СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Опорный конспект по теме №1

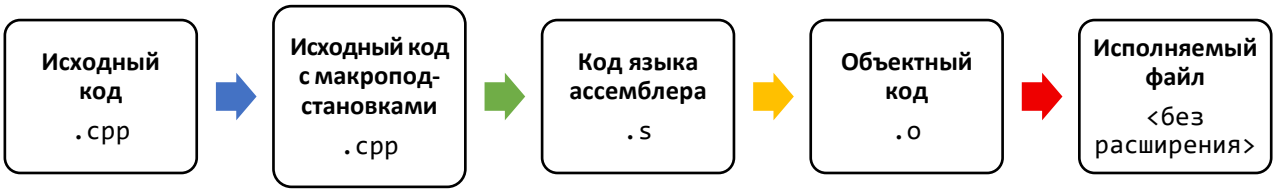
«Система сборки CMake»

1. Вспоминаем процесс создания исполняемого файла

Начнем с того, что для того, чтобы ваша программа вообще была преобразована из исходного кода в исполняемый файл, код должен преодолеть несколько этапов преобразования в разные форматы данных.

Примечание: в данном курсе мы будем пользоваться операционными системами на базе ядра Linux (в классе в основном используем Debian 13), и, следовательно, все примеры и утилиты командной строки рассматриваются в рамках данных систем.

Рассмотрим стандартный путь перемещения описания нашего файла в случае компиляции исходного кода на языке C++ с использованием GNU Compiler Collection (GCC):



Каждый переход, обозначенный стрелкой, имеет свое наименование:

- Преобразование исходного кода с исходный код с макроподстановками называется «*препроцессорная обработка*». За данный этап отвечает специальный компонент компилятора языка, называемый препроцессором (в примере ниже GNU C Preprocessor как часть компилятора GCC).

  Может быть выполнен командой `g++ -E -o {выход}.cpp {вход}.cpp`
- Преобразование исходного кода с макроподстановками в код ассемблера называется ассемблированием (компиляцией в код ассемблера). За этот этап отвечает компилятор конкретного языка (в примере ниже — G++ из GCC).

  Может быть выполнен командой `g++ -S -o {выход}.s {вход}.cpp`
- Преобразование кода ассемблера в объектный код называется компиляцией в объектный код. За этот этап отвечает компилятор языка ассемблера (в примере ниже — GNU Assembler, a.k.a. GAS)

  Может быть выполнен командой `as -o {выход}.o {вход}.s`
- Преобразование объектного кода в исполняемый файл называется линковкой или компоновкой. За этот этап отвечает программа-компоновщик (в примере ниже — GNU Linker, a.k.a. LD).

  Может быть выполнен командой `ld -o {выход} {вход}.o`

Можно заметить, что данные этапы используют разные утилиты для выполнения текущих преобразований. Более того, последовательность преобразований в таком случае является достаточно утомительной.
Именно поэтому в современном мире (да и не очень современном) программисты придумали несколько специальных систем, которые могут облегчить процесс сборки, и назвали их системами сборки.

2. Системы сборки

Традиционной системой сборки для ОС Unix является *Unix Makefiles*, которая, по сути, представляет собой файл с именем Makefile, содержащий набор описанных текстом целей (т.н. таргеты, от англ. *target* — цель), каждая из которых может иметь зависимость от других целей. Цели создаются путем выполнения набора команд текущей оболочки (как правило, Bourne Shell), необходимых для преобразования зависимостей в результирующий файл, как раз и являющийся целью.
Так, процесс создания исполняемого файла можно написать один раз и использовать одну команду

```
make {target}
```

для сборки всего проекта. При добавлении целей команда не изменяется, однако может в качестве аргумента принимать новые описанные цели.

Unix Makefiles является примитивной, но универсальной системой, которая может работать с любыми языками программирования: достаточно лишь указать используемые утилиты в виде команд. Однако, из-за своей примитивности, Makefile тяжело использовать в крупных проектах — в таком случае целей появляется настолько много, что дальнейшее изменение Makefile делает его нечитаемым из-за огромного размера.

В данной теме будет рассмотрена отдельная более удобная система сборки, используемая конкретно с языками C/C++ — CMake.

3. Система сборки CMake

CMake представляет собой кроссплатформенную систему сборки проектов для языков C/C++. CMake также предоставляет текстовый формат описания процесса создания исполняемых файлов на основе целей, однако цели формируются более абстрактно — система сборки сама определяет компилятор, компоновщик, их версию и версию языка (если не указано иначе), а также *генератор для создания файлов сборки* (по умолчанию используется как раз Unix Makefiles), чтобы не пришлось вводить команды для непосредственной компиляции.

Файл системы сборки CMake по умолчанию должен называться **CMakeLists.txt**. Для сборки проекта всегда должен существовать коренной CMakeLists.txt с глобальными настройками всего проекта. Такой файл имеет следующий вид:

```
cmake_minimum_required(VERSION 3.12)
project(example)
…
```

В данном случае используются следующие функции:

- `cmake_minimum_required` — функция, определяющая минимальную версию системы сборки CMake, при которой данный проект может быть гарантированно удачно собран. Версия указывается в скобках после ключевого слова VERSION в формате Semantic Versioning (`{major}.{minor}[.{patch}]`).
- `project` — функция, задающая название всему проекту. В файлах CMake любая последовательность символов трактуется как строка, однако в случае, когда нужная строка содержит пробелы, ее можно заключать в двойные кавычки:

```
project(“New project”)
```

После данного предварительного описания могут быть использованы и другие функции, которые непосредственно взаимодействуют с проектом:

- `add_executable({имя} {…})` — объявление цели на создание исполняемого файла с заданным именем, использующего перечисленные файлы исходного кода .cpp.
- `set(CMAKE_CXX_STANDART {N})` — установить используемый стандарт языка C++ (в качестве {N} выступает одно из чисел 11, 14, 17, 20, 23, обозначающие год выхода стандарта — по сути, версия языка). Аналог опции компилятора `-std=…`.
- `set(CMAKE_CXX_STANDART_REQUIRED ON)` — установить номер используемого стандарта как требование, без которого этот сборка проекта невозможна.
- `set(CMAKE_CXX_COMPILER {…})` — установить путь до используемого компилятора.
- `set(CMAKE_BUILD_TYPE {Debug|Release|TestRelease})` — установка типа сборки (отладочная, релизная, тест-релизная). Аналог ключа `-g`.
- `add_compile_options({…})` — добавить опции компилятора, разделенные пробелом
- `add_compile_definitions({…})` — добавить препроцессорные определения, используемые препроцессором. Аналог препроцессорной директивы `#define`.
- `include_directories([AFTER|BEFORE] {…})` — добавить пути включаемых каталогов (AFTER означает добавление каталога в конец очереди рассмотрения каталогов, BEFORE — добавление в начало очереди). Включаемые каталоги используются препроцессорной директивой #include, заголовочный файл, написанный в треугольных скобках или двойных кавычках может быть найден среди этих каталогов без указания полного пути. Аналог ключа `-I`.
- `link_directories([AFTER|BEFORE] {…})` — добавить пути библиотек. Так компоновщик сможет искать перечисленные библиотеки не только в системных каталогах, но и в указанных здесь. Аналог ключа `-L`.
- `target_link_libraries({имя} {…})` — скомпоновать цель с заданным именем с перечисленными библиотеками. Аналог ключа `-l`. *Замечание*: если указывается библиотека с именем `name`, компоновщик попытается найти библиотеку с именем `libname.a` или `libname.so` с возможным указанием редакции (например, libname.so.1).

- `add_subdirectory({…})` — указать каталог, в котором находится подпроект данного проекта. В каталоге также должен быть определен файл CMakeLists.txt, команды из которого будут также выполнены как часть данной команды.

В CMakeLists.txt также могут присутствовать комментарии, которые не обрабатываются системой сборки. Комментарии начинаются со знака решетки #.

4. Пример файла и процесс сборки проекта

Пример файла CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.12)  # C++20 работает с версии 3.12

project(sample_project)

set(CMAKE_CXX_STANDART 20)
set(CMAKE_CXX_STANDART_REQUIRED ON)
set(CMAKE_BUILD_TYPE Debug)           # Отладочная версия

include_directories(BEFORE
    include
)

add_compile_options(-Wall -Wextra)    # Обрабатываем предупреждения как ошибки

# На всякий случай, если CMAKE_CXX_STANDART не заработает
add_compile_options(-std=c++20)

add_compile_definitions(_NDEBUG)      # Препроцессорное определение для отладки

add_executable(example
    main.cpp
    database.cpp
)

target_link_libraries(example         # Ищем библиотеку libsqlite3.so
    sqlite3
)
```

Для создания необходимых файлов сборки используется команда

`cmake -S {путь_до_исходных_кодов} -B {путь_до_сборки}`

где `{путь_до_исходных_кодов}` — путь до каталога, внутри которого лежат исходные коды (файлы .cpp), `{путь_до_сборки}` — каталог, в котором необходимо разместить файлы сборки.

Для сборки используется команда

`make -C {путь_до_сборки} [цель]`

Цель `{target}` может быть не указана: в таком случае выполнится цель по умолчанию all, которая соберет весь проект. Результат будет размещен в каталоге `{путь_до_сборки}`.

5. Дополнительная информации

Для получения дополнительной информации воспользуйтесь официальной документацией по системе сборки CMake: https://cmake.org/documentation/.