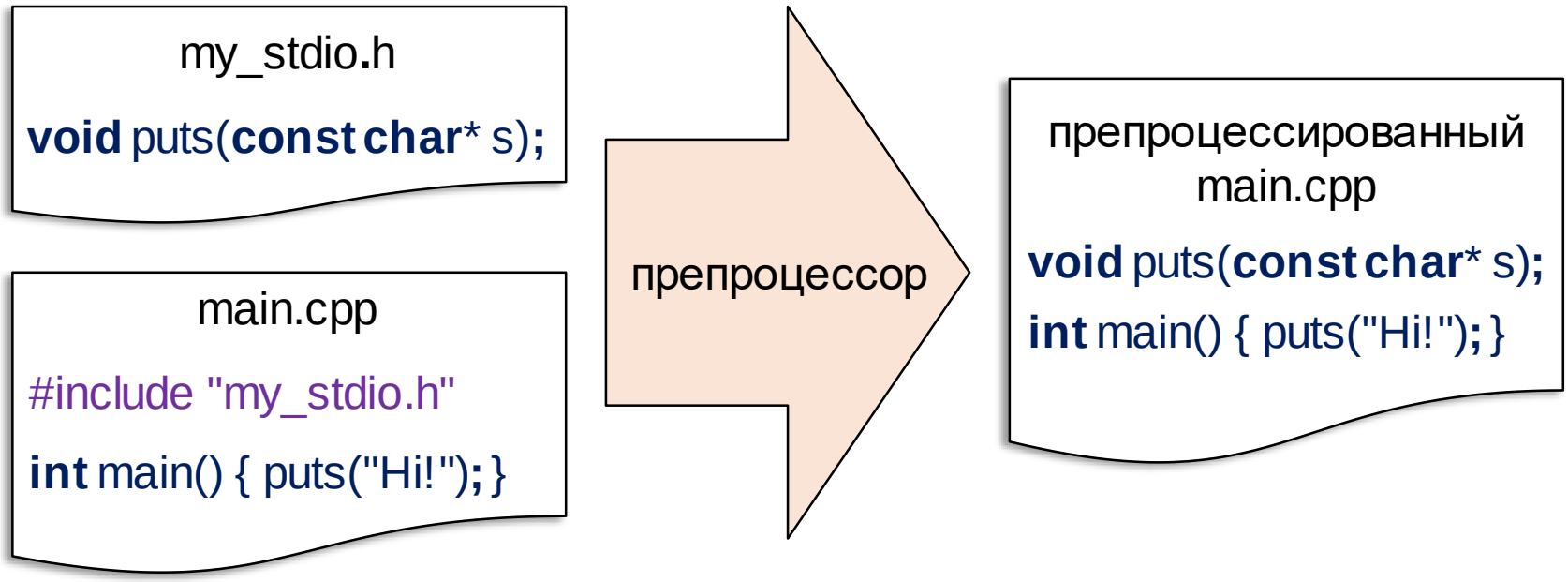
Сборка программы (build)



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

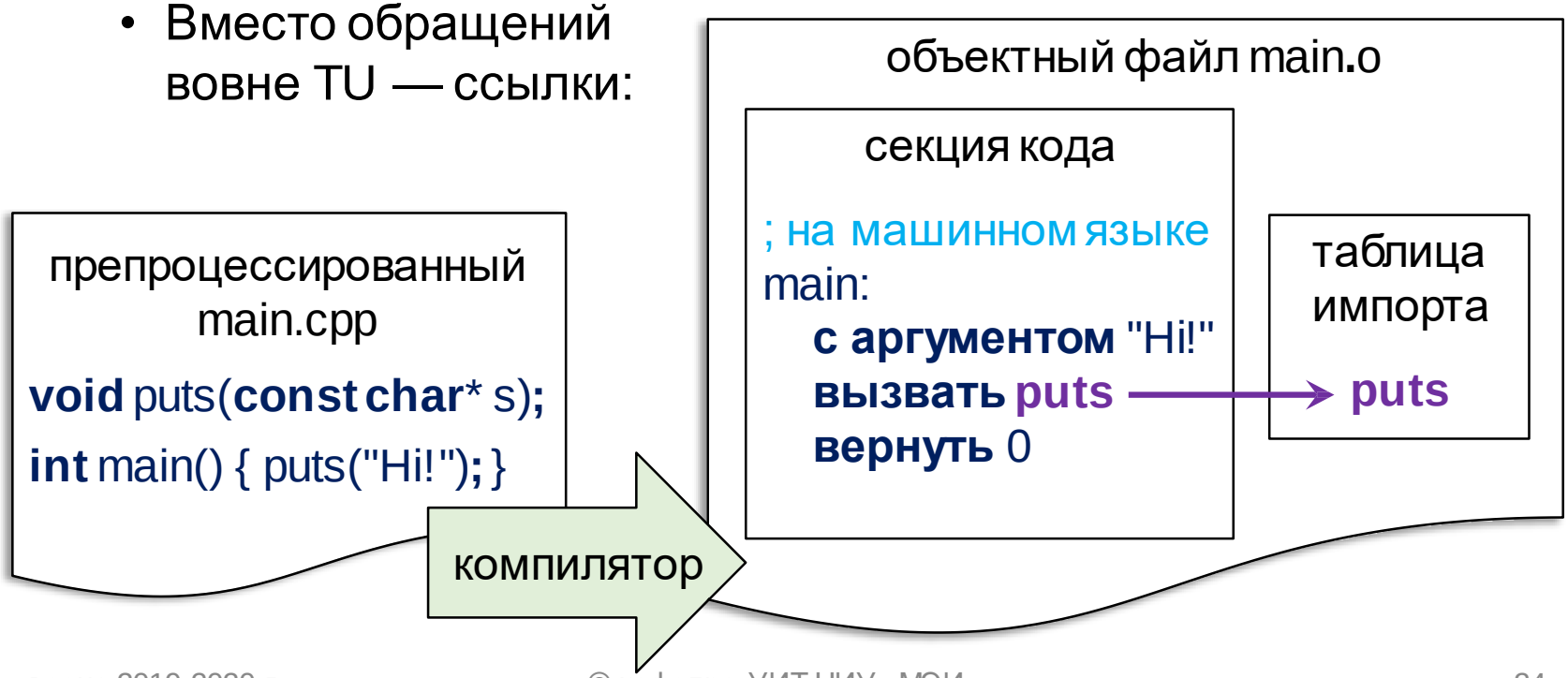
Препроцессор

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



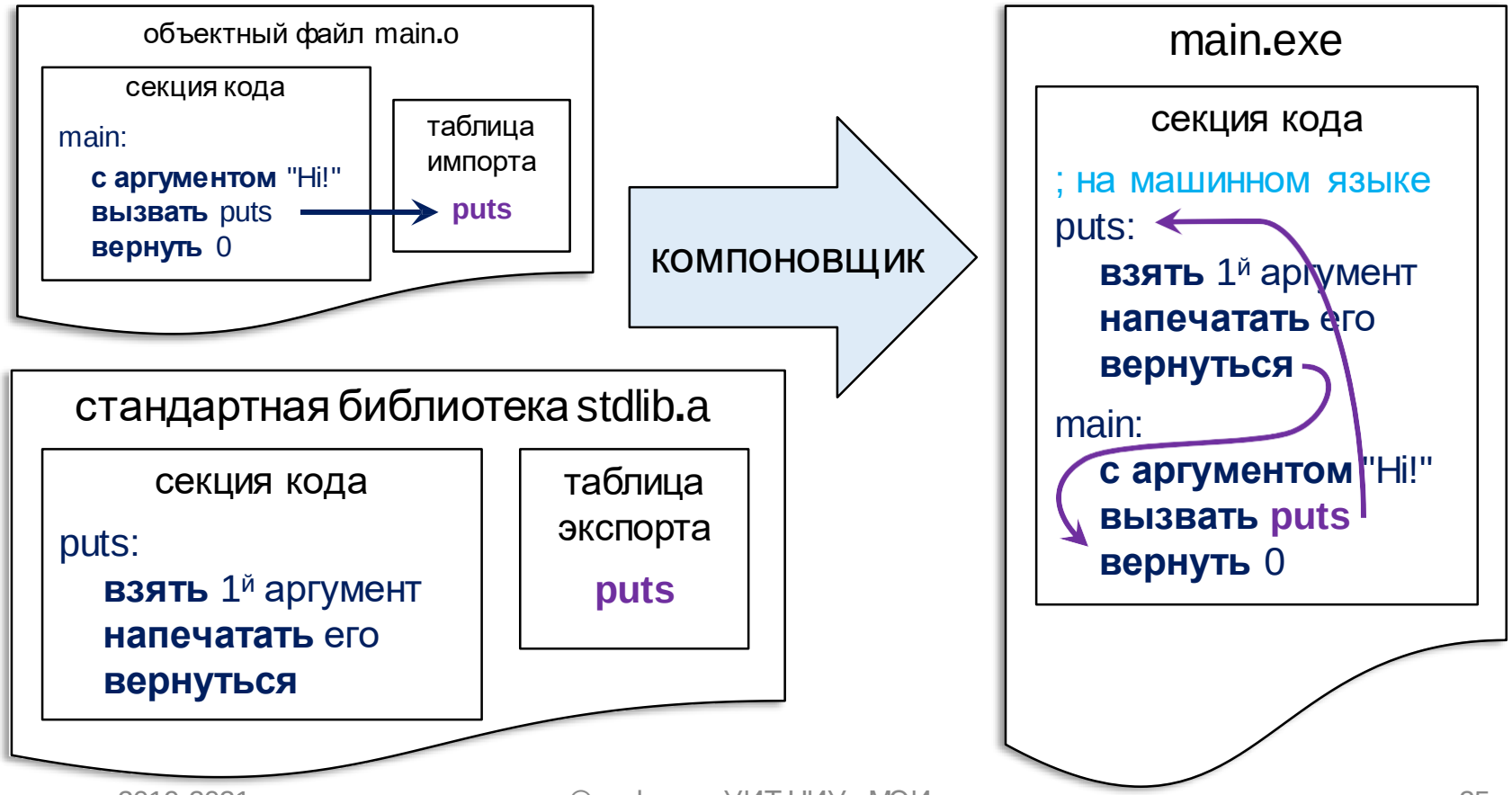
Компилятор

- 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):

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

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

Компоновка:

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

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