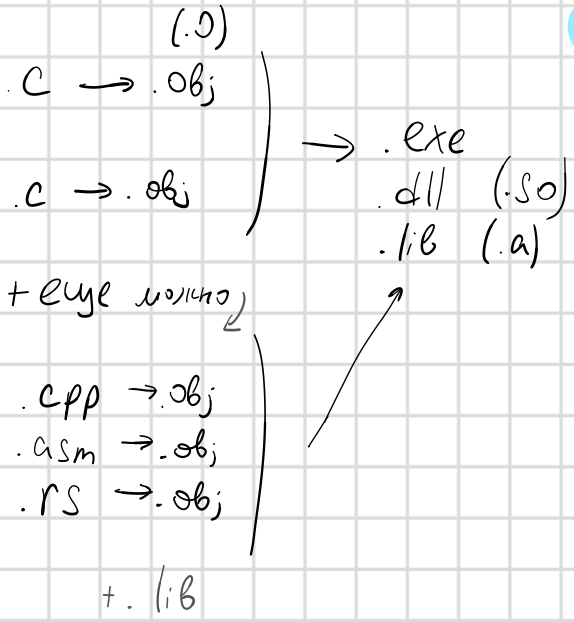
Компиляция



❗ В общем случае (заголовочный файл) h-файл содержит объявление функций (не библиотека)

Обычные библиотеки подключаются на этапе "линковки" и явл. .lib

- Исполняемый код выносим в .c файлы
- не хотим инклудить один с файл в другой (линкер их видит)
- при исп-ии ф-ии из другого файла ее нужно сначала объявить без реализации и скомпилировать оба файла
- h-файлы нужны для избавления от копипасты
- при объявлении h-файла используем #pragma once
- добавляем h-файл в оба файла чтобы избежать ошибки при изменении реализации

сборка статической библиотеки

1) Компилируем clang -c файл.c ...

2) lib /out:printhw_static.lib .\print_hw.o

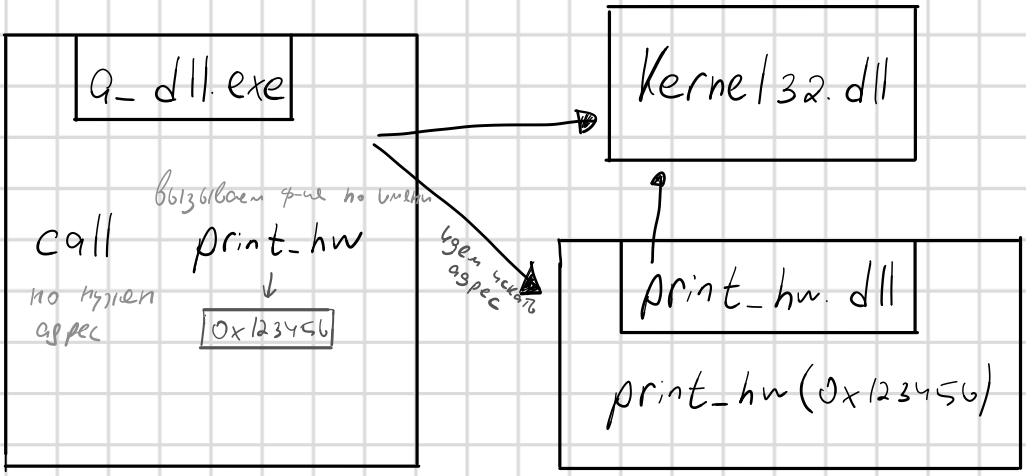
Имя, которое хотим — O-файл(ы), которые хотим объединить

3) если сидим в любом cmd vc vars bat (настроить переменное окружение)

4) Ключик -l {файл} (имена без пробела и .lib)

сборка динамической библиотеки

```
clang -o print_hw.dll -shared .\print_hw.c
```



O2 скорость ↑ (безопасные оптимизации)

O3 доп. опт-и, но может привести к замедлению глобально

LTO (link time optimisation)

для clang ключик: -flto

Компилируем → получаем недокомпилированный один файл → линкер опять вызывает компилятор для него и происходит оптимизации (нельзя оптимизировать ф-ию не зная ее тело)

(13) время линковки может очень сильно замедлится при больших файлах

(13) -flto=thin делит оптимизацию на блоки и увел. скорость линковки

(13) LTO не работает по динамическим библиотекам. Для статических нужен ключик.

(13) происходит сильная связь линкера и компилятора (не заменить другими)

Ключик -D{define} для передачи define в код на этапе компиляции

Ключик -I {путь до папки где перечислены include'ы не включительно}

Ключик -L {путь до lib файлов}

Ключик -l {имя самих файлов}

(13) Название файла с "d" это для оптимизации

можно положить .dll в ось папку и оно заработает, но нужно указать путь

Линковка - круто т.к. объединяет различные файлы, написанные на разном языке в 1 файл.

.c -> .obj |
.c -> .obj |
.c -> .obj | -> > .exe, .dll, .lib
.cpp -> .obj |
.asm -> .obj |
.rs -> .obj |

.h

Когда подключаешь .h - подключаешь ~~НЕ библиотеку~~, а описание функций (интерфейс).

Подключение библиотек .lib

- Код библиотек подключается на этапе линковки
- Разные коды по-разному его компилруют

📦 Отличия от динамической (shared) библиотеки

Характеристика	Статическая библиотека	Динамическая библиотека
Расширение	`.a` (или `.lib`)	`.so` (или `.dll`)
Связывание	Во время линковки	Во время запуска (*run-time*)
Использование памяти	Каждый исполняемый файл содержит копию	Одна копия разделяется между программами
Размер исполняемого файла	Больше	Меньше
Зависимость от внешних файлов	Нет	Да (нужен `.so` при запуске)

Виктория

Все выносы в .c файл

1. Нужно выносить большие взаимосвязанные куски кода в отдельный файл
2. Если мы в разных проектах будем переиспользовать функции, то эти функции стоит вынести в отдельный файл

print_hw реализовано в другом файлу

```
clang -c print_hw.c main.c - ОШИБКА
```

Ошибка связана с тем, что в мэйне присутствует функция которой нет в main.c

РЕШЕНИЕ:

```
#include <stdio.h>
void print_hw(void);

int main (int argc. char *argv[])
{
    print_hw();
    return 0;
}
```

Боле хорошее РЕШЕНИЕ (.h):

`#pragma once` — это директива компилятора, которая используется в заголовочных файлах (`.h`) для **предотвращения множественного включения одного и того же файла** в процессе компиляции.
print_hw.h

```
#pragma once

void print_hw(void);
```

print_hw.c

```
#include <stdio.h>
#incluse "print_hw.h"
void print_hw(int a)
{
...
}
```

На проверке

1. Собрать статическую || динамическую библиотеку
2. Собрать .exe

Статическая:

1. Компилируем файл

```
clang -c .\print_hw.c -m64 -O2 -std=c17
```

2. Исользуем lib (настроим переменную окружения командной строки)

```
lib /out:printhw_static.lib .\print_hw.o
```

- Пример инициализации: Даша скинет
  .lib - тоже самое что и .jar (по сути архив)
- -l <название библиотеки без расширения>

```
clang main.c -m64 -O3 -l .\printhw_static -o main.exe
```

или

```
clang main.c -m64 -O3 -lprinthw_static -o main.exe
```

Динамическая на Windows

1. Собираем файл динамической библиотеки

```
clang -o print_hw.dll -shared \print_hw.c
```

2. Создаем .exe

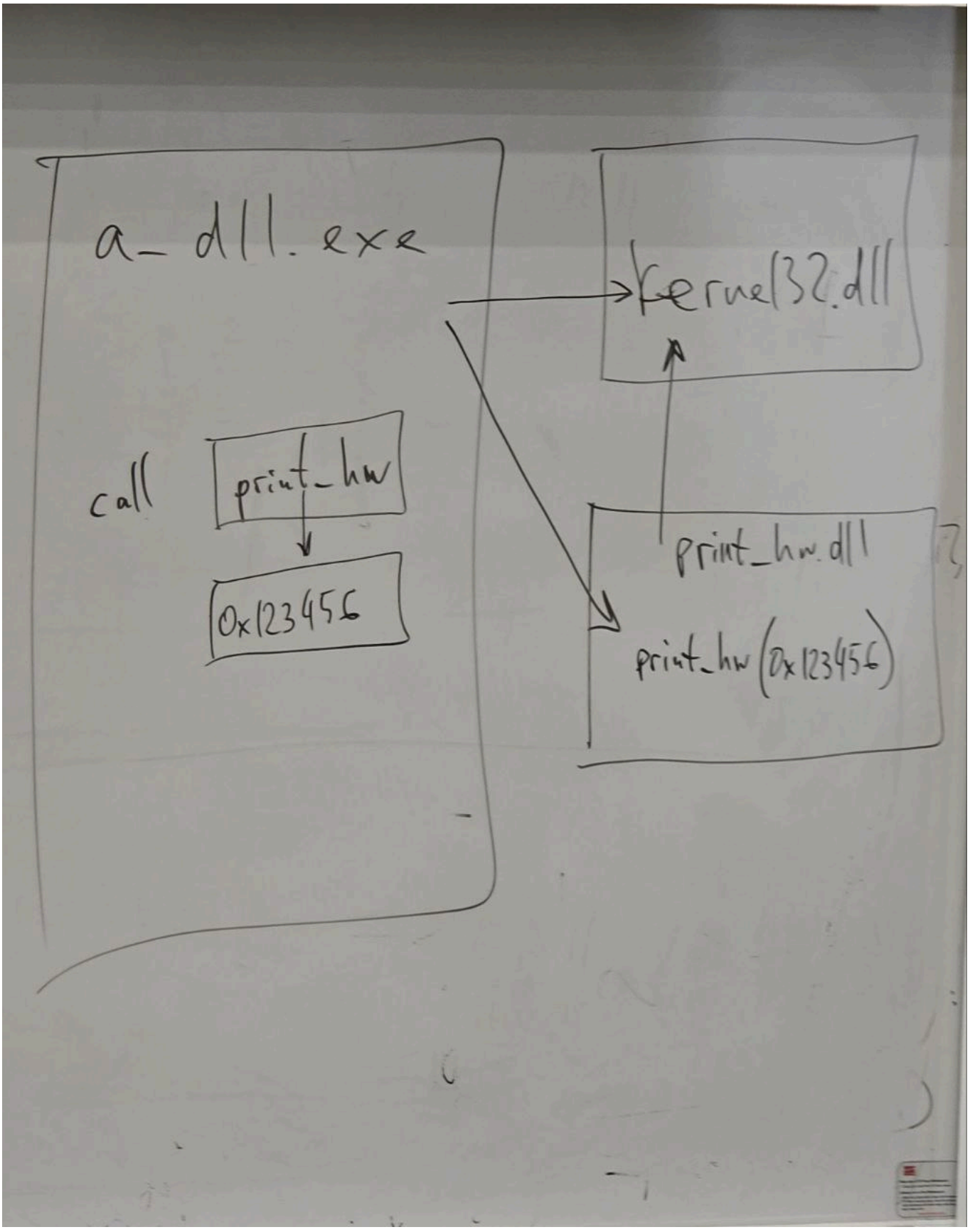
```

```

Получается .exe:
✓Он весит меньше
❌Он зависит от .dll файла и не запустится, если его нет рядом

Как работает .exe для динамической библиотеки

В .exe вначале будете call "print_hw" - после этого мы обращаемся к ОС и она заменяет print_hw на адрес в памяти где реализован данный класс



Динамические на Linux

!!! Все функции если они не static - ==экспортируются !!!
Решение:

```
clang -fvisibility=hidden ...
```

-fvisibility=hidden также очень хорошо оптимизирует код .

Компилятор умный.

InLine (работает только для функции находящейся в этом же файле)

Ключевое слово `inline` говорит компилятору:

> "Эта функция маленькая и часто используется — лучше не трать время на вызов, а просто **вставь её тело прямо туда, где она вызывается**".

Всё что inLine - хранится в кэше, если это не помещается в кэш, то оно перегружается в оперативку и всё начинает работать МЕДЛЕННЕЕ

LTO - link time optimisation

LTO - будет делать inline даже если функция вызывается из другого .c файла

-flto - в .obj будет не машинный код, а особый "полуфабрикат", соответственно нужен особый линкер, который будет собирать этот полуфабрикат в .exe

❌ - Линковка будет НАМНОООООООООООГО дольше т.к. "полуфабрикаты"
Решение:
-flto=thin - работает быстрее на больших проектах

Отличия flto

Флаг	Что делает	Особенности
`-flto`	Полная LTO: весь код загружается в один модуль	Требует много памяти и времени
`-flto=thin`	Тонкая LTO: каждое `.o` обрабатывается отдельно, но обменивается метаинформацией	Быстрее, меньше памяти, почти те же плюсы

OpenCV

В build всё собранное
build\include - Заголовочные файлы (.h)
build\x64\vc15\lib - библиотеки импорта
build\x64\vc15\bin - библиотеки импорта

Иерархия:
opencv_demo.c
library
build
incluse
...

```
clang -DOPENCV .\opencv_demo.c -I .\library\opencv\build\include -L
.\library\opencv\build\x64\vc15\lib -l (opencv_world3416d.lib ||
opencv_world3416.lib)
```

🧠 Вывод:

1. `-DOPENCV` — определяет макрос
2. `-I` — указывает путь к `.h` (include)
3. `-L` — путь к `.lib` файлам
4. `-l` — имя библиотеки (без `.lib`)
   opencv_world3416d - debug version

В VS

Создание проекта -> Desktop Wizard -> empty project

- Проект - один файл: .exe || .dll || .lib
- Solution - множество проектов

Если хочешь несколько проектов - убирай галочку

Как собрать проект в VS:

1. Properties -> Configuration = All configuration, Platform = x64 -> General -> C Language Standart = Latest
2. Properties -> Configuration = All configuration, Platform = x64 ->debugging -> Command Arguments = <аргументы с которыми запускается .exe файл

3. Properties -> Configuration = All configuration, Platform = x64 -> C/C++ -> SDL checks - УБИРАЕМ (он делает варнинги эррорами) -> General -> Additional Include Directory = (Путь до инклюда в Opencv)
4. Properties -> Configuration = All configuration, Platform = x64 -> C/C++ -> Preprocessor -> preprocessor Definitions = OPENCV
5. Properties -> Configuration = All configuration, Platform = x64 -> Linker -> General -> Additional Libary = "путь до lib директории"
6. Properties -> Configuration = All configuration, Platform = x64 -> Linker -> Input -> iditional dependenties = opencv_world3416.lib

Текущая директория (туда можно положить .dll и всё заработает):

Properties -> Configuration = All configuration, Platform = x64 ->debugging -> Working directory

Extension - Cmake tools

Отладка - Memory viewer (посмотри в гитбук, там сказано как настроить)

ВО ВРЕМЯ ОТЛАДКИ МОЖНО МЕНЯТЬ ЗНАЧЕНИ ПЕРЕМЕННЫХ (я всё ещё в это не верю)

В Debug Console (снизу)

```
variable = 10
```

CMAKE

При запуске cmake будете видеть:

- Ninja - обычно на винде
- make - обычно на windows

Cmake list

-L - пути до библиотеки импортов
VS - не любит когда в конце пути к файлу оставляет "/"

```
cmake_minimum_required(VERSION 3.10) - в какой версии хотим собраться

project(cmakke_build VERSION 1.0) - объявления имени проекта и настройки
базовых параметров сборки

set(CMAKE_C_STANDART 17) - задаём стандарт C
set(CMAKE_CXX_STANDART 17) - задаём стандарт C++

add_definitions(-DOPENCV) - макрос для OPENCV

enable_testing() - включает тесты сразу после сборки

include_directories("library/opencv/build/include") - добавить инклюды

link_directories("linrary/opencv/build/x64/vc15/lib") - тоже самое что и -L
```

```
add_executable(cmake_build opencv_demo.c) - создать экзешник cmake_buil.exe
который будет внутри хранить opencv_demo.c

targer_link_library(cmake_build "opencv_world3416.lib") - тоже самое что и -l.
ОБЯЗАТЕЛЬНО ПОСЛЕ add_executable

add_test(NAME arg_run COMMAND cmake_build "picture.png") - после билда сразу
запустить cmake_build с аргументом "picture.png"
```

Что за нас делает среда разработки

```
cmake_minimum_required(VERSION 3.10) - в какой версии хотим собраться

project(cmakke_build VERSION 1.0) - объявления имени проекта и настройки
базовых параметров сборки

set(CMAKE_C_STANDART 17) - задаём стандарт C
set(CMAKE_CXX_STANDART 17) - задаём стандарт C++
```

⚙️ Команда CMake и флаги

```
cmake -S . -B build2 -A64 -DCMAKE_INSTALL_PREFIX=C64
```

Флаги	Назначение
`-S .`	Исходная директория (`.` — текущая папка)
`-B build2`	Папка для файлов сборки (`build2`)
`-A64`	Архитектура: 64-битная сборка (обычно для Windows/VS)
`-D`	Устанавливает переменную CMake
`CMAKE_INSTALL_PREFIX=C64`	Путь, куда будет устанавливаться проект при `install`

После генерации появляется CMakeLists.txt

Build - создаёт .exe файл

```
cmake --build .\build2\ --config Realease
```

```
.\cmale_build.exe
```