




 **ТЕХНОСФЕРА**

# **Препроцессор, компилятор, компоновщик**

Антон Кухтичев





---

## О преподавателях

#02



**Анастасия Семёнова**

Программист,  
Будет принимать  
домашние задания.



**Антон Кухтичев**

Ведущий программист,  
Не будет принимать  
домашние задания.  
Будет вести только  
лекции.



# Состав курса

- Препроцессор, компилятор, компоновщик
- Память в C++
- Функции
- Классы и методы классов
- Сору и move-семантика
- Шаблоны
- Исключения
- STL
- Многопоточность в двух частях

Лекции и примеры будут тут: [https://github.com/mailcourses/technosphere\\_deep\\_cpp](https://github.com/mailcourses/technosphere_deep_cpp)

## О домашних заданиях (1)

- В вашем GitHub должен быть репозиторий `msu_cpp_autumn_2020`;
- Внутри репозитория должны быть директории из двух цифр, вида: 01, 02 и т. д. — это номера домашних заданий;
- Внутри каждой директории могут быть любые файлы реализующие задачу. Обязательным является только файл `Makefile`;
- В `Makefile` обязательно должны быть цель `test`, которая запускает тесты вашего решения;
- Собираться ваш код должен компилятором поддерживающим стандарт C++17;

## О домашних заданиях (2)

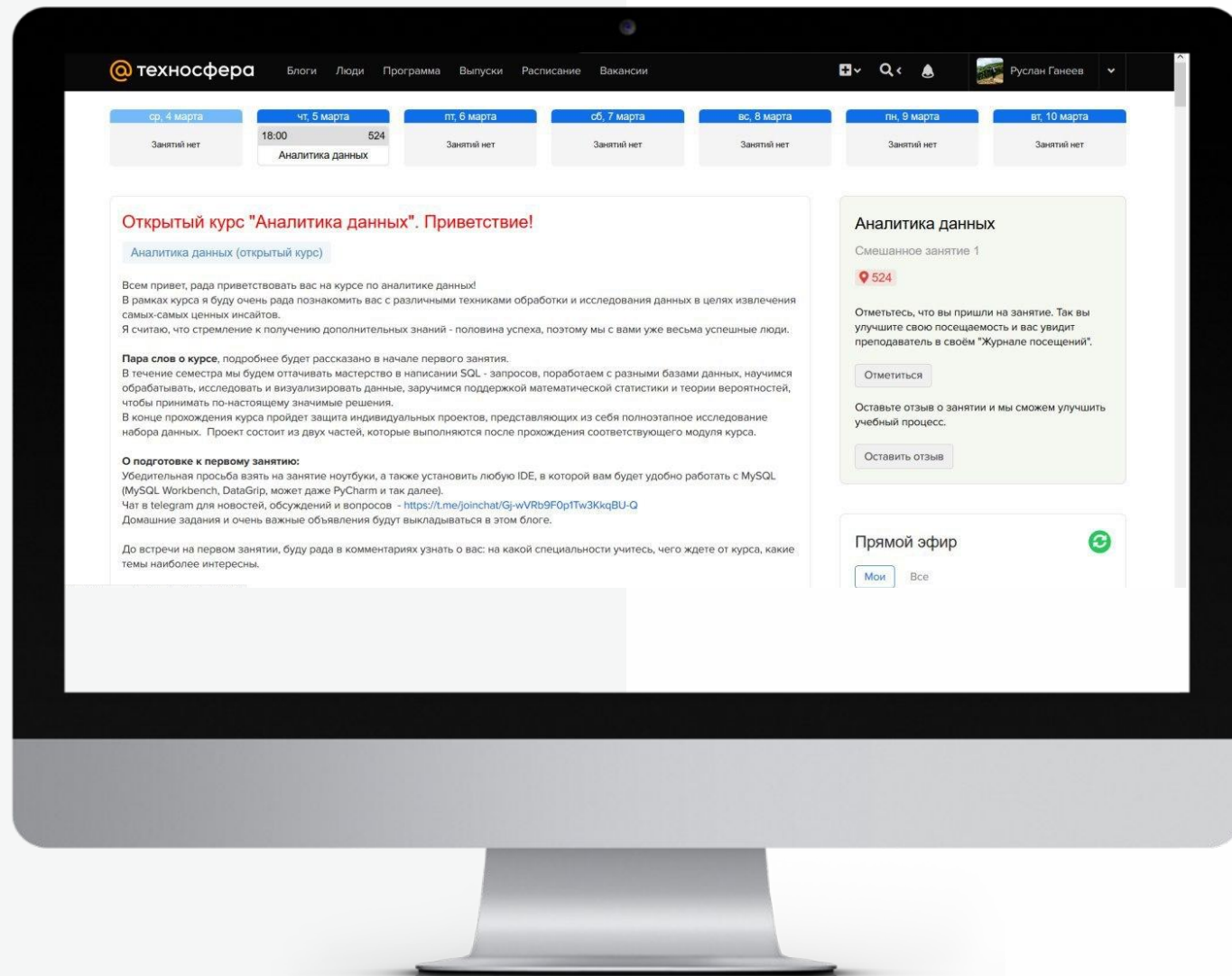
- Внешних зависимостей быть не должно;
- Код решения должен быть отформатирован, так проще его читать. Не забывайте про отступы;
- О том, что вы выполнили работу надо сообщать Семеновой Анастасии, к комментарию необходимо добавить вашу ссылку на GitHub;
- Максимальное количество попыток сдачи одного задания - 3.

Для допуска к экзамену должны быть выполнены **BCE** задания!



# Содержание занятия

1. Этапы компиляции
2. Препроцессор
3. Макросы препроцессора
4. Объектный файл
5. Компиляция
6. компоновка
7. Оптимизация
8. Статические библиотеки
9. Утилита для автоматизации



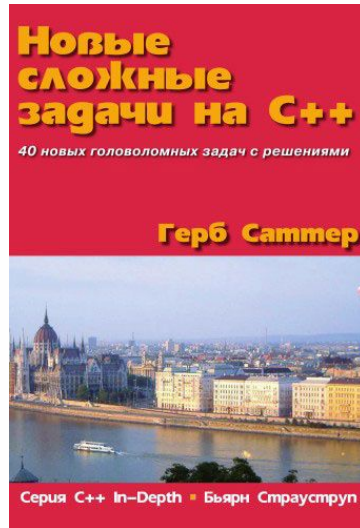
Не забудьте  
отметиться на  
портале!!!

Иначе всё плохо будет.

# Рекомендуемая литература



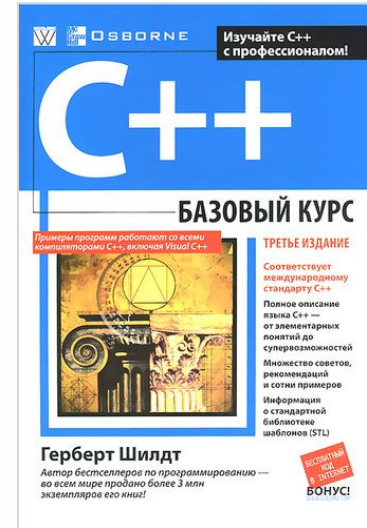
Брюс Эккель



Герб Саттер



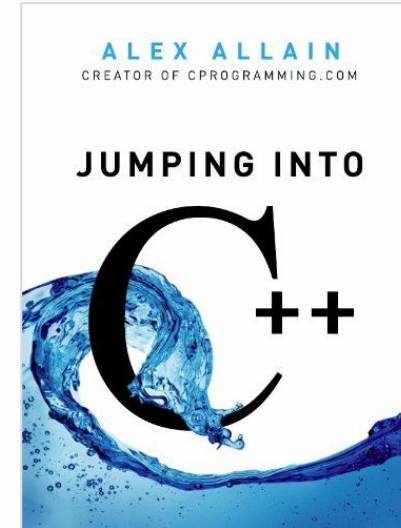
Скотт Мейерс



Герберт Шилдт



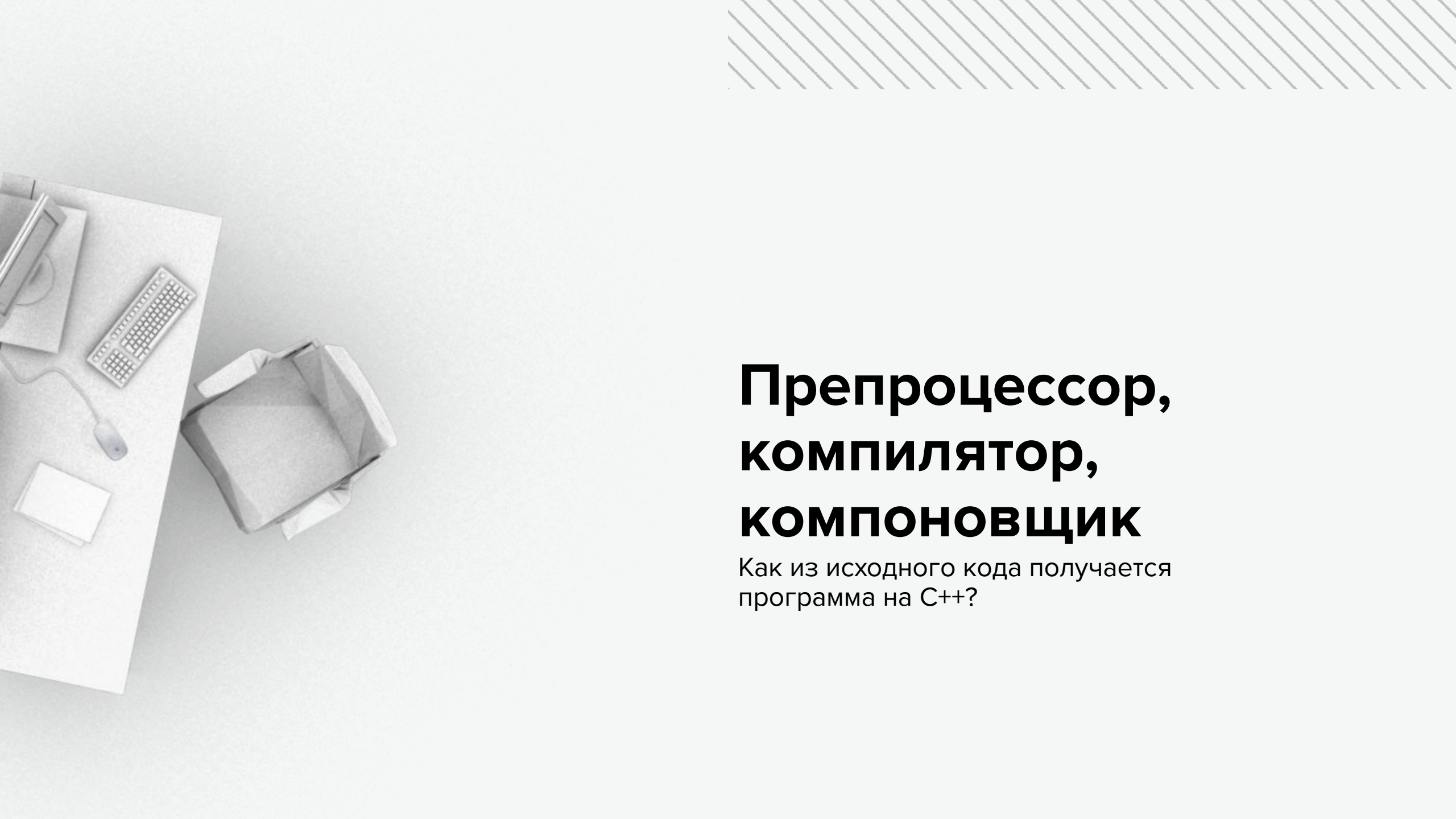
Бьярн Страуструп



Alex Allain

Практика: <https://leetcode.com>





# Препроцессор, компилятор, КОМПОНОВЩИК

Как из исходного кода получается  
программа на C++?



---

## С чего начинается программа?

С чего начинается Родина?  
С картинки в твоем букваре  
С хороших и верных товарищей  
Живущих в соседнем дворе

- Михаил Матусовский

```
// hello.cpp
```

```
#include <iostream>
```


```
int main(int argc, char **argv)
{
    std::cout << "Hello, world! " <<
    std::endl;
}
```

```
$ g++ -std=c++17 hello.cpp -o hello
```

```
$ ./hello
Hello, world!
```



# Этапы компиляции

- 
1. **Препроцессор.** Обработка исходного кода (preprocessing);
  2. **Компиляция.** Перевод подготовленного исходного кода в ассемблерный код (compiling);
  3. **Ассемблирование.** Перевод ассемблерного кода в объектных файл (assembly);
  4. **Компоновка.** Сборка одного или нескольких объектных файлов в один исполняемый файл (linking).

# Препроцессор (1)

- Делаются макроподстановки:
  - определения (#define, #undef);
  - условные включения (#ifdef, #ifndef, #if, #endif, #else and #elif);
  - директива #line;
  - директива #error;
  - директива #pragma;
- Подстановка предопределённых макросов (\_\_LINE\_\_, \_\_FILE\_\_, \_\_DATE\_\_, \_\_cplusplus и др.)
- Результат обработки препроцессором исходного файла называется единицей **трансляции**.

## Препроцессор (2)

- Выполняются директивы `#include`
  - `#include "name"` — целиком вставляет файл с именем "name", вставляемый файл также обрабатывается препроцессором. Поиск файла происходит в директории с файлом, из которого происходит включение;
  - `#include <name>` — аналогично предыдущей директиве, но поиск производится в глобальных директориях и директориях, указанных с помощью ключа `-I`

## Препроцессор (3). Макрос #define (\*)

- #define PI 3.141592

Если при использовании PI будет ошибка компиляции, то в сообщение от компилятора увидите значение 3.14.1592, а не PI!

- #define MAX(x, y) ( x > y ? x : y )

```
int a = 5;
```

```
std::cout << MAX(++a, 0) << std::endl; // а увеличится два  
раза!
```

```
std::cout << MAX(++a, 10) << std::endl; // а тут всего лишь  
один раз!
```

- #define MULT(x, y) x \* y

```
std::cout << MULT(1+2, 3+4) << std::endl; // 1+2*3+4
```

# Препроцессор. Примерчик

```
// example.cpp
#include <iostream>
#define NAME(world) #world
int main(int argc, char **argv)
{
    #line 100
    std::cout << "Hello, " << NAME(world) << " from " <<
    __FILE__ << " and line #" << __LINE__ << std::endl;
}
```

```
g++ -E example.cpp -o example.ii
```

# Компиляция/Ассемблирование

Файлы .cpp/.c - один файл с исходным кодом - один объектный файл. Это называется единица трансляции.

```
# Компиляция и ассемблирование: создать объектный файла
# example.o
g++ -c example.cpp
# или
# Только компиляция: создать ассемблерный код example.s ...
g++ -S example.cpp
# ... а затем ассемблирование: создание объектного файла.
as example.s -o example.o
```



# Объектный файл (1)

- Определяется форматом
  - ELF (Executable and Linkable Format) на Unix-подобных системах;
  - Mach-O (Mach object) на семействе MacOS;
  - Узнать можно командой

```
file <объектный файл>.o
```

```
test.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

```
<объектный файл>.o: Mach-O 64-bit object x86_64
```

## Объектный файл (2)

- Существует три разновидности объектных файлов:
  - **Перемещаемый объектный файл (Relocatable object file)** — можно компоновать с другими объектными файлами для создания исполняемых или общих объектных файлов.
  - **Исполняемый объектный файл (Executable object file)** — можно запускать; в файле указано, как ехес (базовая операционная система) создаст образ процесса программы.
  - **Разделяемый объектный файл (Shared object file)** — загружаются в память во время исполнения.

## Объектный файл (3)

- Состоит из секций
  - Машинный код (.text)
  - Инициализированные данные, с правами на чтение и запись (.data)
  - Инициализированные данные, с правами только на чтение (.rodata)
  - Неинициализированные данные, с правами на чтение/запись (.bss)
  - Таблица символов (.symtab)
- Символы — то, что находится в объектном файле — кортежи из
  - Имя — произвольная строка;
  - Адрес — число (смещение, адрес);
  - Свойства: тип связывания (binding) (доступен ли символ вне файла);
- Декорирование (mangling)

---

# Декорирование (mangling)

Функция в исходном файле:

```
int square(int value);
```

Имя после декорирования:

```
_Z6squarei
```

Есть деманглер!

```
c++filt _Z6squarei
```

# Декорирование (mangling)

```
objdump -d square.o
```

```
square.o:      file format elf64-x86-64
```

```
Disassembly of section .text:
```

```
0000000000000000 <_Z6squarei>:
```

0:	55	push	%rbp
1:	48 89 e5	mov	%rsp,%rbp
4:	89 7d fc	mov	%edi,-0x4(%rbp)
7:	8b 45 fc	mov	-0x4(%rbp),%eax
a:	0f af 45 fc	imul	-0x4(%rbp),%eax
e:	5d	pop	%rbp
f:	c3	retq	

## Объектный файл (4)

- Глобальные символы
  - Символы определенные в одном модуле таким образом, что их можно использовать в других модулях;
  - Например: не-static функции и не-static глобальные переменные;
- Внешние (неопределенные) символы
  - Глобальные символы, которые используются в модуле, но определены в каком-то другом модуле.
- Локальные символы
  - Символы определены и используются исключительно в одном модуле.
  - Например: функции и переменные, определенные с модификатором static.
  - Локальные символы не являются локальными переменными программы



## Утилиты для изучения объектных файлов

- **nm** — выводит перечень символов объектного файла.
- **objdump** — выводит подробную информацию, содержащуюся в объектных файлах.
- **readelf** — выводит информацию об объектных файлах ELF.

## Объектный файл (4)

```
$ objdump -t square.o
```

```
square.o:          file format elf64-x86-64
```

```
SYMBOL TABLE:
```

```
0000000000000000 l      df *ABS* 0000000000000000 square.cpp
0000000000000000 l      d   .text 0000000000000000 .text
. . .
0000000000000000 l      d   .comment 0000000000000000 .comment
0000000000000000 g      F   .text 0000000000000010 _Z6squarei
```



## Объектный файл (5)

```
$ readelf -s main.o
```

Таблица символов «.symtab» содержит 11 элементов:

Чис:	Знач	Разм	Тип	Связ	Vis	Индекс имени
0:	000000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND
1:	000000000000000000	0	FILE	LOCAL	DEFAULT	ABS main.cpp
...						
8:	000000000000000000	19	FUNC	GLOBAL	DEFAULT	1 main
9:	000000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND i
10:	000000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND _Z6squarei

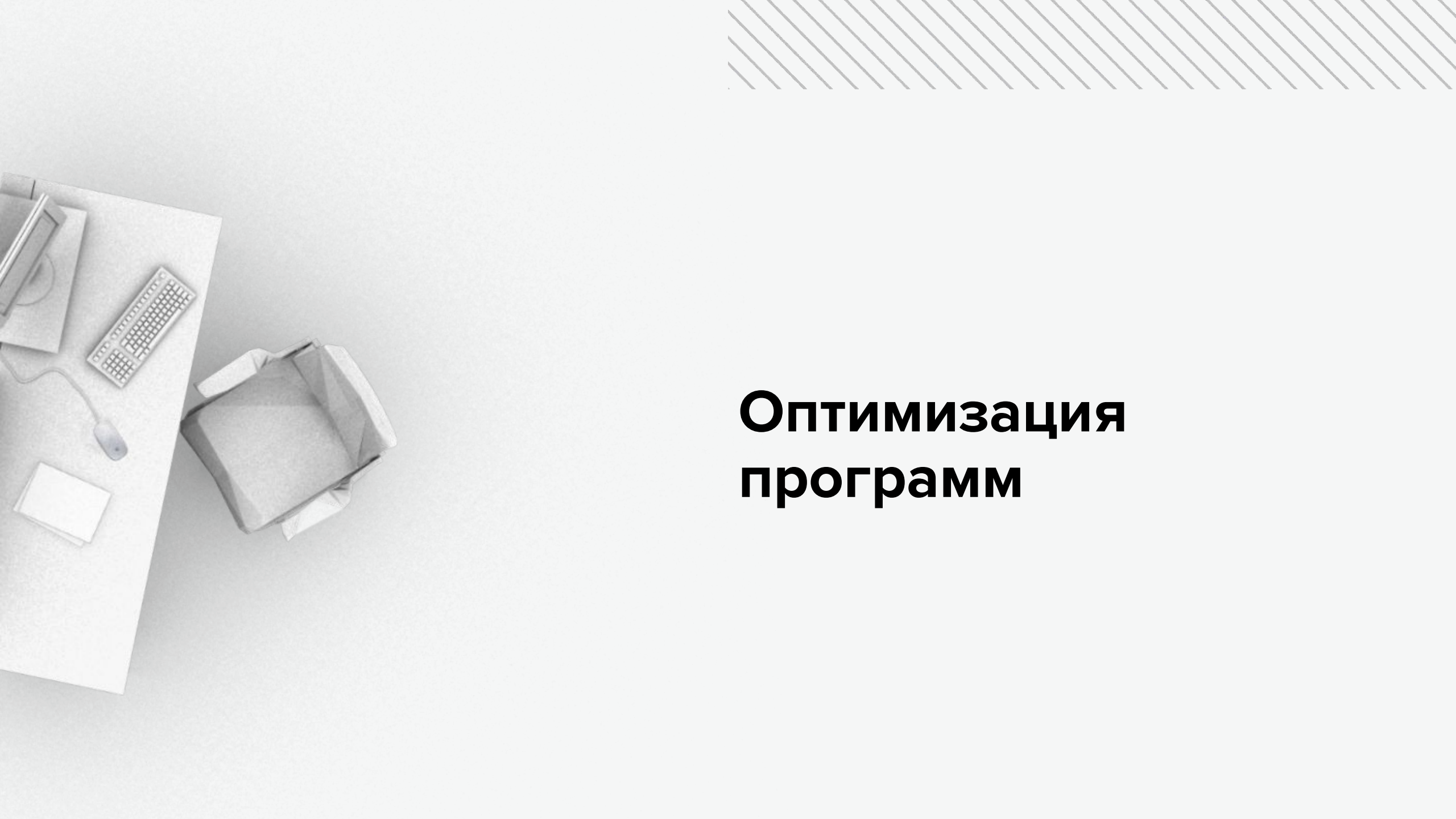
# Компоновка (1)

Компоновщик собирает из одного и более объектных файлов исполняемый файл.

```
$ g++ -o my_prog main.o square.o  
$ ./my_prog  
$ echo $?  
4
```

## Компоновка (2)

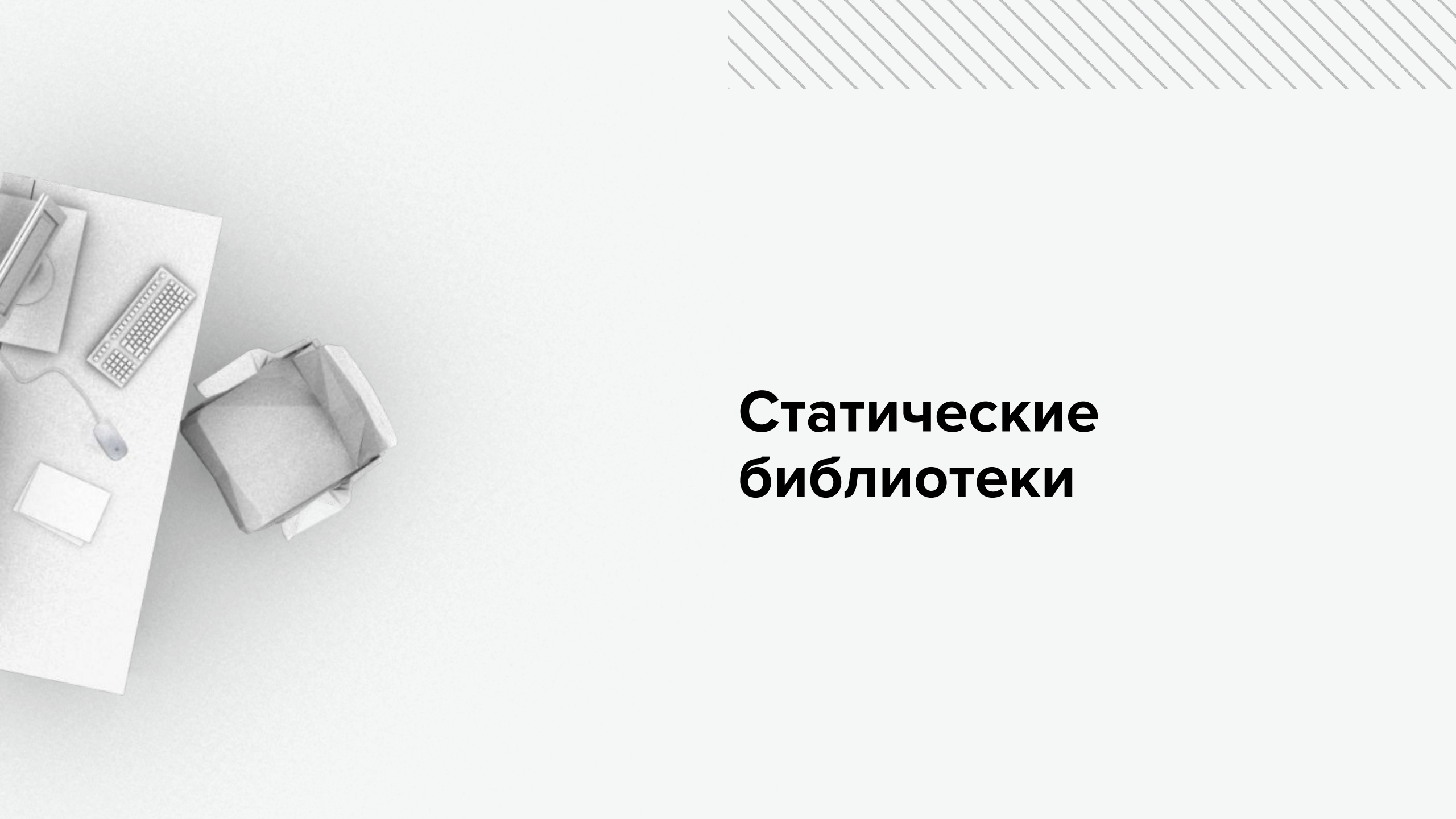
- Организация программы как набор файлов с исходным кодом, а не один монолитный файл.
- Организовывать библиотеки функций, являющихся общими для разных программ;
- Раздельная компиляция:
  - Меняем код в одном файле, компилируем только его, повторяем компоновку;
  - Нет необходимости повторять компиляцию остальных файлов с исходным кодом.
- Исполняемые файлы и образ программы в памяти содержит только те функции, которые действительно используются.



# Оптимизация программ

# Ключи оптимизации

- -O0 - отключение оптимизации (по умолчанию);
- -O1 - Пытается уменьшить размер кода и ускорить работу программы. Соответственно увеличивается время компиляции. При указании -O активируются следующие флаги: -fthread-jumps, -fdefer-pop.
- -O2 - GCC выполняет почти все поддерживаемые оптимизации, которые не включают уменьшение времени исполнения за счет увеличения длины кода.
- -O3 - Оптимизирует еще немного. Включает все оптимизации -O2 и также включает флаг -finline-functions и -fweb.



# Статические библиотеки

# Статические библиотеки

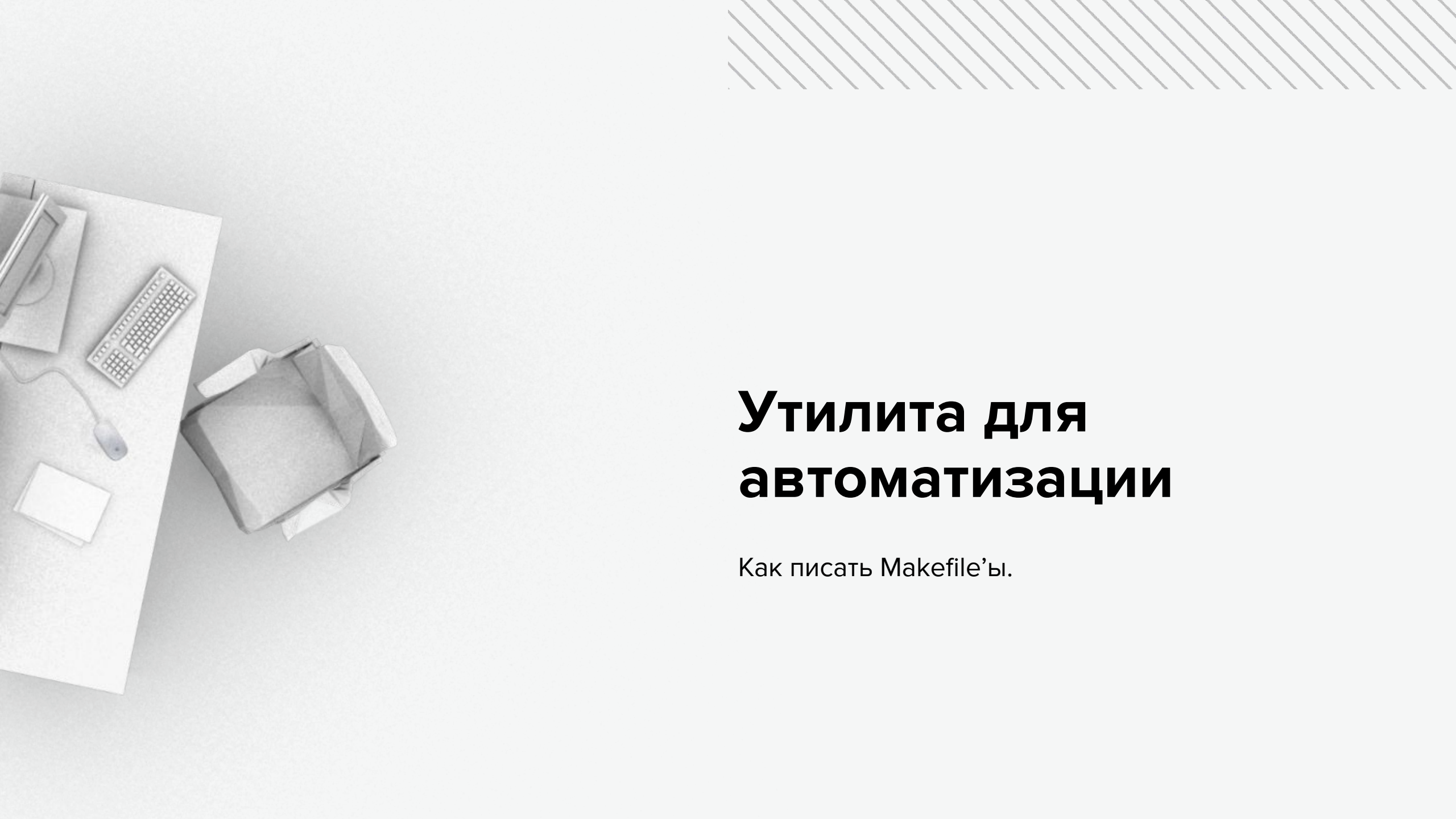
```
$ ar rc libsquare.a square.o  
libsquare.a
```

В Unix принято, что статические библиотеки имеют префикс lib и расширение .a.

```
g++ -o my_prog main.o -L. -lsquare
```

- -L - путь, в котором компоновщик будет искать библиотеки
- -l - имя библиотеки

Статические библиотеки нужны только при сборке.



# Утилита для автоматизации

Как писать Makefile'ы.



# Утилита make

Синтаксис:

цель: зависимости  
[tab] команда

Скрипт как правило находится в файле с именем **Makefile**.

Вызов:

make цель

Цель вызывается, если явно не указать цель:

make

# Плохой пример

CC=g++

FLAGS=-std=c++17 -Wall -Pedantic -Wextra -Wno-unused-variable

all: my\_prog

my\_prog: main.cpp square.cpp square.h

\$(CC) \$(FLAGS) -o my\_prog main.cpp square.cpp

clean:

rm -rf \*.o my\_prog

# Хороший пример

CC=g++

FLAGS=-std=c++17 -Wall -Pedantic -Wextra -Wno-unused-variable  
all: my\_prog

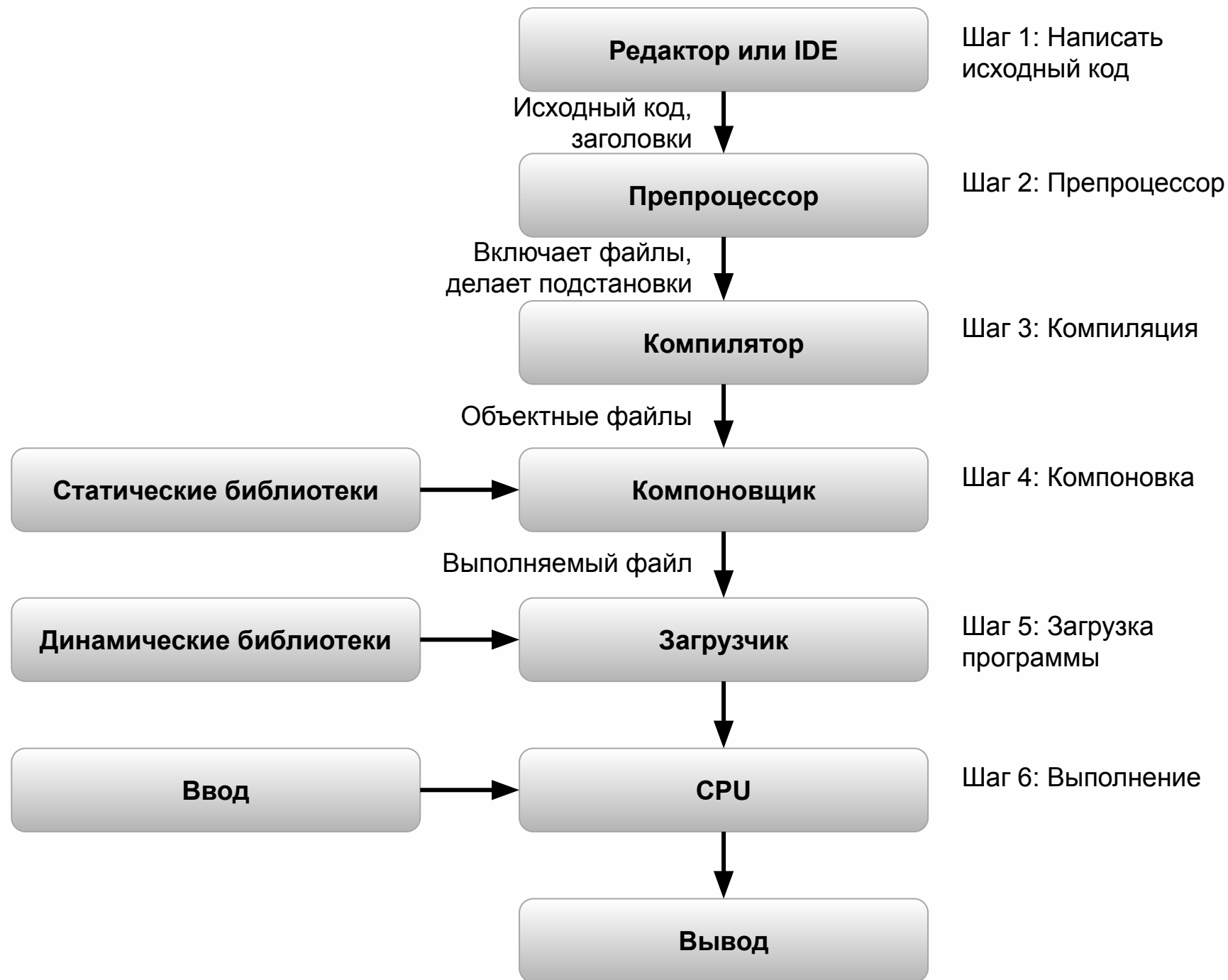
my\_prog: main.o square.o  
\$(CC) \$(FLAGS) -o my\_prog main.o square.o

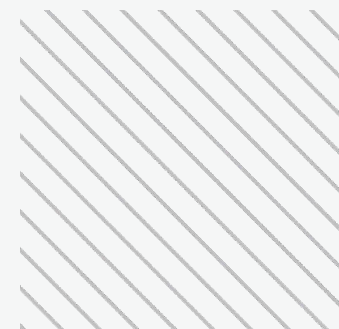
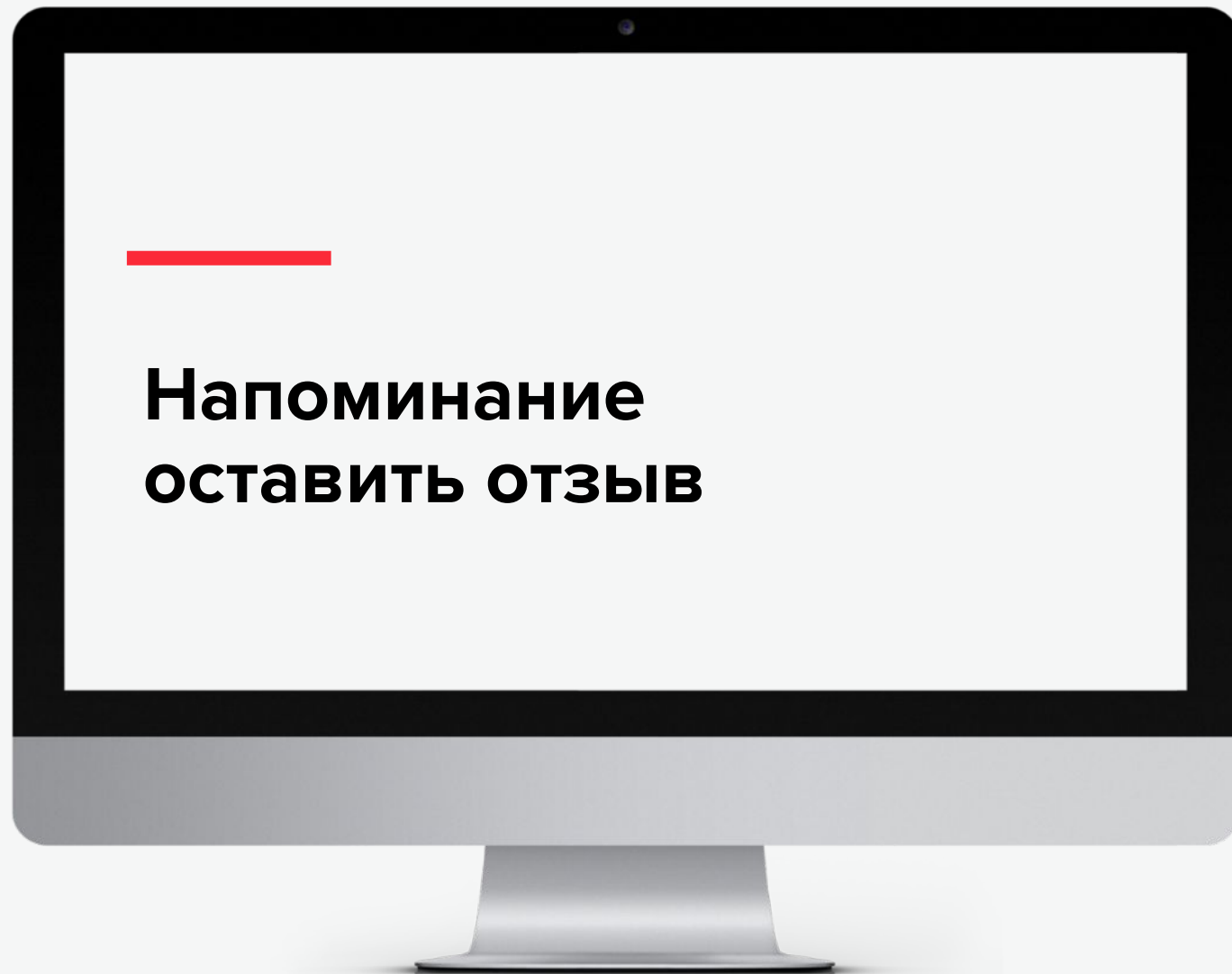
main.o: main.cpp square.h  
\$(CC) \$(FLAGS) -c main.cpp

square.o: square.cpp square.h  
\$(CC) \$(FLAGS) -c square.cpp

clean:  
rm -rf \*.o my\_prog

# Итог





**СПАСИБО  
ЗА ВНИМАНИЕ**

