Этапы компиляции

Исходный код, например,

main.cpp:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, world!";
    return 0;
}
```

Проходит через определённый конвейер и на выходе получается исполняемый файл

Директивы препроцессора

#define

#elif

#else

#endif

#error

#if

#ifdef

#ifndef

#import

#include

#line

#pragma

#undef

#using

Первый этап компиляции, препроцессинг: g++ -E main.cpp > output.cpp

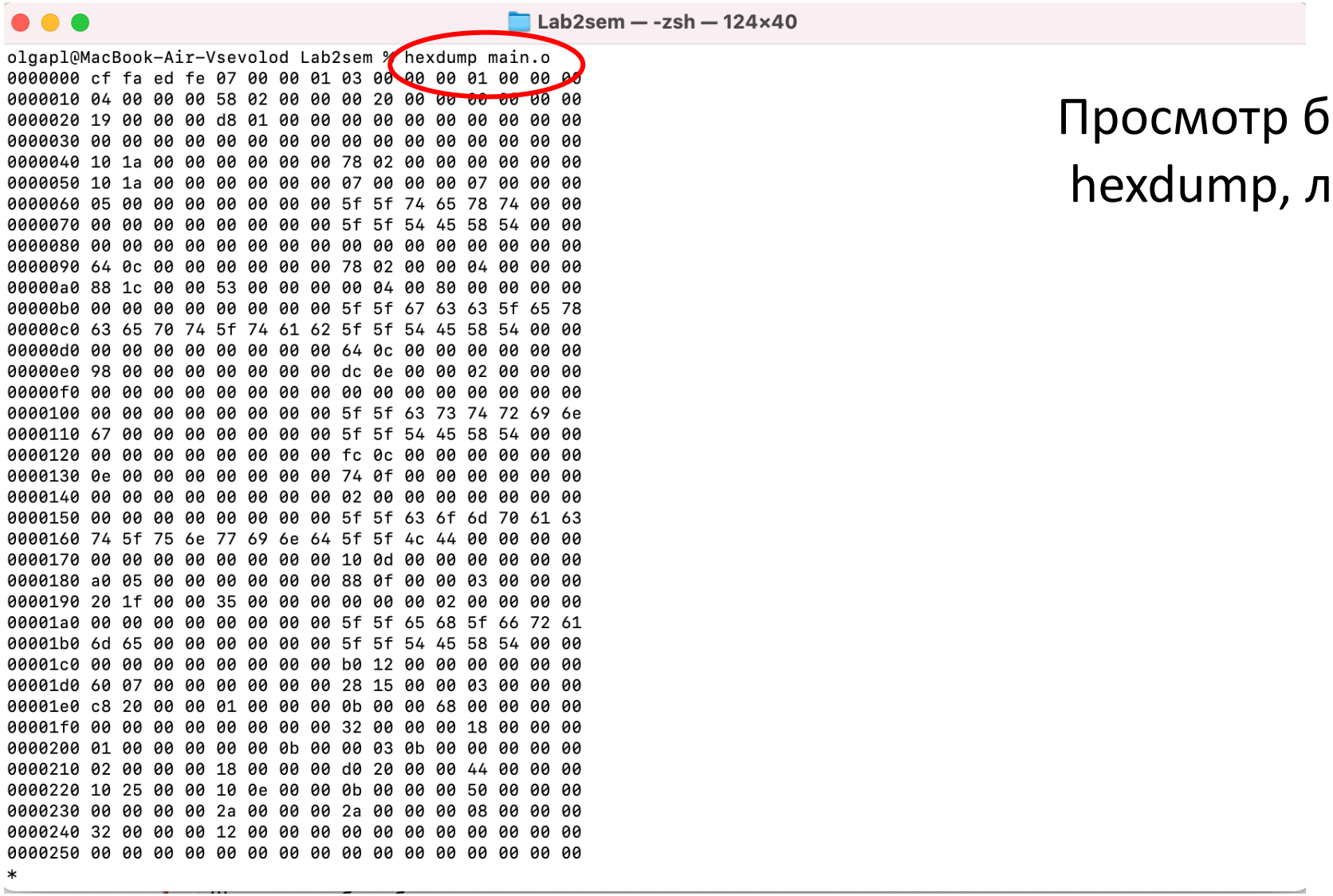
Компиляция
g++ -S main.cpp

Сайт, чтобы посмотреть, какая команда C++ соответствует какой ассемблерной: godbolt.org

Lab2sem — vim main.s — 147×37

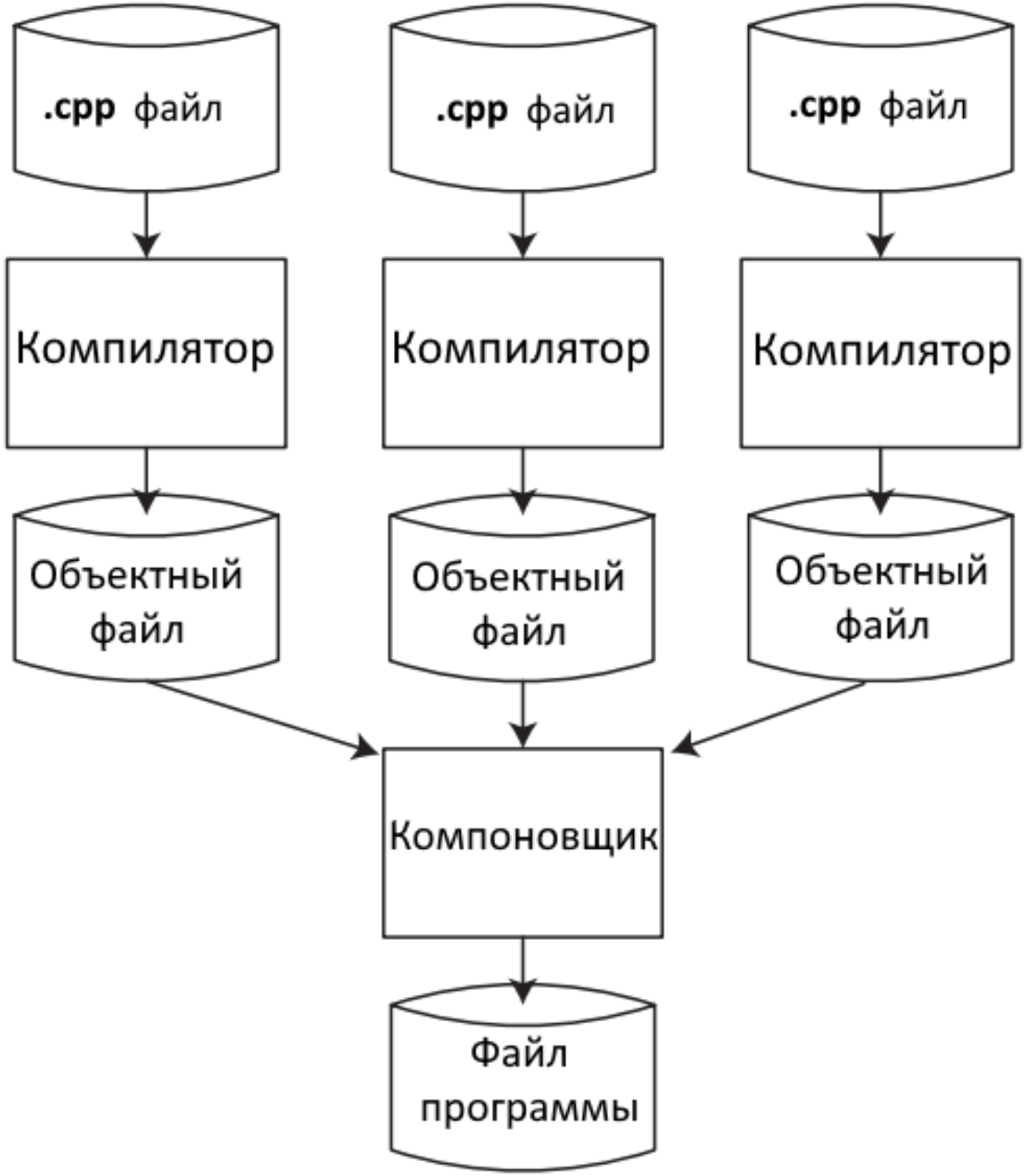
```
        je      LBB5_16
## %bb.12:
Ltmp26:
        callq   __ZNSt3__1L15__get_nullptr_tEv
        movq    %rax, %rcx
Ltmp27:
        movq    %rcx, -200(%rbp)                ## 8-byte Spill
        jmp     LBB5_13
LBB5_13:
        movq    -200(%rbp), %rax                ## 8-byte Reload
        movq    %rax, -144(%rbp)
Ltmp28:
        leaq    -144(%rbp), %rdi
        callq   __ZNKSt3__19nullptr_tcvPT_INS_15basic_streambufIcNS_11char_traitsIcEEEEEEv
        movq    %rax, %rcx
Ltmp29:
        movq    %rcx, -208(%rbp)                ## 8-byte Spill
        jmp     LBB5_14
LBB5_14:
        movq    -208(%rbp), %rax                ## 8-byte Reload
        movq    %rax, -16(%rbp)
        movq    -16(%rbp), %rax
        movq    %rax, -8(%rbp)
        movl    $1, -148(%rbp)
        jmp     LBB5_17
LBB5_15:
Ltmp30:
        movq    %rdx, %rcx
        movq    %rax, %rsi
                                        ## kill: def $ecx killed $ecx killed $rcx
        movq    %rsi, -128(%rbp)
        movl    %ecx, -132(%rbp)
Ltmp31:
        leaq    -120(%rbp), %rdi
        callq   __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEED1Ev
Ltmp32:
"main.s" line 481 of 1528 --31%-- col 7
```

Сборка
g++ -c main.cpp



Просмотр бинарного файла:
hexdump, либо nm

Линковка
g++ -o main main.o (либо main.cpp)



Компиляция — алгоритмически сложный процесс, для больших программных проектов требующий существенного времени и вычислительных возможностей.

Благодаря наличию в процессе сборки программы этапа компоновки (связывания) возникает возможность *раздельной компиляции*.

В модульном подходе программный код разбивается на несколько файлов .cpp, каждый из которых компилируется отдельно от остальных.

Это позволяет значительно уменьшить время перекомпиляции при изменениях, вносимых лишь в небольшое количество исходных файлов. Также это даёт возможность замены отдельных компонентов конечного программного продукта, без необходимости пересборки всего проекта.

Процедурная и объектная декомпозиция

- **Процедурной декомпозицией** называют представление разрабатываемой программы в виде совокупности вызывающих друг друга подпрограмм. Каждая подпрограмма в этом случае выполняет некоторую операцию, а вся совокупность подпрограмм решает поставленную задачу.

- **Объектной декомпозицией** называют процесс представления предметной области задачи в виде совокупности функциональных элементов – объектов, обменивающихся в процессе выполнения программы входными воздействиями – сообщениями.

Что такое библиотека и какая она бывает?

Библиотека – это пакет кода, который предназначен для повторного использования многими программами. Обычно библиотека C++ состоит из двух частей:

- *заголовочный файл*, который определяет функциональность, которую библиотека предоставляет (предлагает) программам, использующим ее;
- *предварительно скомпилированный двоичный файл*, который содержит реализацию этой функциональности, предварительно скомпилированную в машинный код.

Существует два типа библиотек:

- **статические** библиотеки;
- **динамические** библиотека

Статическая библиотека (иногда называемая *archive*, "архив") состоит из подпрограмм, которые скомпилированы и линкуются непосредственно с вашей программой.

Когда вы компилируете программу, использующую статическую библиотеку, все функции статической библиотеки, которые использует ваша программа, становятся частью вашего исполняемого файла.

В Windows статические библиотеки обычно имеют расширение **.lib** (*library*, библиотека), а в Linux – расширение **.a** (*archive*, архив).

Динамическая библиотека (также называемая shared library, "общая библиотека") состоит из подпрограмм, которые загружаются в ваше приложение во время выполнения.

Когда вы компилируете программу, использующую динамическую библиотеку, библиотека не становится частью вашего исполняемого файла – она остается отдельной единицей.

В Windows динамические библиотеки обычно имеют расширение **.dll** (*dynamic link library*, библиотека динамической компоновки), а в Linux – расширение **.so** (*shared object*, общий объект).

Header-only библиотека

Header-only библиотеки — это библиотеки, которые полностью состоят из заголовочных файлов. У них **нет файлов реализации** (.cpp), которые нужно компилировать отдельно.

Преимущества:

1. Простота
2. Кроссплатформеность
3. Лёгкость интеграции

Недостатки:

1. Время компиляции
2. Размер исполняемого файла
3. Сложность скрытия реализации

Примеры header-only

Математика

```
#include <glm/glm.hpp>
```

Логирование

```
#include "spdlog/spdlog.h”
```

Тестирование

```
#include "doctest.h"
```

Статические библиотеки

Преимущества

- **Полная автономность (портативность):** Исполняемый файл содержит всё необходимое для работы. Его можно скопировать на любую совместимую систему, и он запустится, независимо от наличия там нужных библиотек.
- **Производительность:** Поскольку весь код находится в одном исполняемом файле, вызов функций происходит немного быстрее (нет накладных расходов на поиск и загрузку библиотеки в память во время выполнения).

Недостатки

- **Большой размер исполняемого файла:** Если несколько программ используют одну и ту же статическую библиотеку, её код будет полностью продублирован в каждом исполняемом файле. Это увеличивает занимаемое место на диске и в оперативной памяти при одновременном запуске этих программ.
- **Обновления библиотек:** Чтобы обновить библиотеку, необходимо **перекомпилировать каждую программу**, которая её использует. Это трудоёмко для разработчиков и неудобно для пользователей.

Динамические библиотеки

Преимущества

1. **Экономия дискового пространства и памяти:** Один файл библиотеки на диске может использоваться множеством программ. При запуске этих программ ОС загружает код библиотеки в оперативную память один раз.
2. **Лёгкость обновлений:** Чтобы обновить библиотеку, достаточно заменить один файл библиотеки в системе. Все программы, которые её используют, автоматически начнут использовать новую версию после перезапуска.

Недостатки

1. **Проблемы совместимости:** Если библиотека обновлена до новой версии, старая программа, которая рассчитывала на старый интерфейс, может не работать. И если в системе нет нужной версии библиотеки, программа тоже не запустится.
2. **Сложность распространения:** Нужно либо убедиться, что нужная библиотека уже есть в целевой системе, либо поставлять её вместе с программой.

Создание статической библиотеки

Чтобы собрать статическую библиотеку, нужен *.cpp и *.h

Статическая библиотека — это просто архив объектных файлов (.o**/**.obj**), а не готовый к выполнению код.** Её не нужно "линковать" при создании, потому что она ещё не знает, с какой программой будет использоваться.

Создание объектного файла:

```
g++ -c <name>.cpp
```

Упаковка в статическую библиотеку:

```
ar rcs <libname>.a <objectName>.o
```

Использование статической библиотеки

Для использования библиотеки нужно два файла:

<libname>.a

<name>.h, который подключается к исполняемому файлу

В зависимости от того, где располагается *.a:

```
g++ <main>.cpp  <libname>.a -o <projectname>
```

//если в корне

```
g++ <main>.cpp -L./lib -<lname> -o <projectname>
```

// -L. искать библиотеки в текущей папке

// -L./lib искать в папке lib

// -lmylib подключить libmylib.a (префикс lib и суффикс .a добавляются автоматически)

Утилита make

Утилита make предназначена для автоматизации преобразования файлов из одной формы в другую. По отметкам времени каждого из имеющихся объектных файлов (при их наличии) она может определить, требуется ли их пересборка.

Правила преобразования задаются в скрипте с именем Makefile, который должен находиться в корне рабочей директории проекта. Сам скрипт состоит из набора правил, которые в свою очередь описываются:

1. **целями** (то, что данное правило делает);
2. **реквизитами** (то, что необходимо для выполнения правила и получения целей);
3. **командами** (выполняющими данные преобразования).

Makefile

В общем виде синтаксис Makefile можно представить так:

```
# Отступ (indent) делают только при помощи символов табуляции,
# каждой команде должен предшествовать отступ
<цели>  :       <реквизиты>
                <команда #1>
                ...
                <команда #n>
```

То есть правило make – это ответы на три вопроса:

{Из чего делаем? (реквизиты)} ---> [Как делаем? (команды)] ---> {Что делаем? (цели)}

Несложно заметить что процессы трансляции и компиляции очень красиво ложатся на эту схему:

{исходные файлы} ---> [трансляция] ---> {объектные файлы}

{объектные файлы} ---> [линковка] ---> {исполнимые файлы}

Пример Makefile

Для компиляции main.cpp достаточно очень простого Makefile:

```
main: main.cpp
	g++ -c main.cpp
	g++ -o main main.o
```

Данный Makefile состоит из одного правила, которое, в свою очередь состоит из цели — main, реквизита — main.cpp, и последовательности команд.

Теперь, для компиляции достаточно дать команду make в рабочем каталоге. По умолчанию make станет выполнять самое первое правило, если цель выполнения не была явно указана при вызове:

```
$ make <цель>
```

Makefile для модульной программы

```
program:        program.o mylib.o
                g++ -o program program.o mylib.o
program.o:      program.cpp mylib.hpp
                g++ -c program.cpp
mylib.o:        mylib.cpp mylib.hpp
                g++ -c hylib.cpp
```

После запуска make попытается сразу получить цель program, но для ее создания необходимы файлы program.o и mylib.o, которых пока еще нет. Поэтому выполнение правила будет отложено и make станет искать правила, описывающие получение недостающих реквизитов. Как только все реквизиты будут получены, make вернется к выполнению отложенной цели. Отсюда следует, что make выполняет правила рекурсивно.

Источник отсюда

CMake

Кроссплатформенное программное средство автоматизации сборки программного обеспечения из исходного кода.

Не занимается непосредственно сборкой, а лишь генерирует файлы сборки из предварительно написанного файла сценария CMakeLists.txt и предоставляет простой единый интерфейс управления.
Помимо этого, способно автоматизировать процесс **установки** и **сборки пакетов.**

Привет, мир!

Если вы создаете файл hello.txt со следующим содержимым:

```
message("Hello world!")                    # A message to print
```

... вы можете запустить его из командной строки с помощью `cmake -P hello.txt`. -P Опция запускает данный скрипт, но не создает конвейер сборки.

```
$ cmake -P hello.txt
Hello world!
```

В CMake каждая переменная представляет собой строку. Вы можете заменить переменную внутри строкового литерала, окружив ее ${}.

Чтобы определить переменную внутри скрипта, используйте set команду. Первый аргумент – это имя присваиваемой переменной, а второй аргумент - ее значение

```
$ cat hello.txt
message("Hello ${NAME}!")   # Substitute a variable into the message
$ cmake -P hello.txt
Hello !

$ cat hello.txt
set(THING "funk")
message("We want the ${THING}!")

$ cmake -P hello.txt
We want the funk!
```

Имитация структуры данных, используя префиксы

```
$ cat hello.txt
set(JOHN_NAME "John Smith")
set(JOHN_ADDRESS "123 Fake St")
set(PERSON "JOHN")
message("${${PERSON}_NAME} lives at
${${PERSON}_ADDRESS}.")

$ cmake -P hello.txt
John Smith lives at 123 Fake St.
```

В CMake каждый оператор представляет собой **команду**, которая принимает список строковых аргументов и **не имеет возвращаемого значения**. Аргументы разделяются пробелами (без кавычек). Как мы уже видели, set команда определяет переменную в области видимости файла (можно прокидывать переменные "выше" при сборке)

```
$ cat hello.txt
math(EXPR MY_SUM "1 + 1") # Evaluate 1 + 1; store result in MY_SUM
message("The sum is ${MY_SUM}.")
math(EXPR DOUBLE_SUM "${MY_SUM} * 2") # Multiply by 2; store result in DOUBLE_SUM
message("Double that is ${DOUBLE_SUM}.")

$ cmake -P hello.txt
The sum is 2.
Double that is 4
```

Команды управления потоком

Даже операторы управления потоком являются командами. Команды if/endif выполняют вложенные команды условно. Пробелы не имеют значения, но обычно для удобства чтения во вложенных командах делается отступ.

Ниже проверяется, установлена ли встроенная переменная CMake WIN32:

```
If (WIN32)
message "You're running CMake on Windows."
endif
```

В CMake также есть while/endwhile команды

В CMake есть специальное правило подстановки аргументов без кавычек.

- Если весь аргумент представляет собой ссылку на переменную без кавычек, а значение переменной содержит точки с запятой, CMake **разделит значение на точки с запятой** и передаст **несколько аргументов заключающей команде**.

```
set(ARGS "EXPR;T;1 + 1")
math(${ARGS}) # Equivalent to calling math(EXPR T "1 + 1")
message("${T}") # Prints: 2;
```

- С другой стороны, **аргументы в кавычках никогда не разбиваются на несколько аргументов**, даже после замены. CMake всегда передает строку в кавычках в качестве одного аргумента, оставляя точки с запятой нетронутыми:

```
set(ARGS "EXPR;T;1 + 1")
message("${ARGS}") # Prints: EXPR;T;1 + 1
```

CMake в проекте

Проекты, использующие **CMake**, содержат один или несколько конфиг файлов, которые всегда имеют одно и то же название: **CMakeLists.txt**. Внутри одной директории находится только один конфиг файл. Самый основной находится прямо в корне проекта. При необходимости (когда нужно делить проект на отдельные модули, где описывается своя логика сборки) заводятся дополнительные конфиг файлы во вложенных директориях.

Пример CMakeLists.txt

```
# Минимальная версия CMake
cmake_minimum_required(VERSION 3.10)
# Название проекта
project(MyProgram)
# Указываем, что используем C++17
set(CMAKE_CXX_STANDARD 17)
# Создаём исполняемый файл "MyProgram"
# Указываем все .cpp файлы, которые нужны
add_executable(MyProgram
        main.cpp # Главный файл
        calculator.cpp # Файл с функциями калькулятора
        utils.cpp # Вспомогательные функции
)
# Говорим, где искать .h файлы (заголовки)
target_include_directories(MyProgram
        . # Текущая папка
        include # Папка include, если она есть
)
# Если нужно подключить библиотеки (например, math)
target_link_libraries(MyProgram math) # math - математическая библиотека
```

Пример CMakeLists.txt создания статической библиотеки

```
# Минимальная версия CMake
cmake_minimum_required(VERSION 3.10)

# Название библиотеки
project(MathLib)

# Создаём статическую библиотеку
# STATIC – значит, библиотека будет статической
add_library(MathLib STATIC
            mathlib.cpp # Файл с кодом библиотеки
)

# Говорим, где искать .h файлы
# PUBLIC – значит, эти пути будут видны и тем, кто использует библиотеку
target_include_directories(MathLib PUBLIC
       ${CMAKE_CURRENT_SOURCE_DIR}/include # Папка include
)
```

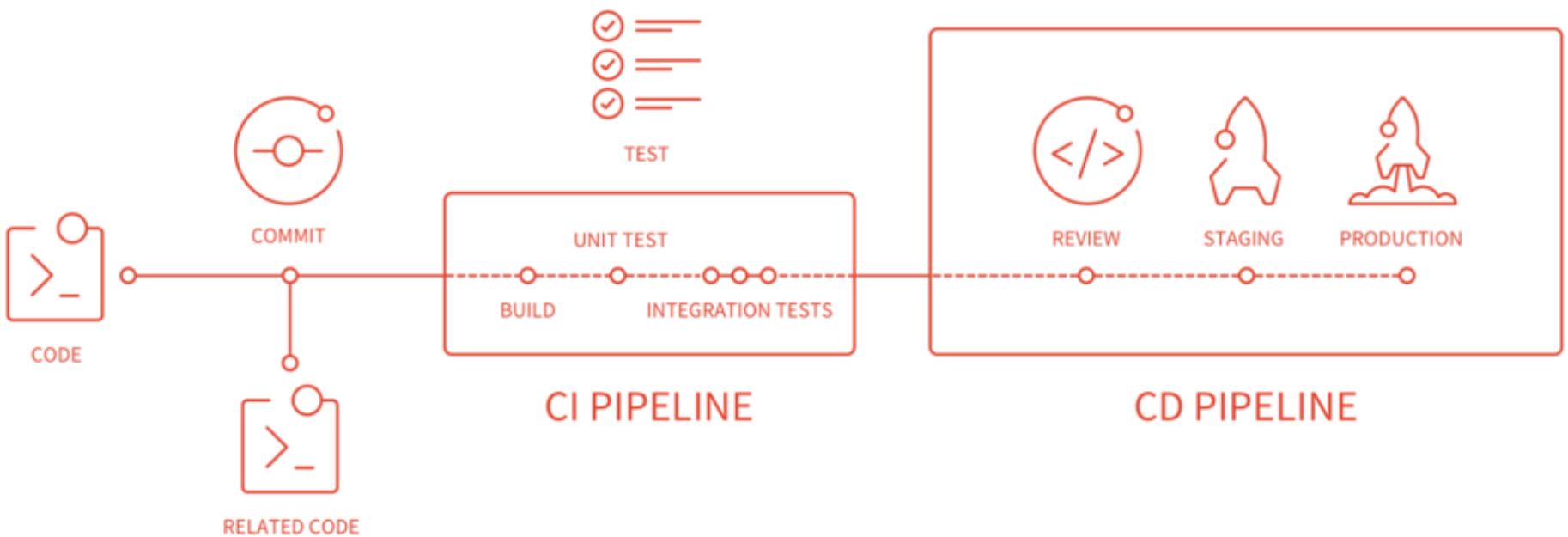
Пример CMakeLists.txt создания статической библиотеки

```
cmake_minimum_required(VERSION 3.23)

set(project "mymath")
project(${project})

set(CMAKE_CXX_STANDARD 17)

set(${project}_SOURCES
       mymath.cpp)

set(${project}_HEADERS
       mymath.h)

set(${project}_SOURCE_LIST
${${project}_SOURCES}
${${project}_HEADERS})

add_library(${project}
       STATIC
       ${${project}_SOURCE_LIST})
```

Современный софт собирают на конвейерах CI/CD.

Конвейер CI/CD — это система управления доставкой программного обеспечения для создания, тестирования и предоставления программного обеспечения как услуги.

GitLab CI


TEST
COMMIT
UNIT TEST
REVIEW
STAGING
PRODUCTION
BUILD
INTEGRATION TESTS
CODE
RELATED CODE
CI PIPELINE
CD PIPELINE