CMake

Содержание

1. **Многообразие систем сборки**
2. CMake: система сборки, которая ничего не собирает
3. CMake: язык программирования, на котором не хочется писать

Многообразие систем сборки

b | 2

GNU Make

MSBuild

GNU Autotools

GNU Make

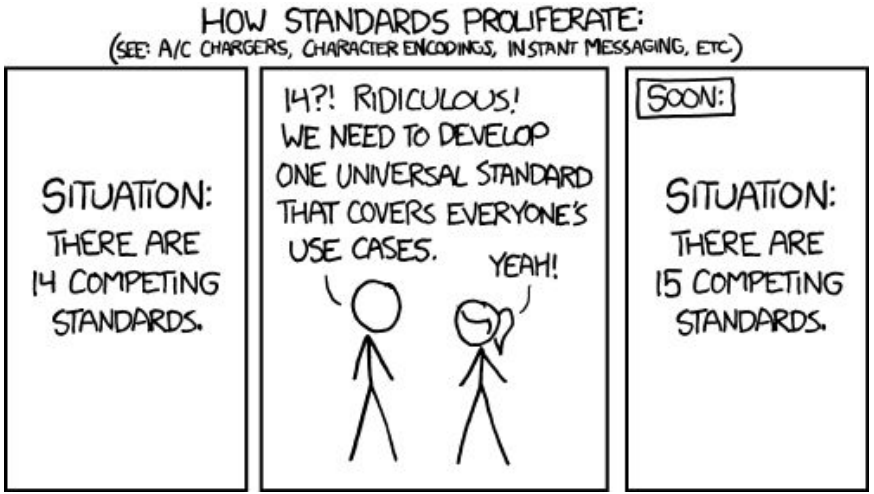
✓ Универсальный инструмент: не привязан к компиляции
✓ Не скрывает деталей и хорош для изучения процесса сборки
✓ Прост для базовых сценариев

✗ Не является кроссплатформенным
✗ Сложен в параметризации (конфигурации Release/Debug, доп. опции)

? Смешивает императивный и декларативный стили

Миру нужен герой

- Кроссплатформенный
- Расширяемый
- Не новый конкурирующий стандарт



Содержание

1. Многообразие систем сборки
2. **CMake: система сборки, которая ничего не собирает**
3. CMake: язык программирования, на котором не хочется писать

Минималистичный проект

```
.
|-- CMakeLists.txt
`-- src
    `-- main.cpp
```

Этап 1: конфигурация

```
.
|-- build
|   `-- Debug
|       |-- CMakeCache.txt
|       `-- Makefile
|-- CMakeLists.txt
`-- src
    `-- main.cpp
```

```
$ cmake -S . -B build/Debug \
   -DCMAKE_BUILD_TYPE=Debug
```

CMake не собирает проект

Он генерирует файлы для других систем сборки

Этап 2: сборка

```
.
|-- build
|   `-- Debug
|       |-- CMakeFiles
|       |   `-- hello.dir
|       |       `-- src
|       |           |-- main.cpp.o
|       |           `-- main.cpp.o.d
|       |-- CMakeCache.txt
|       |-- hello
|       `-- Makefile
|-- CMakeLists.txt
`-- src
    `-- main.cpp
```

```
$ cmake -S . -B build/Debug \
   -DCMAKE_BUILD_TYPE=Debug

$ cmake --build build/Debug
```

Не вызываем make напрямую

Этап 2: сборка

```
.
|-- build
|   `-- Debug
|       |-- CMakeFiles
|       |   `-- hello.dir
|       |       `-- src
|       |           |-- main.cpp.o
|       |           `-- main.cpp.o.d
|       |-- CMakeCache.txt
|       |-- hello
|       `-- Makefile
|-- CMakeLists.txt
`-- src
    `-- main.cpp
```

```
$ cmake -S . -B build/Debug \
   -DCMAKE_BUILD_TYPE=Debug

$ cmake --build build/Debug

$ ./build/Debug/hello

Hello, World
```

Очистка

```
.
|-- build
|   `-- Debug
|       |-- CMakeCache.txt
|       `-- Makefile
|-- CMakeLists.txt
`-- src
    `-- main.cpp
```

```
$ cmake -S . -B build/Debug \
   -DCMAKE_BUILD_TYPE=Debug

$ cmake --build build/Debug

$ ./build/Debug/hello

Hello, World

$ cmake --build build/Debug \
   --target clean
```

Артефакты сборки удалены

Сгенерированные файлы остались

Очистка

```
.
|-- CMakeLists.txt
`-- src
    `-- main.cpp
```

Это — Out-of-source build

```
$ cmake -S . -B build/Debug \
   -DCMAKE_BUILD_TYPE=Debug

$ cmake --build build/Debug

$ ./build/Debug/hello

Hello, World

$ cmake --build build/Debug \
   --target clean

$ rm -rf build
```


What is an "out-of-source" build?

In-source build

✗

```
.
|-- CMakeCache.txt
|-- CMakeFiles
|   |-- hello.dir
|   |   `-- src
|   |       |-- main.cpp.o
|   |       `-- main.cpp.o.d
|   `-- Makefile.cmake
|-- CMakeLists.txt
|-- hello
|-- Makefile
`-- src
    `-- main.cpp
```

```
$ cmake .

$ cmake --build .

$ ./hello

Hello, World!

$ cmake --build . --target clean

$ git clean -df
```

Артефакты сборки в корне проекта

In-source build

- ✗ Артефакты лежат вместе с исходниками
- ✗ Невозможно поддерживать несколько конфигураций (Release/Debug)

Нет причин собирать в режиме in-source

Используйте out-of-source build

CMAKE_BUILD_TYPE

	GCC		MSVC		
Debug:	-g		/Zi	/Ob0 /Od	
Release:	-O3	-DNDEBUG	/O2	/Ob2	/DNDEBUG
RelWithDebInfo:	-O2 -g	-DNDEBUG	/Zi /O2	/Ob1	/DNDEBUG
MinSizeRel:	-Os	-DNDEBUG	/O1	/Ob1	/DNDEBUG

Для каждого компилятора свой набор опций

Modules/Compiler/GNU.cmake#L56

Множество конфигураций

```
.
|-- build
|   |-- Debug
|   |   |-- CMakeCache.txt
|   |   |-- hello
|   |   `-- Makefile
|   `-- Release
|       |-- CMakeCache.txt
|       |-- hello
|       `-- Makefile
|-- CMakeLists.txt
`-- src
    `-- main.cpp
```

```
$ cmake -S . -B build/Debug \
    -DCMAKE_BUILD_TYPE=Debug

$ cmake --build build/Debug


$ cmake -S . -B build/Release \
    -DCMAKE_BUILD_TYPE=Release

$ cmake --build build/Release
```

Запуск CMake

```
cmake -S . -B build/Debug -DCMAKE_BUILD_TYPE=Debug
```

```
cmake --build build/Debug
```

- Это уже слишком многословно
- Могут быть еще параметры
- Выбор каталога делегируется пользователю

CMakePresets.json

```
.
|-- CMakeLists.txt
|-- CMakePresets.json
`-- src
    `-- main.cpp
```

```
$ cmake --preset Debug

$ cmake --build --preset Debug


$ cmake --preset Release

$ cmake --build --preset Release
```

CMakePresets.json

```
"configurePresets": [
  {
    "name": "Base",
    "generator": "Ninja",
    "binaryDir":
      "${sourceDir}/build/${presetName}"
  },
  {
    "name": "Debug",
    "inherits": "Base",
    "cacheVariables": {
      "CMAKE_BUILD_TYPE": "Debug"
...
```

```
$ cmake --preset Debug

$ cmake --build --preset Debug


$ cmake --preset Release

$ cmake --build --preset Release
```


cmake-presets(7)

Настройки запуска CMake

- `CMakePresets.json` — общие
- `CMakeUserPresets.json` — личные, добавлены в `.gitignore`

Эти файлы распознаются IDE

cmake-presets(7)

Содержание

1. Многообразие систем сборки
2. CMake: система сборки, которая ничего не собирает
3. **CMake: язык программирования, на котором не хочется писать**

Make ~> CMake

```
CXXFLAGS=-std=c++17 -O3 -Wall
LDLIBS=-lm

hello: main.cpp
  $(CXX) $(CXXFLAGS) -o $@ $^ $(LDLIBS)
```

Make ~> CMake

```
CXXFLAGS=-std=c++17 -O3 -Wall
LDLIBS=-lm

hello: main.cpp
  $(CXX) $(CXXFLAGS) -o $@ $^ $(LDLIBS)
```

```
add_executable(hello main.cpp)
```

Что такое target?

Make ~> CMake

```
CXXFLAGS=-std=c++17 -O3 -Wall
LDLIBS=-lm

hello: main.cpp
  $(CXX) $(CXXFLAGS) -o $@ $^ $(LDLIBS)
```

```
add_executable(hello main.cpp)
```

Что такое target?

Make: target == файл (кроме .PHONY)

CMake: target — объект

Make ~> CMake

```
CXXFLAGS=-std=c++17 -O3 -Wall
LDLIBS=-lm

hello: main.cpp
  $(CXX) $(CXXFLAGS) -o $@ $^ $(LDLIBS)
```

```
add_executable(hello main.cpp)
```

Наша задача:

– Создать target

– Заполнить его свойства

```
hello = {
  NAME = hello
  TYPE = EXECUTABLE
  SOURCES = main.cpp
  BINARY_DIR = ./build/
  ...
}
```

Make ~> CMake

```
CXXFLAGS=-std=c++17 -O3 -Wall
LDLIBS=-lm

hello: main.cpp
  $(CXX) $(CXXFLAGS) -o $@ $^ $(LDLIBS)
```

```
add_executable(hello main.cpp)

set_target_properties(hello
  PROPERTIES
    CXX_STANDARD 17
    CXX_STANDARD_REQUIRED ON
    CXX_EXTENSIONS OFF)
```

CXX_EXTENSIONS ON => -std=gnu++17

Make ~> CMake

```
CXXFLAGS=-std=c++17 -O3 -Wall
LDLIBS=-lm

hello: main.cpp
  $(CXX) $(CXXFLAGS) -o $@ $^ $(LDLIBS)
```

```
add_executable(hello main.cpp)

set_target_properties(hello
  PROPERTIES
    CXX_STANDARD 17
    CXX_STANDARD_REQUIRED ON
    CXX_EXTENSIONS OFF)

target_compile_options(
  hello PRIVATE -Wall)
```

Make ~> CMake

```
CXXFLAGS=-std=c++17 -O3 -Wall
LDLIBS=-lm

hello: main.cpp
  $(CXX) $(CXXFLAGS) -o $@ $^ $(LDLIBS)
```

```
add_executable(hello main.cpp)

set_target_properties(hello
  PROPERTIES
    CXX_STANDARD 17
    CXX_STANDARD_REQUIRED ON
    CXX_EXTENSIONS OFF)

target_compile_options(
  hello PRIVATE -Wall)

target_link_libraries(
  hello PRIVATE m)
```

Make ~> CMake

```
CXXFLAGS=-std=c++17 -O3 -Wall
LDLIBS=-lm

hello: main.cpp
  $(CXX) $(CXXFLAGS) -o $@ $^ $(LDLIBS)
```

```
CMAKE_BUILD_TYPE=Release
```

```
add_executable(hello main.cpp)

set_target_properties(hello
  PROPERTIES
    CXX_STANDARD 17
    CXX_STANDARD_REQUIRED ON
    CXX_EXTENSIONS OFF)

target_compile_options(
  hello PRIVATE -Wall)

target_link_libraries(
  hello PRIVATE m)
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.22)

project(Hello)

add_executable(hello main.cpp)

set_target_properties(
  hello
  PROPERTIES
    CXX_STANDARD 17
    CXX_STANDARD_REQUIRED ON
    CXX_EXTENSIONS OFF)

target_compile_options(hello PRIVATE -Wall)
target_link_libraries(hello PRIVATE m)
```

Все это можно считать созданием объекта и заполнением его свойств

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.22)

project(Hello)

add_executable(hello main.cpp)

set_target_properties(
  hello
  PROPERTIES
    CXX_STANDARD 17
    CXX_STANDARD_REQUIRED ON
    CXX_EXTENSIONS OFF)

target_compile_options(hello PRIVATE -Wall)
target_link_libraries(hello PRIVATE m)
```

Диагностики

```
if(MSVC)
  target_compile_options(hello PRIVATE /W4 /WX)
else()
  target_compile_options(
    hello
    PRIVATE
      -Wall
      -Wextra
      -Werror
      -pedantic)
endif()
```

Специфичных опций слишком много, чтобы полностью абстрагироваться от конкретного компилятора

CMake language

```
foreach(word Hello World)
  message("- ${word}")
endforeach()
```

```
function(foo arg1 arg2)
  <commands>
endfunction()
```

```
if("/a//b/c" PATH_EQUAL "/a/b/c")
   …
endif()
```

```
macro(<name> [<arg1> ...])
  <commands>
endmacro()
```


cmake-language(7)

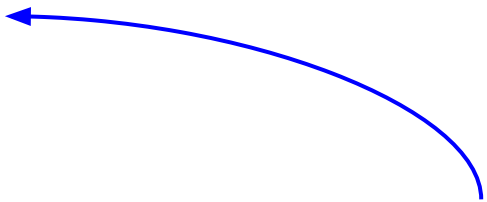
Модульность

1. Структура проекта
2. Библиотеки

Модули

- C++20 Modules все еще не поддерживаются CMake
  https://gitlab.kitware.com/cmake/cmake/-/issues/18355

- Будем структурировать проект по старинке
  - Canonical Project Structure
  - The Pitchfork Layout (PFL)



Разберем только CPS

Canonical Project Structure

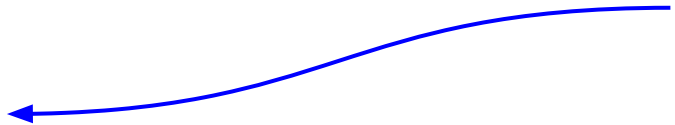
```
.
|-- CMakeLists.txt
`-- src
   |-- CMakeLists.txt
   `-- libsolver
       |-- CMakeLists.txt
       `-- libsolver
           |-- sqrt.cpp
           `-- sqrt.hpp
```

1. hpp и cpp файлы вместе

Canonical Project Structure

```
.
|-- CMakeLists.txt
`-- src
   |-- CMakeLists.txt
   `-- libsolver
       |-- CMakeLists.txt
       `-- libsolver
           |-- sqrt.cpp
           `-- sqrt.hpp
```

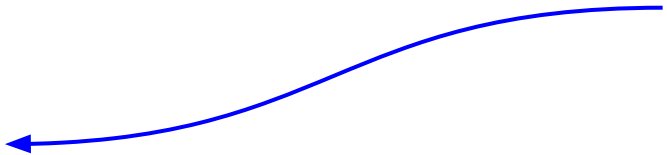
1. hpp и cpp файлы вместе
2. `g++ -I src/libsolver`



Canonical Project Structure

```
.
|-- CMakeLists.txt
`-- src
   |-- CMakeLists.txt
   `-- libsolver
       |-- CMakeLists.txt
       `-- libsolver
           |-- sqrt.cpp
           `-- sqrt.hpp
```

1. hpp и cpp файлы вместе
2. `g++ -I src/libsolver`
3. `#include <libsolver/sqrt.hpp>`



src/libsolver/CMakeLists.txt

```
set(target_name solver)

add_library(${target_name} STATIC
  libsolver/sqrt.hpp libsolver/sqrt.cpp)

include(CompileOptions)
set_compile_options(${target_name})

target_include_directories(${target_name}
  PUBLIC
    ${CMAKE_CURRENT_LIST_DIR})

target_link_libraries(${target_name} PRIVATE m)
```

PRIVATE, INTERFACE, PUBLIC

PRIVATE — элемент используется только таргетом

INTERFACE — элемент используется только клиентами таргета

PUBLIC — элемент используется и таргетом, и его клиентами

Пример: PUBLIC

```
target_include_directories(solver PUBLIC src/libsolver)
target_link_libraries(app PRIVATE solver)
```

```
g++ -o libsolver.so -I src/libsolver ...
g++ -o app -I src/libsolver ...
```

Пример: INTERFACE

✗

```
target_include_directories(solver INTERFACE src/libsolver)
target_link_libraries(app PRIVATE solver)
```

```
g++ -o libsolver.so ...
g++ -o app -I src/libsolver ...
```

Типичный сценарий для INTERFACE – библиотеки шаблонов

Пример: PRIVATE

✗

```
target_include_directories(solver PRIVATE src/libsolver)
target_link_libraries(app PRIVATE solver)
```

```
g++ -o libsolver.so -I src/libsolver ...
g++ -o app ...
```

Типы библиотек

1. Shared Library (Dynamic Library, Shared Object)
2. Static Library
3. Object Library (CMake-specific)
4. Header-only Library (C++-specific)

Сравнение типов библиотек

	Shared	Static	Object	Header-only
CMake-тип	SHARED	STATIC	OBJECT	INTERFACE
Именование	lib*.so	lib*.a	—	*.hpp
Формат	ELF	ar-архив	Список файлов в CMake	Код C++

https://wiki.debian.org/StaticLinking

NB: Shared vs Static deps

```
add_library(solver SHARED …)
target_link_libraries(solver PRIVATE m)
target_link_libraries(app PRIVATE solver)

g++ -o libsolver.so -lm
g++ -o app -lsolver
```

NB: Shared vs Static deps

```
add_library(solver STATIC …)
target_link_libraries(solver PRIVATE m)
target_link_libraries(app PRIVATE solver)

ar cr libsolver.a *.o
g++ -o app -lsolver -lm
```

Использование сторонних библиотек

```
.
`-- external
   |-- CMakeLists.txt
   `-- cxxopts
        `-- ???
```

```
add_subdirectory(cxxopts)
```

Использование сторонних библиотек

```
.
`-- external
   |-- CMakeLists.txt
   `-- cxxopts
        `-- ???
```

???:

- Исходники сторонней библиотеки
- Git submodule
- CMakeLists.txt

Вендоринг исходников

✓ Надежно. Remember the left-pad!

✗ Утяжеление репозитория
✗ Сложность обновления библиотек

Git Submodules

✓ Репозиторий остается легковесным
✓ Простое обновление

? Субъективно неудобно

Git Tools - Submodules

FetchContent

```
include(FetchContent)

FetchContent_Declare(
  cxxopts
  GIT_REPOSITORY https://github.com/jarro2783/cxxopts.git
  GIT_TAG v2.2.1
  GIT_SHALLOW TRUE
  PREFIX ${CMAKE_CURRENT_BINARY_DIR})

FetchContent_MakeAvailable(cxxopts)
```

Источники

- Примеры: csc-cpp/cpp-examples/01-cmake
- CMake:
  - CMake Tutorial
  - Modern CMake
- Структура проекта:
  - Canonical Project Structure
  - The Pitchfork Layout (PFL)
- GoogleTest Quickstart
- ctest(1)
- https://wiki.debian.org/StaticLinking