

# Углублённое программирование на C++


## Этапы компиляции

Кухтичев Антон



20 марта 2025 года

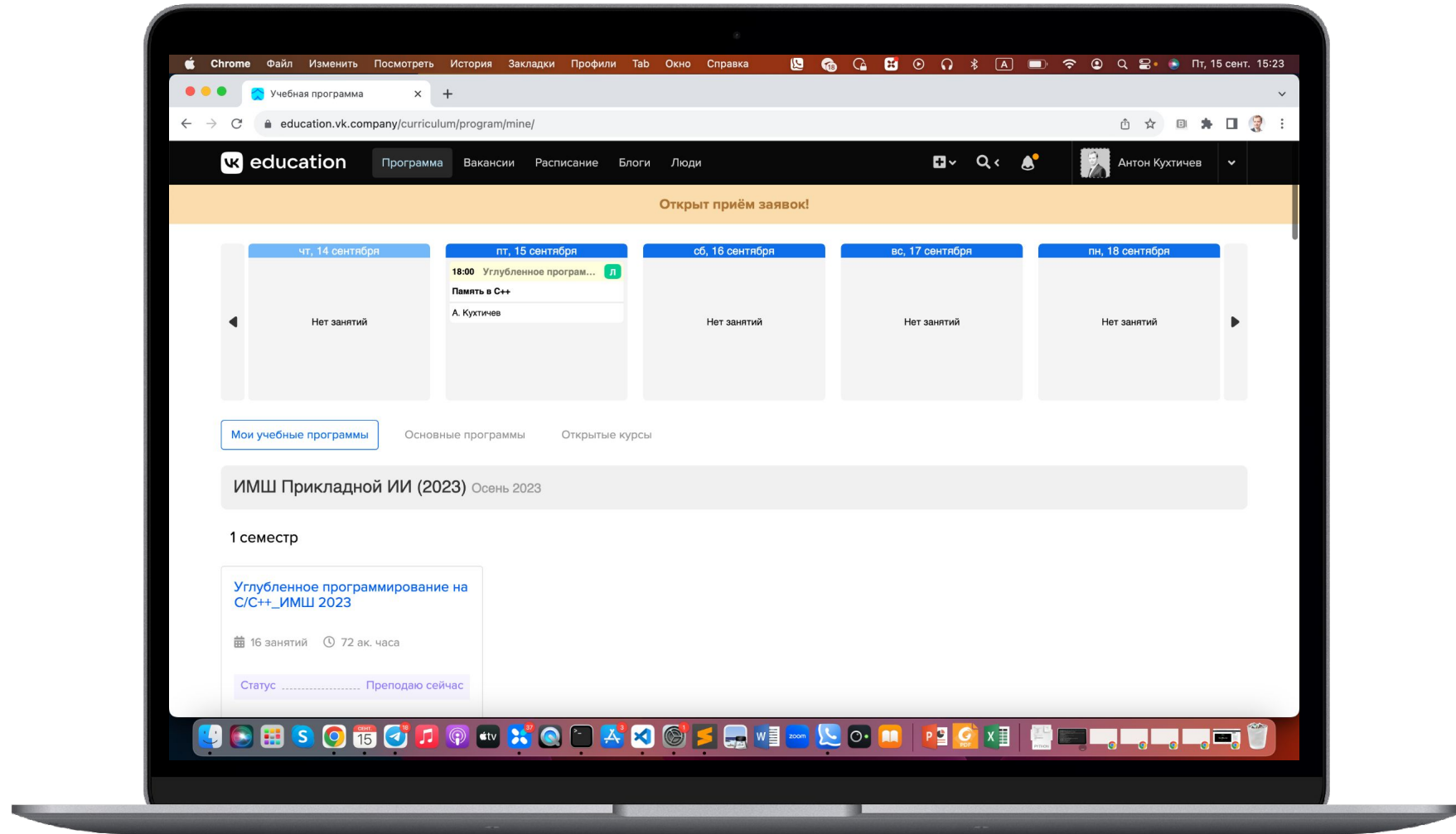
# C++ обошёл C в рейтинге языков\*

Mar 2025	Mar 2024	Change	Programming Language		Ratings	Change
1	1			Python	23.85%	+8.22%
2	3	▲		C++	11.08%	+0.37%
3	4	▲		Java	10.36%	+1.41%
4	2	▼		C	9.53%	-1.64%
5	5			C#	4.87%	-2.67%
6	6			JavaScript	3.46%	+0.08%
7	8	▲		Go	2.78%	+1.22%
8	7	▼		SQL	2.57%	+0.65%
9	10	▲		Visual Basic	2.52%	+1.09%
10	15	▲▲		Delphi/Object Pascal	2.15%	+0.94%

\* <https://www.tiobe.com/tiobe-index/>

# Напоминание отметиться на портале

и оставить отзыв  
после лекции



# Содержание занятия

- Этапы компиляции
- Препроцессор
- Объектный файл
- Компиляция
- Компоновка
- Полезные флаги компиляции
- Статические библиотеки
- Динамические библиотеки
- Модули<sup>C++20</sup>
- Утилита для автоматизации

# Цель занятия

- Сформировать знание об этапах компиляции в C++ и их назначении
- Сформировать понимание роли объектного файла
- Сформировать знание об основных флагах компиляции, чтобы уверенно работать с процессом сборки программ



# Мем недели



**Когда ты знаешь только JavaScript,  
а твои коллеги обсуждают C++**

# 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++20 hello.cpp -o hello
```

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



# Трансляторы

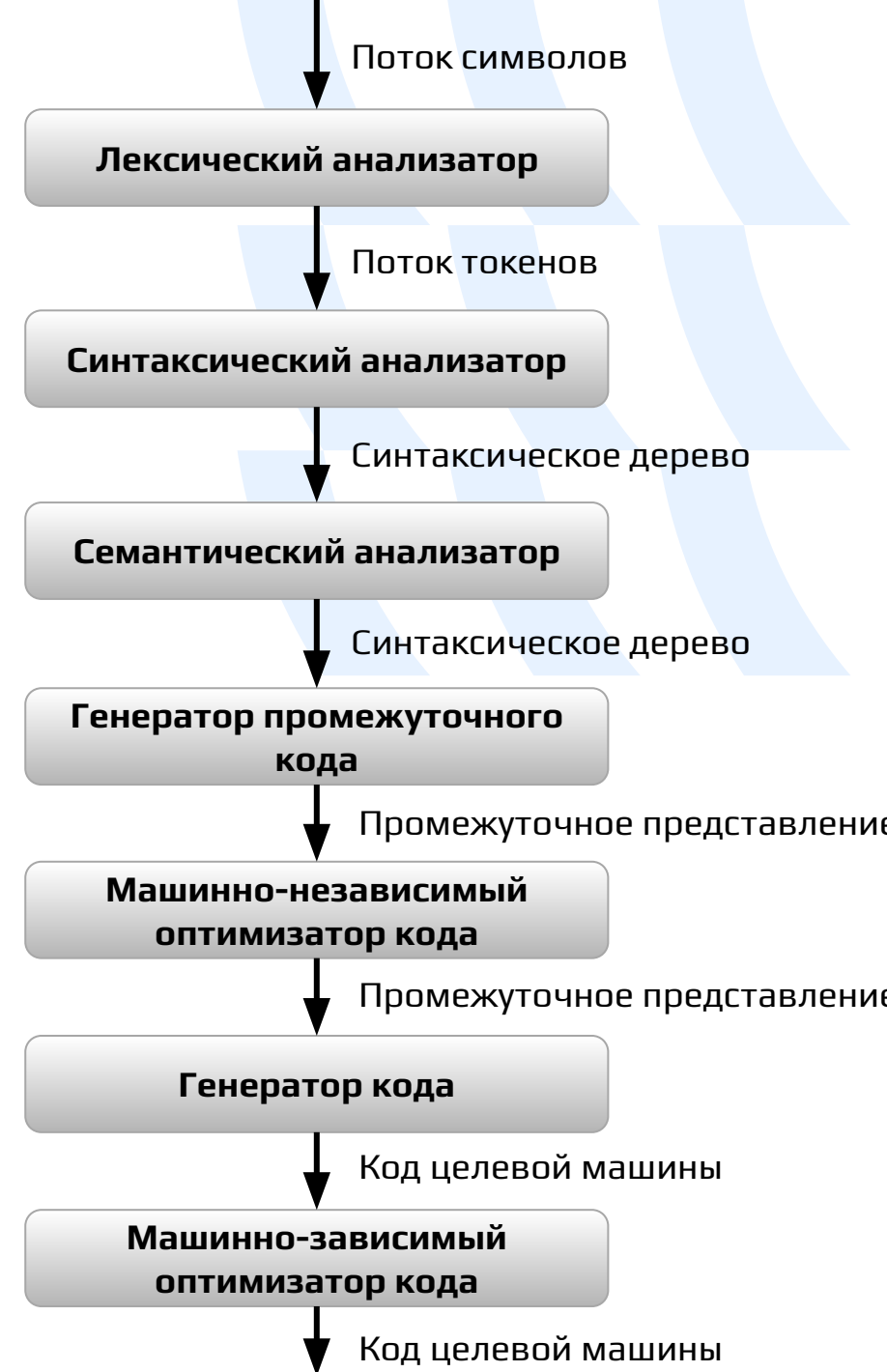
## 1. Компиляторы

Транслирует исходный текст программы на язык ассемблера или в машинные команды.

## 2. Интерпретаторы

Транслирует исходный текст в операции, которые могут состоять из нескольких групп машинных команд, и немедленно выполняет эти операции.

# Структура компилятора



Альфред Ахо, Джеффри Ульман, Моника С. Лам. "Компиляторы: принципы, технологии и инструментарий".

Константин Владимиров. "Оптимизирующие компиляторы. Структура и алгоритмы"

# 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



# Препроцессор (1)

- Делаются макроподстановки:
  - определения (`#define`, `#undef`);
  - условные включения (`#ifdef`, `#ifndef`, `#if`, `#endif`, `#else`, `#elifdef`, `#elifndef` и `#elif`);
  - директива `#line`;
  - директива `#error`, `#warning`;
  - 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 (\*)

- Объектно-подобный макрос

```
#define <NAME> <CODE>
```

- Функционально-подобный макрос

```
#define <NAME>(<PARAMETERS>) <CODE>
```



Мейерс С. “Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов”. Правило 1. Предпочитайте `const` и `inline` использованию `#define`.

# Препроцессор (4). Макрос #define

- `#define PI 3.141592`
- `#define MAX(x, y) ( x > y ? x : y )`
- `#define MULT(x, y) x * y`



## Препроцессор (4). Макрос #define

- #define PI 3.141592

Если при использовании `PI` будет ошибка компиляции, то в сообщении от компилятора увидите значение `3.141592`, 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 (5). Условная компиляция

```
#ifndef MY_MACRO
```

```
std::cout << "Hello" << std::endl;
```

```
#else
```

```
std::cout << "Bye" << std::endl;
```

```
#endif
```

```
$ g++ -std=c++20 macro.cpp -o macro
```

```
$ ./macro
```

```
Hello
```

```
$ g++ -std=c++20 -DMY_MACRO macro.cpp -o macro
```

```
$ ./macro
```

```
Bye
```

## Препроцессор (6). Двойное включение

- Чтобы защититься от двойного включения одного и того же заголовочного файла, и не словить ошибку компиляции, используется **страж включения** (**include guard**, или предохранитель включения).

```
#ifndef HEADER_NAME_HPP
```

```
#define HEADER_NAME_HPP
```

```
// Содержимое заголовочного файла
```

```
#endif
```

- Большинство компиляторов поддерживают отдельную директиву

```
#pragma once
```

```
// Содержимое заголовочного файла
```



Джош Лоспинозо “С++ для профи. Молниеносный старт”. Отдельные моменты компиляции.  
Стивен Дьюхерст. “С++. Священные знания” Тема 62. Стражи включения

## Препроцессор (7). Переменное количество аргументов

```
#define execute(com, ...) com(__VA_ARGS__)
```

```
execute(sprintf, "%s=%d", "len", 10) // sprintf("%s=%d", "len", 10)
```

# Препроцессор. Примерчик

```
// example.cpp
1. #include <iostream>
2. #define NAME(world) #world
3. int main(int argc, char **argv)
4. {
5.     #line 100
6.     std::cout << "Hello, " << NAME(world) << " from "
               << __FILE__ << " and line #" << __LINE__
               << std::endl;
7. }
```

// Опция -E запускает только препроцессор.

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

# Компиляция



# Компиляция/Ассемблирование

Файлы .cpp/.c — один файл с исходным кодом — один объектный файл. Это называется **единица трансляции (Translation unit, TU)**.

```
# Компиляция и ассемблирование: создать объектный файл
# 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-0 (Mach object) на семействе MacOS;
  - PE (Portable Executable) для Windows;

– 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-0 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)

## Объектный файл (4)

- 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)

```
$ 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 (5)

- 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 (6)

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

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

```
SYMBOL TABLE:
```

```
0000000000000000 1      df *ABS*      0000000000000000 square.cpp
0000000000000000 1      d  .text  0000000000000000 .text
. . .
0000000000000000 1      d  .comment  0000000000000000 .comment
0000000000000000 g      F  .text  0000000000000010 _Z6squarei
```

# Объектный файл (7)

```
$ 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



# 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.

# Некоторые полезные флаги



# Ключи оптимизации

« Хочется стабильности — -02, »  
хочется квестов — -03

- -00 — отключение оптимизации (по умолчанию);
- -01 — пытается уменьшить размер кода и ускорить работу программы.  
Соответственно увеличивается время компиляции. При указании -0 активируются следующие флаги: -fthread-jumps, -fdefer-pop.
- -02 — GCC выполняет почти все поддерживаемые оптимизации, которые не включают уменьшение времени исполнения за счет увеличения длины кода.
- -03 — оптимизирует ещё немного. Включает все оптимизации -02 и также включает флаг -finline-functions и -fweb.

## Другие ключи (1)

- `-Wunused-variable` — предупреждение об неиспользуемых переменных;
- `-Wall` — вывод сообщений о всех предупреждениях или ошибках, возникающих во время компиляции программы. "Агрегатор" базовых предупреждений;
- `-Wextra` — "агрегатор" дополнительных предупреждений;
- `-Werror` — делает все предупреждения ошибками;
- `-pedantic` — требует строгого соответствия стандарту;
- `-Wsign-conversion` — предупреждение о попытке конвертировать из знакового в беззнаковое;



Помоги компилятору помочь тебе  
<https://habr.com/ru/post/490850/>

## Другие ключи (2)

- `-fsanitize=address` — умеет ловить использование освобождённой памяти, переполнения и утечки;
- `-g` — запрашивает, чтобы компилятор и компоновщик генерировали и сохраняли информацию о символе в самом исполняемом файле (для `gdb`);
- `-pg` — генерирует информацию необходимую для профилировщика `gprof` (файл `gmon.out`);
- `-fno-builtin` — предотвращает распознавание компилятором стандартных библиотечных функций (`memcpy`, `printf`, `strlen`) как встроенных функций.

# Статические и динамические 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.

# Динамические библиотеки

В Unix принято, что динамические библиотеки имеют префикс `lib` и расширение `.so`.

```
$ g++ -std=c++20 -fPIC -shared square.cpp -o libsquare.so
```

- флаг `-fPIC` (Position Independent Code) - генерируемый компилятором код должен быть независимым от адресов;

```
$ g++ -std=c++20 -L. main.cpp -lsquare -o main
```

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

```
$ LD_LIBRARY_PATH=./:${LD_LIBRARY_PATH} ./main
```

- `LD_LIBRARY_PATH` — путь, где линковщик будет искать динамические библиотеки

# Модули<sup>C++20</sup>



# Что не так с include'ами?

- По сути это аккуратный способ скопировать все из одного файла и вставить в другой;
- Риск создания длительного времени компиляции (особенно для сложного шаблонного кода).



# Преимущество модулей

- Модули импортируется один раз и буквально бесплатно;
- Не имеет значения, в каком порядке вы импортируете модуль;
- Модули позволяют выразить логическую структуру кода;
- Кроме того, вы можете объединить несколько модулей в более крупный модуль и предоставить их вашему клиенту в виде логического пакета;
- Благодаря модулям нет необходимости разделять исходный код на интерфейс и часть реализации.



[C++ Russia 2019. Дмитрий Кожевников — Модули в C++20 — правда или вымысел?](#)

C++Russia 2025. Константин Владимиров — Каша из топора: модули в C++, проблемы и решения

# Линковка на уровне модулей

- До появления стандарта C++20 в C++ было два типа линковки - внутренняя и внешняя
  - **Внутренняя линковка** – имена с внутренней линковкой недоступны вне единицы компиляции
  - **Внешняя линковка** – имена с внешней линковкой доступны извне единицы компиляции
- Модули добавляют новый тип линковки - линковку модулей (*module linkage*):
  - **Линковка модулей** – имена с линковкой модулей доступны только внутри модуля. У имен задана линковка модуля, если они не имеют внешней линковки и не экспортируются



# Примерчик

```
// mexample.cpp
export module mexample;

#include <iostream>

export int square(int value) { return value * value; }
```

```
g++ -std=c++20 -fmodules-ts -c mexample.cpp
g++ -std=c++20 -fmodules-ts -c main.cpp
g++ main.o mexample.o -o ./main
```

# 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 Makefile

```
CC=g++
```

```
FLAGS=-std=c++20 -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 Makefile

```
CC=g++
```

```
FLAGS=-std=c++20 -Wall -Werror -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:
```

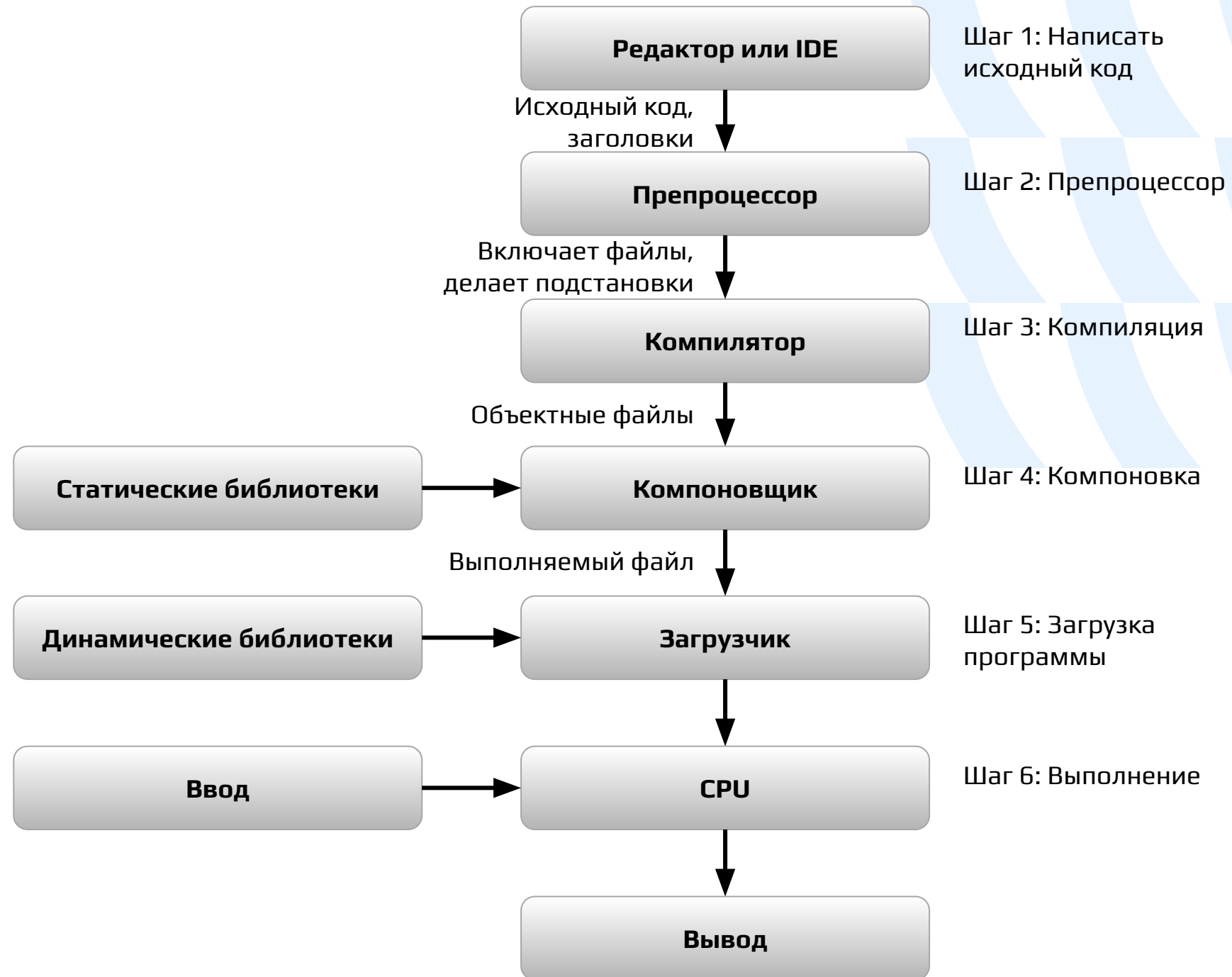
```
    rm -rf *.o my_prog
```

# Code time!



- Пробуем написать динамическую библиотеку;
- Смотрим работу препроцессора, компилятора, компоновщика;
- Побалуемся с флагами;
- Смотрим основные ошибки, которые может выдать компилятор.

# Итог



# Подведём итоги

1

**Компиляция программы проходит через несколько этапов:** препроцессинг, компиляцию, ассемблирование и компоновку, каждый из которых выполняет свою роль в преобразовании исходного кода в исполняемый файл

2

Препроцессор обрабатывает директивы **#include**, **#define** и **условную компиляцию**, выполняя подстановки и формируя единицу трансляции, передаваемую компилятору

3

**Компилятор переводит исходный код в ассемблерный код**, анализируя его синтаксис и семантику, а затем применяет оптимизации перед генерацией объектного файла

4

**Объектный файл содержит машинный код и метаданные**, включая таблицу символов, секции **.text**, **.data**, **.bss**, которые используются для сборки финального исполняемого файла

5

**Компоновщик объединяет объектные файлы и библиотеки в исполняемый файл**, разрешая ссылки на внешние символы и устраняя избыточность

6

**Флаги компиляции (-Wall, -Werror, -O2 и другие) позволяют управлять процессом сборки**, включая генерацию предупреждений, оптимизацию и отладку

7

Различают **статические (.a) и динамические (.so) библиотеки**, которые используются для сокращения размера исполняемых файлов и повторного использования кода

8

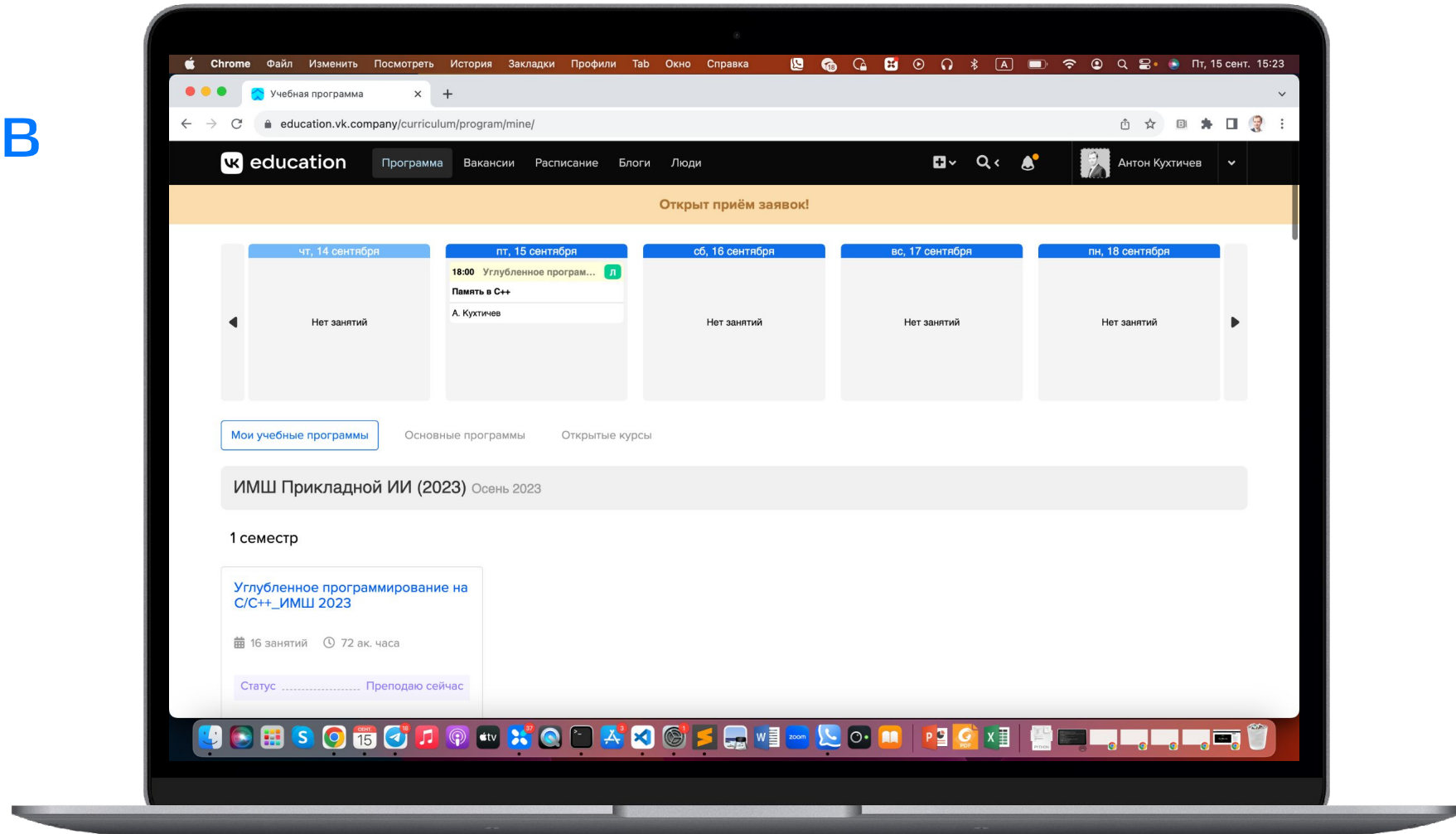
**C++20 вводит поддержку модулей, упрощающих структуру кода и уменьшающих время компиляции**, заменяя заголовочные файлы на более эффективный механизм импорта

9

**Для автоматизации сборки используют Makefile**, в котором прописаны зависимости и команды компиляции, что ускоряет процесс разработки

# Напоминание оставить отзыв

Это правда важно





Спасибо  
за внимание!