

C++ developer. Basic

Модульность.

Линковка.

Проверить, идет ли запись



**Меня хорошо
видно && слышно?**



Тема вебинара

Модульность. Линковка.



Карина Дорожкина

Research Development Team Lead

Более 10 лет опыта разработки на C/C++.

Долгое время занималась развитием ПО в области безопасности транспортного сектора.

Имею опыт руководства несколькими командами и проектами с разнообразным стеком технологий.

Спикер конференций C++ Russia, escar Europe.

dorozhkinak@gmail.com

Правила вебинара



Активно
участвуем



Off-topic обсуждаем
в telegram **#C++-basic-2023-03**

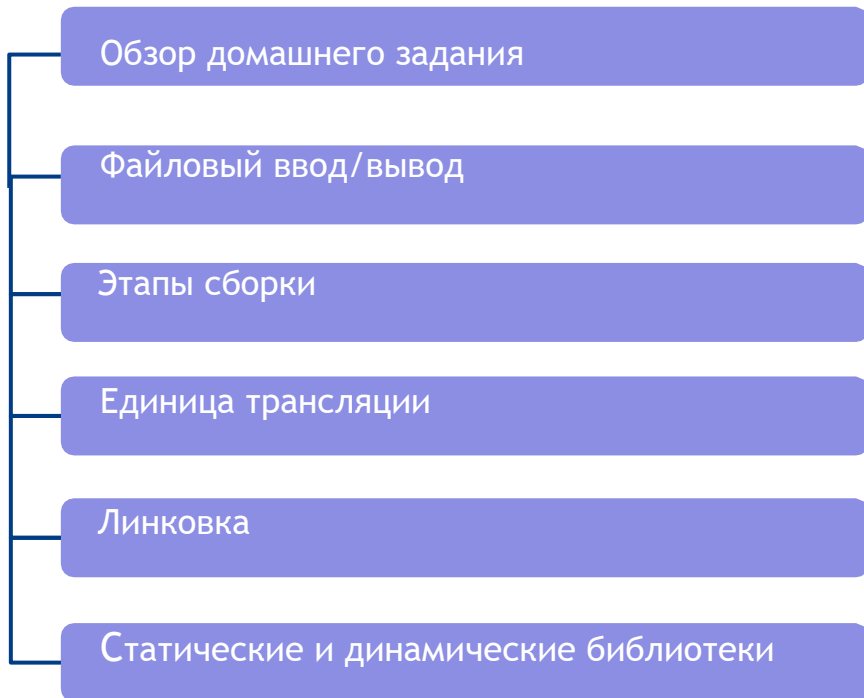


Задаем вопрос в
чат



Вопросы вижу в чате, могу
ответить не сразу

Маршрут вебинара



Файловый ввод-вывод

Файловый ввод/вывод

Почему файлы:

- Хранение
 - Между запусками программы
 - Передача данных
 - Долгосрочное



Файловый ввод/вывод

Почему файлы:

- Хранение
 - Между запусками программы
 - Передача данных
 - Долгосрочное
- Войдёт много данных



Файловый ввод/вывод

Почему файлы:

- Хранение
 - Между запусками программы
 - Передача данных
 - Долгосрочное
- Войдёт много данных
- Удобно обрабатывать



Файловый ввод/вывод

Почему файлы:

- Хранение
 - Между запусками программы
 - Передача данных
 - Долгосрочное
- Войдёт много данных
- Удобно обрабатывать
- Базовое понятие большинства ОС



Файловый ввод/вывод

Почему файлы:

- Хранение
 - Между запусками программы
 - Передача данных
 - Долгосрочное
- Войдёт много данных
- Удобно обрабатывать
- Базовое понятие большинства ОС
- Более востребовано, чем консоль



Файловый ввод/вывод

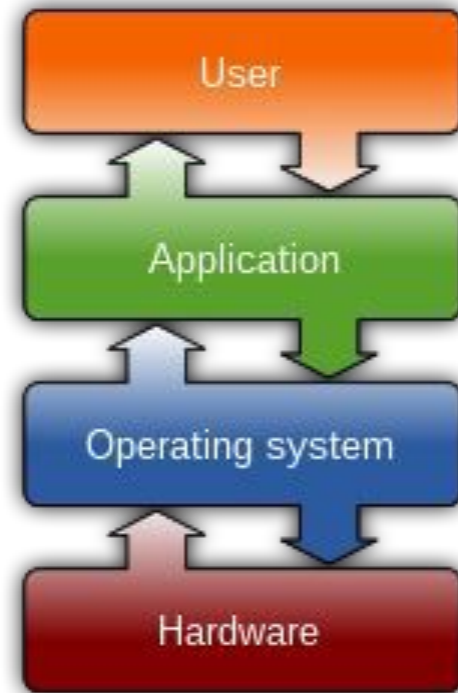
Что такое файл?

Это интерфейс взаимодействия с ФС.

Файловая система - организация хранения данных на

носителях. Каждая файловая система - своя организация.

У одной ОС может быть несколько ФС.



Файловый ввод/вывод

Как работать с файлами в коде?

Для Windows:

```
HANDLE CreateFile(  
    LPCSTR lpFileName,  
    DWORD dwDesiredAccess,  
    DWORD dwShareMode,  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    DWORD dwCreationDisposition,  
    DWORD dwFlagsAndAttributes,  
    HANDLE hTemplateFile  
);  
  
BOOL ReadFile(  
    HANDLE hFile,  
    LPVOID lpBuffer,  
    DWORD nNumberOfBytesToRead,  
    LPDWORD lpNumberOfBytesRead,  
    LPOVERLAPPED lpOverlapped  
);
```

Для Linux:

```
int open(const char *pathname, int flags, mode_t mode);  
  
ssize_t read(int fd, void *buf, size_t nbytes);
```



Файловый ввод/вывод

Как работать с файлами в коде?

Для Windows:

```
HANDLE CreateFile(  
    LPCSTR lpFileName,  
    DWORD dwDesiredAccess,  
    DWORD dwShareMode,  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    DWORD dwCreationDisposition,  
    DWORD dwFlagsAndAttributes,  
    HANDLE hTemplateFile  
);
```

```
BOOL ReadFile(  
    HANDLE hFile,  
    LPVOID lpBuffer,  
    DWORD nNumberOfBytesToRead,  
    LPDWORD lpNumberOfBytesRead,  
    LPOVERLAPPED lpOverlapped  
);
```

Для Linux:

```
int open(const char *pathname, int flags, mode_t mode);
```

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

Чтобы открыть
файл

Чтобы читать файл

Файловый ввод/вывод

Как работать с файлами в коде?

Писать на “чистых” API для каждой платформы:

- Не универсально - нужно написать столько версий кода сколько ОС
- Дорого - нужно изучать и поддерживать версию под каждую ОС
- Не “плюсовое”
- Не имеет смысла при решении общих задач



Файловый ввод/вывод

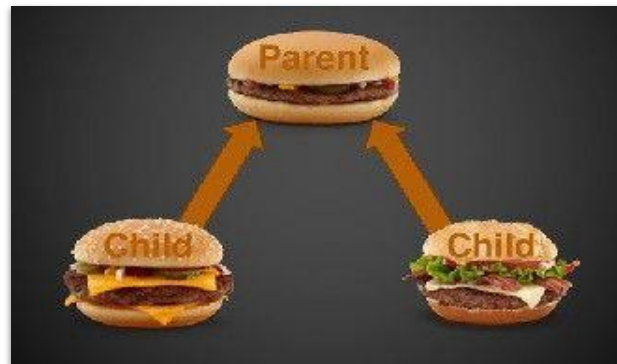
Как работать с файлами в коде?

Писать на “чистых” API для каждой платформы:

- Не универсально - нужно написать столько версий кода сколько ОС
- Дорого - нужно изучать и поддерживать версию под каждую ОС
- Не “плюсово”
- Не имеет смысла при решении общих задач

Давайте сделаем абстракцию “работа с файлом”:

1. Определяем интерфейс, без специфики ОС
2. Реализуем его под каждую нужную платформу
3. Работаем **единообразно**



Файловый ввод/вывод

В C++ задача уже решена в STL

- заголовок `<fstream>`
- содержит несколько абстракций

`std::ofstream` объект для вывода в файл (out, выходной файловый поток)

`std::ifstream` объект для чтения из файла (in, входной файловый поток)

`std::fstream` объект файла, который можно и читать и писать (in + out)



Файловый ввод/вывод

Пример C++:

```
01. #include <fstream>
02.
03. int main(int, char**) {
04.
05.     std::fstream file{"filename.txt"};
06.
07.     file << "Hello, World!" << std::endl;
08.
09.     return 0;
10. }
```



Файловый ввод/вывод

Пример C++:

Подключаем STL

```
01. #include <fstream>
02.
03. int main(int, char**) {
04.
05.     std::fstream file{"filename.txt"};
06.
07.     file << "Hello, World!" << std::endl;
08.
09.     return 0;
10. }
```

Файловый ввод/вывод

Пример C++:

```
01. #include <fstream>
02.
03. int main(int, char**) {
04.
05.     std::fstream file{"filename.txt"};
06.
07.     file << "Hello, World!" << std::endl;
08.
09.     return 0;
10. }
```

Подключаем STL

Создали объект доступа к файлу, открыли файл

Файловый ввод/вывод

Пример C++:

```
01. #include <fstream>
02.
03. int main(int, char**) {
04.
05.     std::fstream file{"filename.txt"};
06.
07.     file << "Hello, World!" << std::endl;
08.
09.     return 0;
10. }
```

Подключаем STL

Создали объект доступа к файлу, открыли файл

Это не побитовый сдвиг, а переопределённый оператор

Файловый ввод/вывод

Пример C++:

```
01. #include <fstream>
02.
03. int main(int, char**) {
04.
05.     std::fstream file{"filename.txt"};
06.
07.     file << "Hello, World!" << std::endl;
08.
09.     return 0;
10. }
```

Подключаем STL

Создали объект доступа к файлу, открыли файл

Признак конца строки

Это не побитовый сдвиг, а переопределённый оператор

Файловый ввод/вывод

Пример C++:

```
01. #include <fstream>
02.
03. int main(int, char**) {
04.
05.     std::fstream file{"filename.txt"};
06.
07.     file << "Hello, World!" << std::endl;
08.
09.     return 0;
10. }
```

Подключаем STL

Создали объект доступа к файлу, открыли файл

Признак концевой строки

Это оператор сдвига, а не оператор

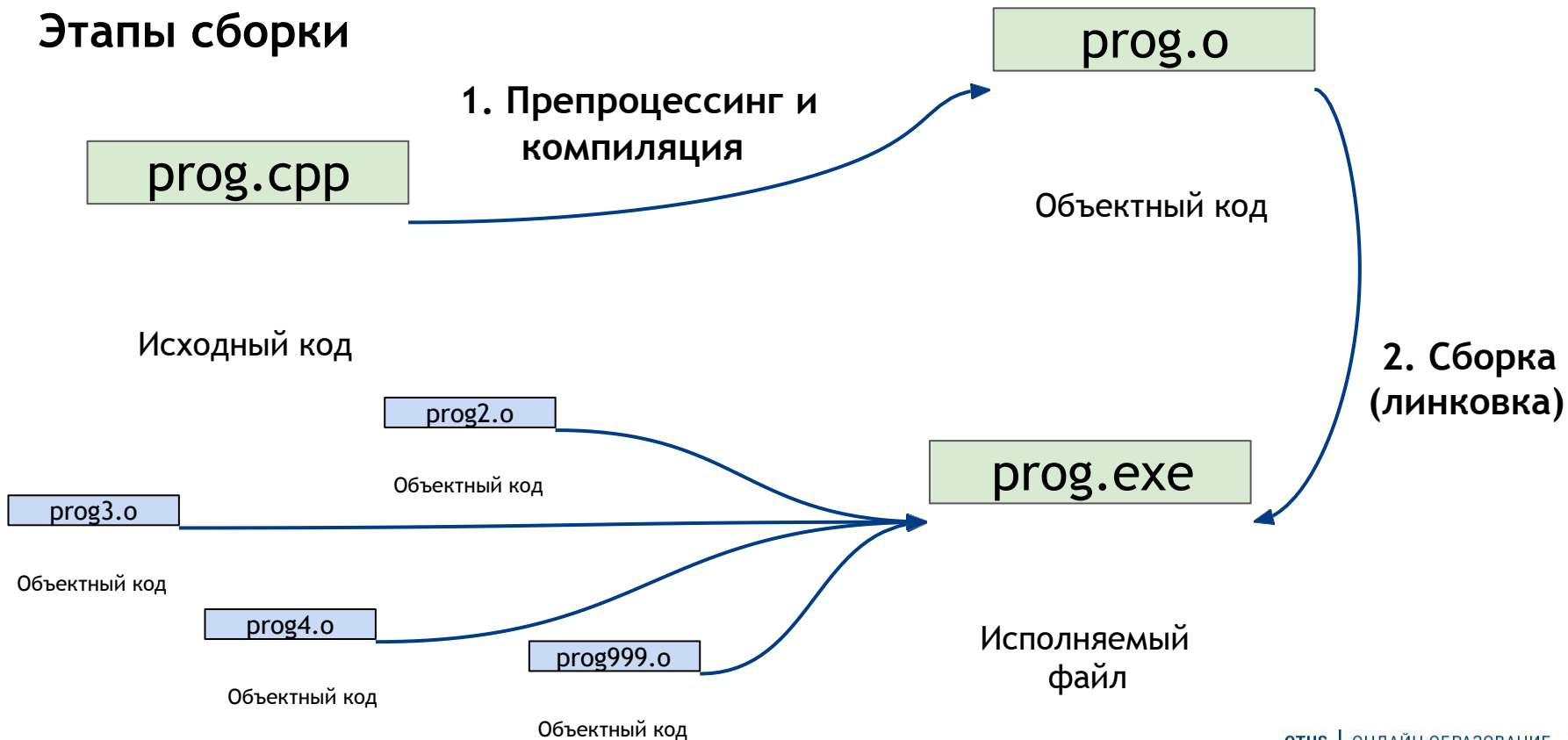
Смотри пример file_io.cpp

Этапы сборки



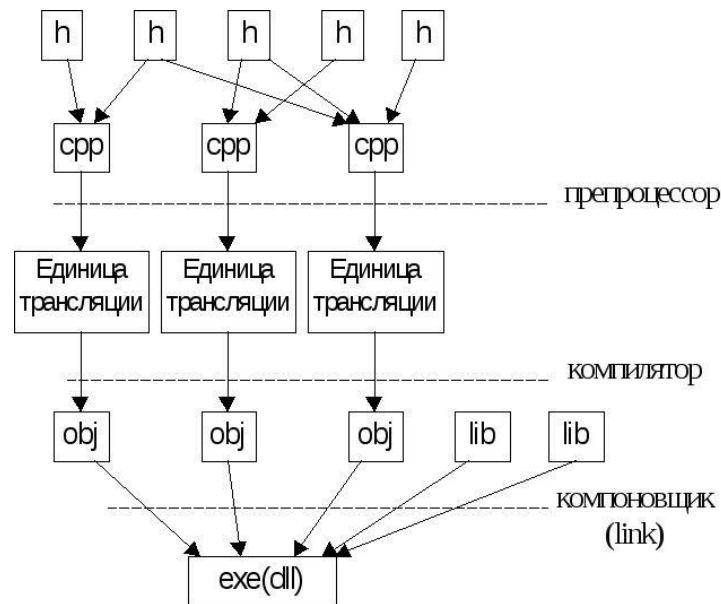
Процесс компиляции

Этапы сборки



Единица трансляции

- Берём .cpp файл
- Выполняем все директивы `#include`
- Результирующий файл будет **единицей трансляции**
- Одна единица трансляции даёт **один объектный файл**
- Удобно
 - Поддерживать структуру проекта
 - Распределять работу по команде



Процесс компиляции

Шаг 1. Препроцессинг и компиляция

- Выполняются все include-ы (и вложенные тоже, см. compilation.i)
- Работа с исходным кодом, проверка синтаксиса
- Оптимизация
- Машинный код как результат в объектном файле (010101010)
- С таблицей символов, с заглушками для адресов памяти



Объектный файл

- `objdump -d compilation.o`

Disassembly of section `__TEXT,__text`:

0000000000000014 <__Z19some_other_functionf>:

14: ff 43 00 d1	sub	sp, sp, #16
18: e0 0f 00 bd	str	s0, [sp, #12]
1c: 1f 20 03 d5	nop	
20: ff 43 00 91	add	sp, sp, #16
24: c0 03 5f d6	ret	



Name mangling

C++ overloading

```
void some_function(int a);
```

```
void some_function(std::string& s);
```

```
00000000000000044 <__Z13some_functioni>:  
44: fd 7b be a9      stp    x29, x30, [sp, #-32]!
```

```
0000000000000006c <__Z13some_functionRNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE>:  
6c: fd 7b be a9      stp    x29, x30, [sp, #-32]!
```

Таблица символов

Утилита nm показывает символы

```
nm compilation.o
```

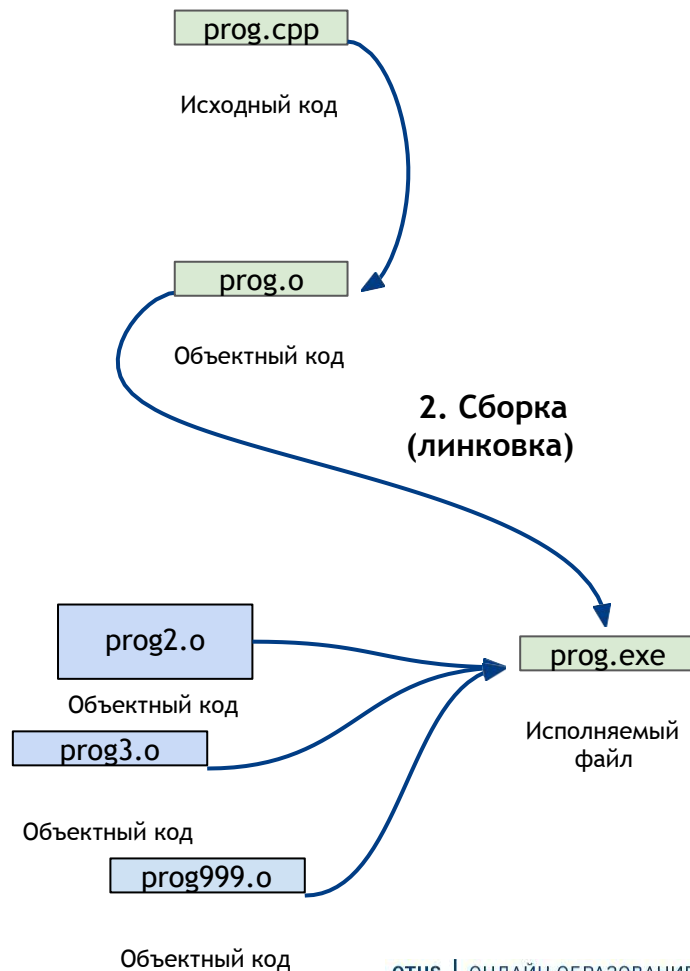
Содержит информацию об адресе смещения, типе символа и т. д.

```
0000000000000006c T __Z13some_functionRNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
00000000000000044 T __Z13some_functioni
00000000000000094 T __Z19some_other_functionf
```

Процесс компиляции

Шаг 2 - Сборка (линковка, компоновка)

- Основываясь на таблице символов
- ...которая есть у каждого объектного файла
- Строит связи между объектными файлами
- В итоге из разрозненных кусочков
- ...появляется работоспособная программа!



Траблшутинг

Отсутствие header guard => symbol redefinition

Либо

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H
#endif // FUNCTIONS_H
```

Либо

```
#pragma once
```

functions.h

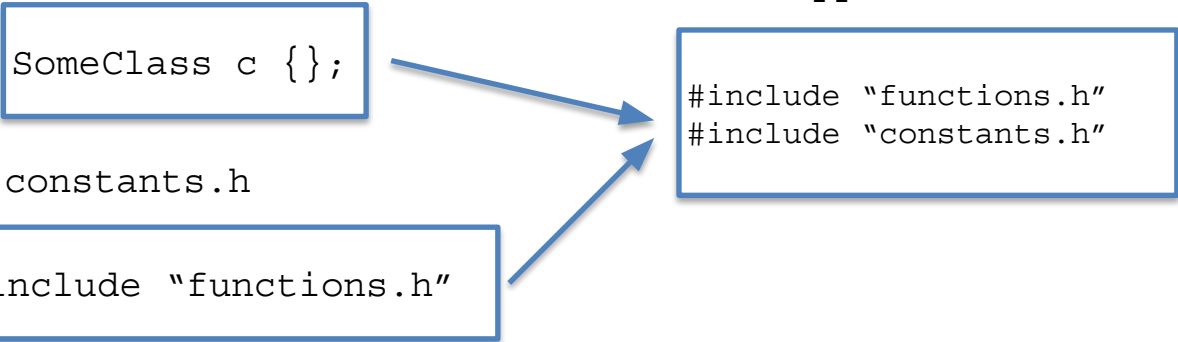
```
SomeClass c {};
```

constants.h

```
#include "functions.h"
```

main.cpp

```
#include "functions.h"
#include "constants.h"
```



Траблшутинг

Забыли определение => Undefined symbol

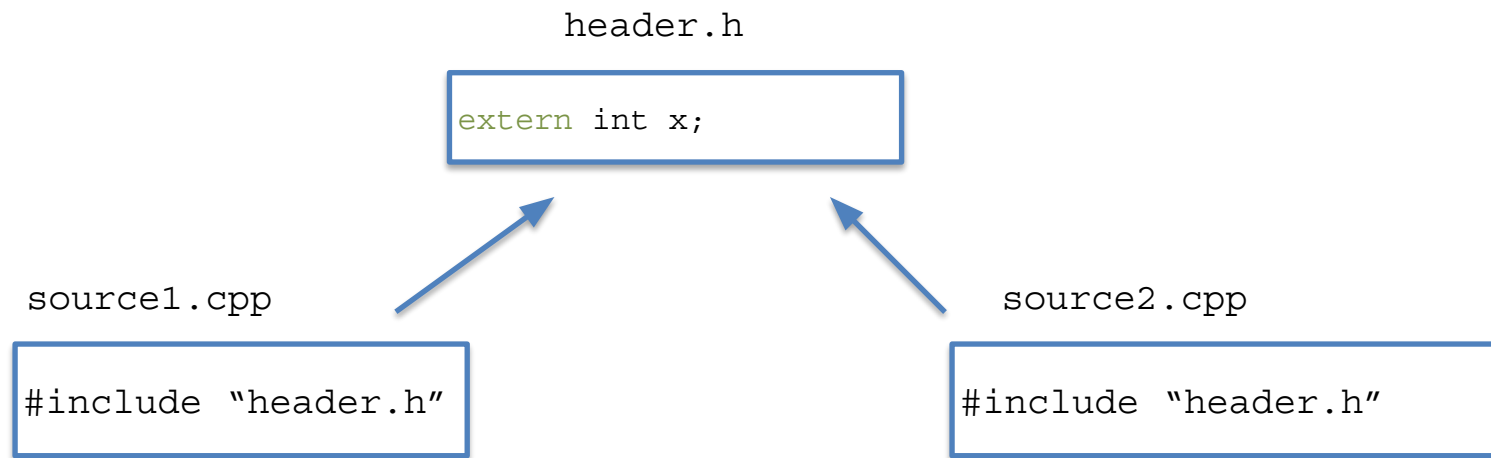
```
compilation_with_headers.cpp.o:-1: error: Undefined symbols for architecture  
arm64:
```

```
  "function_impl()", referenced from:  
    _main in compilation_with_headers.cpp.o
```



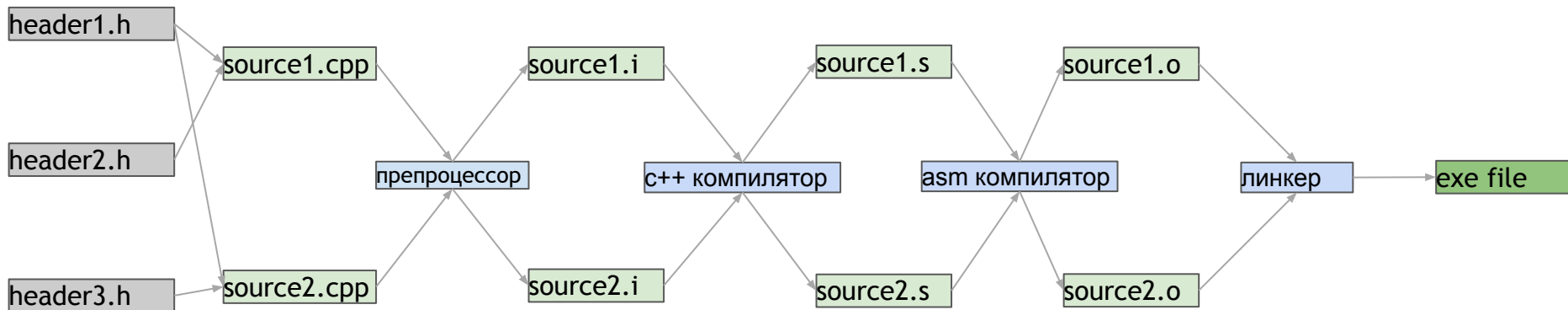
Траблшутинг

Один и тот же символ в нескольких единицах трансляции => Duplicate symbol



Процесс компиляции

Этапы сборки



Получение результатов стадий сборки

Флаги GCC

- E Preprocess only; do not compile, assemble or link; .i
- S Compile only; do not assemble or link; .s
- c Compile and assemble, but do not link; .o

Флаги MSVC

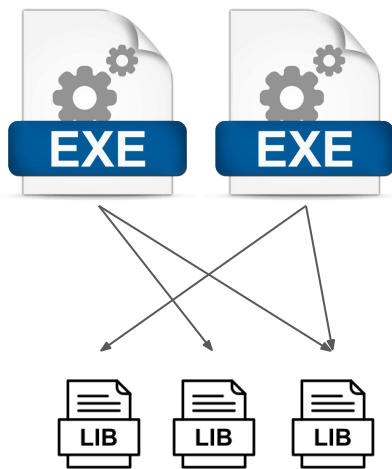
- /P Preprocessing
- /FA Assembly code; .asm
- /FAC Machine and assembly code; .cod
- /FAS Source and assembly code; .asm
- /FACS Machine, source, and assembly code; .cod

Статические и динамические библиотеки

Библиотеки

Это способ использовать готовое
в других местах

- Статические
- Динамические



Статическая библиотека

static library

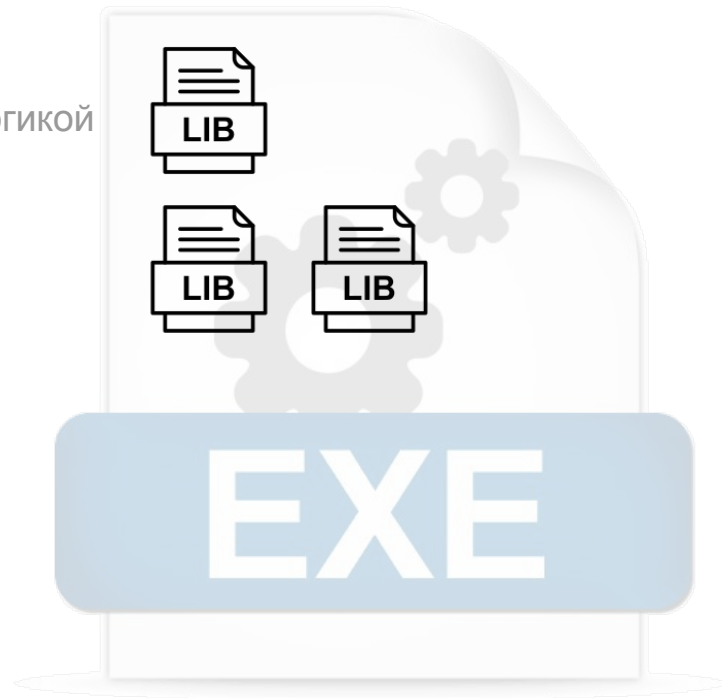
1. Берём несколько единиц трансляции с необходимой логикой



Статическая библиотека

static library

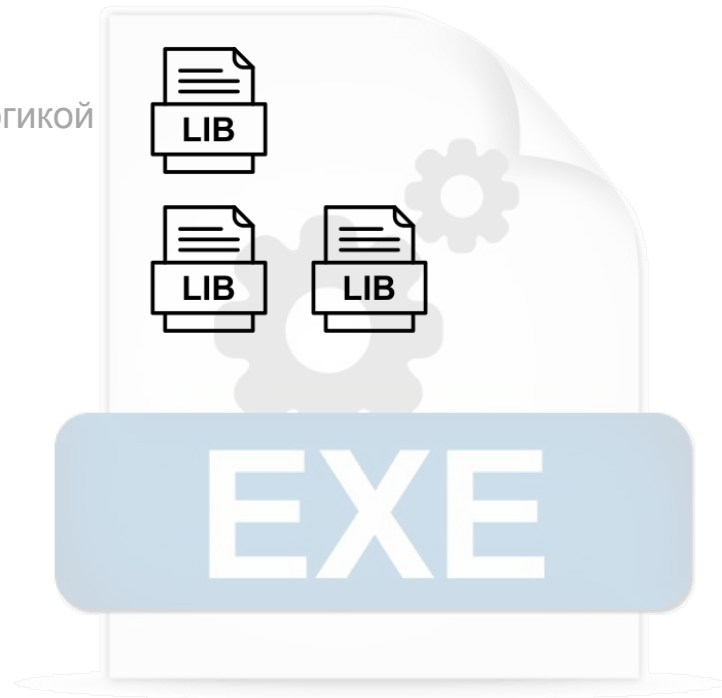
1. Берём несколько единиц трансляции с необходимой логикой
2. Делаем из них несколько объектных модулей



Статическая библиотека

static library

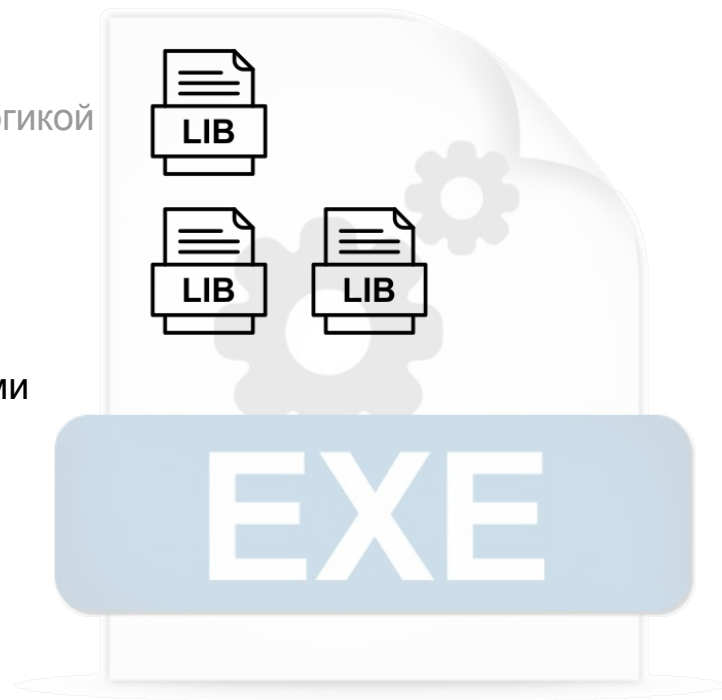
1. Берём несколько единиц трансляции с необходимой логикой
2. Делаем из них несколько объектных модулей
3. Архивируем объектные файлы в один файл



Статическая библиотека

static library

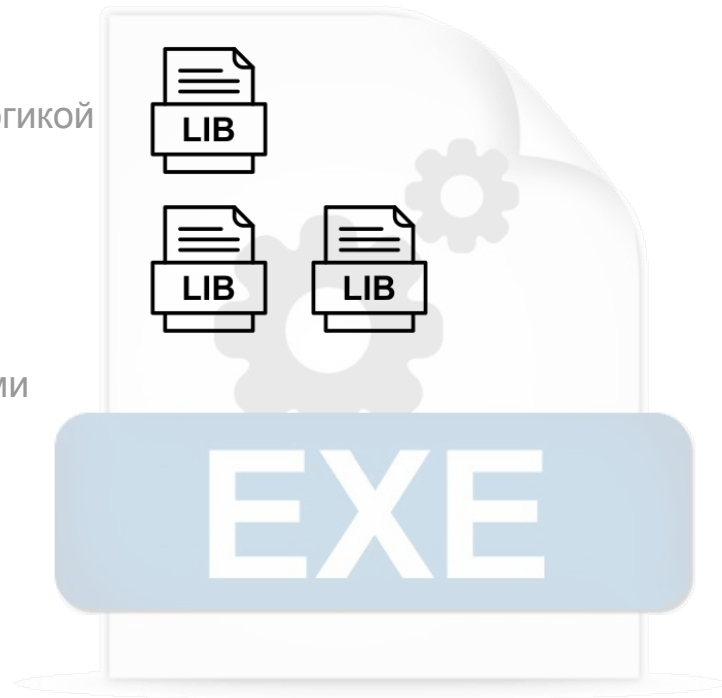
1. Берём несколько единиц трансляции с необходимой логикой
2. Делаем из них несколько объектных модулей
3. Архивируем объектные файлы в один файл
4. Добавляем набор заголовочных файлов с объявлениями



Статическая библиотека

static library

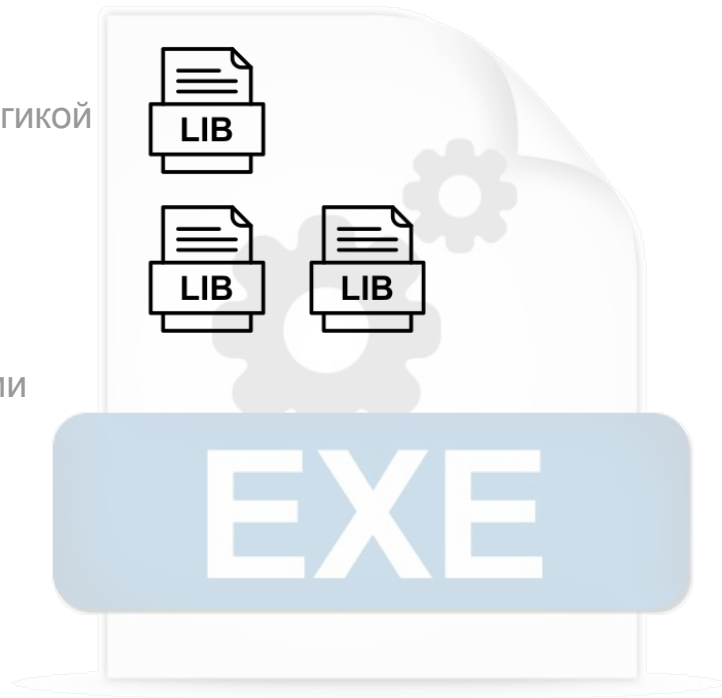
1. Берём несколько единиц трансляции с необходимой логикой
2. Делаем из них несколько объектных модулей
3. Архивируем объектные файлы в один файл
4. Добавляем набор заголовочных файлов с объявлениями
5. Отдаём всем, кому нужен наш код



Статическая библиотека

static library

1. Берём несколько единиц трансляции с необходимой логикой
2. Делаем из них несколько объектных модулей
3. Архивируем объектные файлы в один файл
4. Добавляем набор заголовочных файлов с объявлениями
5. Отдаём всем, кому нужен наш код
6. Код будет добавлен в каждый исполняемый файл



Статическая библиотека

sum.h

1

```
#pragma once  
  
int sum(const int a, const int b);
```

CMakeLists.txt

3

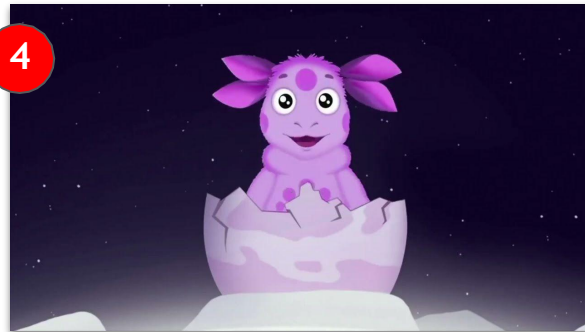
```
cmake_minimum_required(VERSION 3.5)  
  
project(static_library)  
  
add_library(sumLib STATIC  
            sum.cpp  
            )
```

sum.cpp

2

```
#include "sum.h"  
  
int sum(const int a, const int b) {  
    return a + b;  
}
```

4

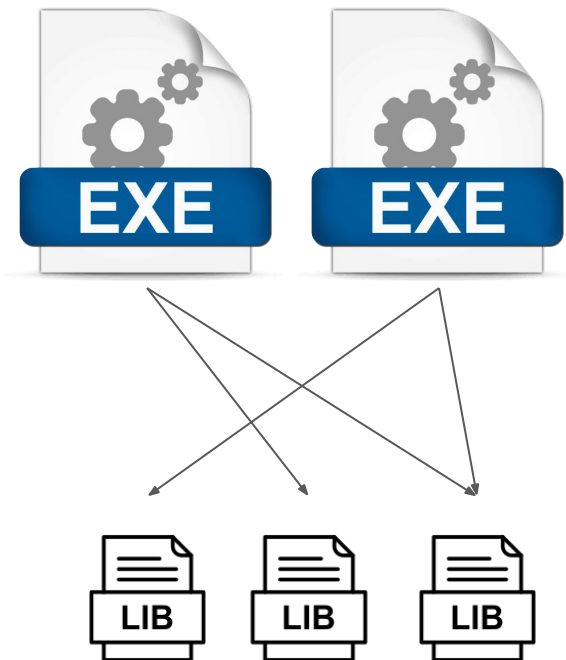


Windows: **SUMLIB.LIB**
Linux: **LIBSUMLIB.a**

Динамическая библиотека

dynamic library, shared library

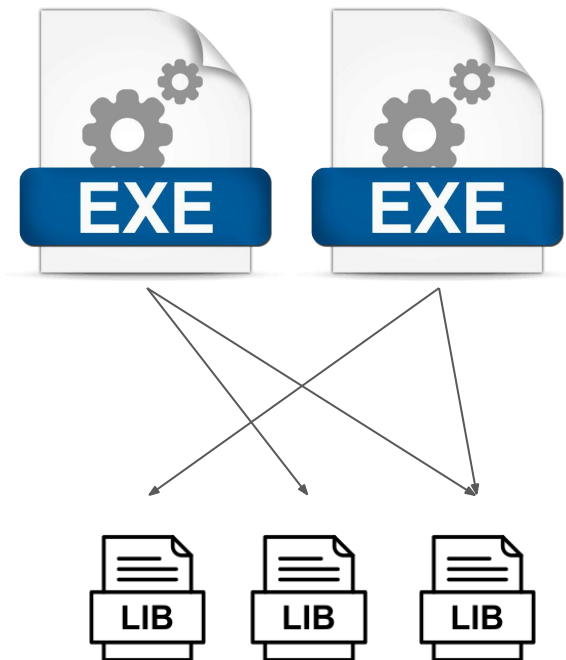
1. Берём несколько единиц трансляции с необходимой логикой
2. Делаем из них несколько объектных файлов
3. Линкуем всё вместе
4. Фактически, получая готовый к исполнению код



Динамическая библиотека

dynamic library, shared library

1. Берём несколько единиц трансляции с необходимой логикой
2. Делаем из них несколько объектных файлов
3. Линкуем всё вместе
4. Фактически, получая готовый к исполнению код
5. Добавляем набор заголовочных файлов с объявлениями
6. Отдаём всем, кому нужен наш код

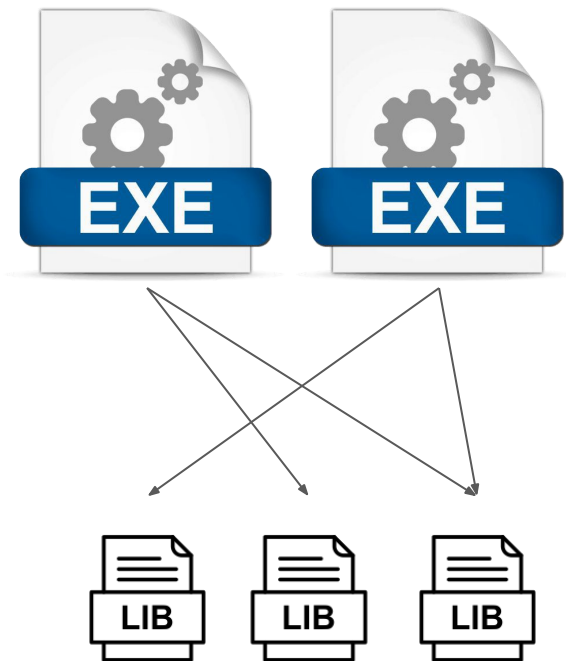


Динамическая библиотека

dynamic library, shared library

1. Берём несколько единиц трансляции с необходимой логикой
2. Делаем из них несколько объектных файлов -

ассемблируем и линкуем
3. Фактически, получая готовый к исполнению код
4. Добавляем набор заголовочных файлов с объявлениями
5. Отдаём всем, кому нужен наш код
6. Код будет добавлен лежать в одном месте
7. ... а использоваться, потенциально, в нескольких проектах



Динамическая библиотека

sum.h

1

```
#pragma once  
  
int sum(const int a, const int b);
```

CMakeLists.txt

3

```
cmake_minimum_required(VERSION 3.5)  
  
project(dynamic_library)  
  
add_library(sumLib SHARED  
            sum.cpp  
            )
```

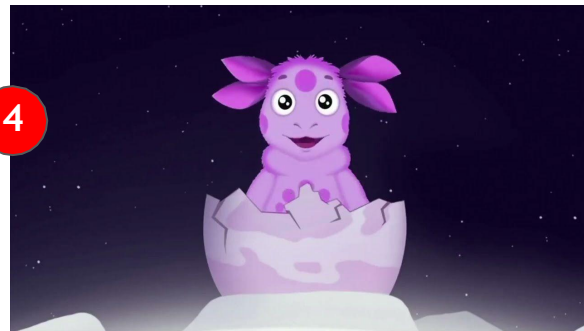
Это лишь разница
“STATIC” → “SHARED”?

sum.cpp

2

```
#include "sum.h"  
  
int sum(const int a, const int b)  
{ return a + b;  
}
```

4



Windows: SUMLIB.DLL и SUMLIB.LIB
Linux: LIBSUMLIB.SO

Динамическая библиотека

Бинарный файл хранит информацию об используемых динамических библиотеках

```
$ ldd /bin/ls
linux-vdso.so.1 => (0x00007fff3f3a5000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f0418ac7000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f04186fd000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f041848d000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f0418289000)
/lib64/ld-linux-x86-64.so.2 (0x00007f0418ce9000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f041806c000)
```

Когда запускается программа `ls`, ОС передаёт управление в `ld-linux.so.2` вместо нормальной точки входа в приложение. В свою очередь `ld-linux.so.2` ищет и загружает требуемые библиотеки, затем передаёт управление на точку старта приложения.

Сравнение библиотек?

Статические (static) библиотеки:

- Проще создавать, проще использовать
- Дублируют код
- Нужно пересобирать исполняемый файл

Динамические (dynamic, shared) библиотеки:

- Не дублируют код
- Обновление без исполняемого файла
- Динамическая загрузка и выгрузка (как плагины)
- Ошибка при запуске, если библиотека потерялась



Подведём итоги

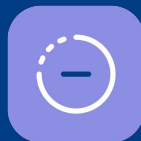
1. Подготовились к выполнению ДЗ узнав как работать с файлами
2. Познакомились с определением “единица трансляции”
3. Прошлись по этапам сборки: компиляция, линковка
4. Узнали как переиспользовать готовое с помощью библиотек
5. Начали готовиться к следующему занятию по CMake



Вопросы?



Ставим “+”,
если вопросы есть



Ставим “-”,
если вопросов нет



**Заполните, пожалуйста, опрос о
занятии
по ссылке в чате**