Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б10-мм

Автоматизированная конвертация из систем сборки в CMake для языков C/C++

Лоскутов Владислав Евгеньевич

Отчёт по учебной практике
в форме
«Производственное
задание»

Научный
руководитель:старший преподаватель кафедры ИАС, Смирнов К.К.

Консультант
старший разработчик, ООО «Софтком», Бабанов П. А.

Санкт-Петербург
2024

Введение

Современное программное обеспечение часто разрабатывается и собирается с использованием различных систем сборки [1]. Без них трудно представить создание продукта с множеством различных компонентов.

Системы сборки описывают логику при компиляции проекта с помощью исходных файлов [2], что облегчает интегрируемость при работе над проектом с разных компьютеров.

CMake — это инструмент автоматизации сборки проектов, который облегчает процесс управления кроссплатформенной разработкой [3]. При этом сам CMake не является сборочной системой, а использует данные в системе, имея транслятор из своего языка описания проектов в широкий спектр других. В этом его преимущество.

Но при разработке программ могут использоваться множество других систем сборки, логика работы которых отличается между собой. Make – одна из них.

В сообществе разработчиков часто появляются вопросы[1] о конвертации файлов Makefile в файлы CMakeLists. К тому же, на самом официальном сайте, посвящённом CMake, в статье о конвертации ничего не сказано об автоматическом конвертере[2]. Компания ООО «Софтком» стокнулась с такой проблемой, которая возникает из-за необходимости актуализировать legacy-проекты, которые используют make.

В данной работе требуется изучить общие принципов работы сборочных систем, а также создать программа для конвертации конфигурационных файлов make в файлы для системы сборки CMake.

---

[1] https://stackoverflow.com/questions/40860769/convert-makefile-into-cmakelists-where-to-start

[2] https://cmake.org/cmake/help/book/mastering-cmake/chapter/Converting%20Existing%20Systems%20To%20CMake.html

1. Постановка задачи

Целью работы является разработка программы для конвертации конфигурационных файлов проекта в файлы для системы сборки CMake. Для достижения цели были поставлены следующие задачи:

1. Исследовать логику работы конвертеров систем сборки
2. На основе существующих решений построить логику работы собственного конвертера в CMake.
3. Написать собственное приложение для конвертации make в CMake

2. Обзор

2.1 Обзор существующих решений

Обзор был написан, исходя из следующих критериев:

- Актуальность конвертируемой системы сборки;
- Работоспособность конвертера в ситуациях, когда исходные файлы генерируются в процессе сборки;
- Поддержка условной компиляции;
- Работа с многофайловой сборкой проекта;
- Возможность конвертации проектов, написанных на языке C/C++;

2.1.1 ProjectConverter[3]

ProjectConverter представляет из себя проект с открытым исходным кодом, позволяющий конвертировать проекты из систем сборки KEIL или IAR в CMake. ProjectConverter конвертирует файлы, написаные на языке Python. Программа использует самописный парсер и поддерживает такие команды, как инициализация переменных, генерацию таргетов и линковку бибиотек к таргетам.

2.1.2 Qmake2cmake[4]

QMake2cmake также является open-souce-проектом для конвертации систем сборок. Данный проект позволяет конвертировать систему сборки QMake в CMake. Языком программирования, для которых работает конвертер, является C++. Принцип работы заключается в поиске команд по ключевым словам и символам и последующей их конвертации в CMake. Программа также использует

[3] https://github.com/phodina/ProjectConverter
[4] https://github.com/diegostamigni/qmake2cmake

библиотеку Qtдля создания пользовательского интерфейса.

2.1.3 AutoCMake[5]

Automake является конвертером из системы сборки autotools в CMake. Поддерживаемый язык программирования - Python. Поиск и работа с исходными файлами осуществляется в помощью поиска по формату файла (для библиотек - .h, .hpp, .hxx, .h++, .hh; для исходных файлов - .c, .cpp, .cxx, .c++, .cc). Программа определяет инициализированные переменные по знаку “=” .

2.2 Обзор используемых технологий

2.2.1 C++

C++ используется как язык для написания программного кода для конвертера.

2.2.2 CMake

В приложении для конвертации системы сборки make в CMake как раз используется CMake для удобной сборки исходных файлов проекта.

2.2.3 Boost

Библиотека boost для C++ используется в проекте для удобной работы со строками и файлами.

2.2.4 stl

stl является классической библиотекой C++, в текущей работе stl используется для хранения и управления данными в процессе работы программы.

2.3 Выводы

Как можно видеть, попытки написания конвертера для CMake предпринимались различными разработчиками. Однако решения, которое позволяло бы конвертировать систему сборки make в CMake не существует. В следствии, возникает потребность в создании

[5] https://github.com/fritzone/autocmake

собственного конвертера.

3. Описание решения

3.1 Сравнение Makefile и CMakeLists

make и CMake делают одну и ту же вещь - вызывают различные инструменты (компилятор, линкер, скрипт) с определенным набором аргументов в определенной последовательности[6]. Но принципы их работы значительно отличаются. Так, make явно работает на уровне зависимостей, в то время как CMake работает на более высоком декларативном уровне. К тому же, для make вид зависимости не имеет значения, в отличие от CMake. А это значит, что мы не можем преобразовать абсолютно любой файл для make в файл для CMake.

Стоит обратить внимание, что не все команды, которые указаны в Makefile, необходимо преобразовывать в CMakeLists, поскольку CMake может с ними работать по умолчанию, например, в CMakeLists не нужно указывать правила компиляции. В тоже время в CMakeLists необходимо указывать команды, которые никак не зависят от Makefile.

3.2 Требования к программе

Копмания ООО «Софтком» потребовала, чтобы программа поддерживала следующее:

- Команда include
- Условная обработка
- Распознавание исполняемых файлов и библиотек, а также промежуточных исполняемых файлов
- Пользовательские команды и переменные

3.3 Программная реализация

На вход подаются 2 файла:

1. Название makefile, который будет конвертироваться в CMakeLists
2. Название используемого компилятора

Во время считывания строки идёт её анализ.

[6] https://www.harness.io/harness-devops-academy/what-is-build-automation

Изначально командная строка проверяется на наличие команды include, то есть, пытаемся ли мы во время работы с Makefile считать другой. В данном случае функция, занимающаяся преобразованием, просто вызовется рекурсивно с аргументом include, поскольку команда include как раз отвечает за приостановку чтения одного makefile и перехода к другому.

Также стоит отметить, что программа способна работать с ситуациями, когда команда makefile записана в виде нескольких строк, например:

Листинг 1: Пример работы команды include

```
include Makefile2
    Makefile3
```

Чтение нескольких строк в команде include происходит следующим образом: во время чтения строки мы считываем следующие строчки, которые не были никак опознаны конвертером.

Так, если следующая строчка будет опознана как переменная, то это будет значить, что она не относится к команде include.

После проверки на makefile мы проверяем строчку на условную компиляцию. В makefile она выглядит как:

Листинг 2: Пример условной компиляции

```
ifdef var
    ...
    endif
```

В CMake есть собственный аналог условной компиляции:

Листинг 3: Ожидаемый результат работы конвертера

То есть, если мы встречаем строку с аргументом **ifdef** и

```
if (var) {
    ...
    }
```

параметром **var**, то мы превращаем её в **if(var) {,** если же мы встретили параметр **endif**, то это означает ограничение нашей области **ifdef**, и его можно просто заменить на фигурную скобку.

Следущей проверкой является проверка на таргеты. В make они выглядят следующим образом:

Листинг 4: Пример таргета

```
targets : rules
        recipes
```

В make таргеты представляют из себя один большой граф, где множество одних объектов может зависеть от других

Таргеты можно классифицировать по трём видам поведения:

1) По виду зависимости (от исходных файлов, от объектов, смешанный)

2) По "конечности" (таргет является зависимым от другого объекта, таргет не зависит от других объектов)

3) По содержанию исходных файлов (файлы уже содержат нужную информацию, файлы компилируются в процессе сборки)

Третий пункт можно разделить её на два

3.1) В recipes указан компилятор

3.2) Компилятора в recipes нет

Изначально таргеты проверяются на наличие то, генерируются ли исходные файлы в процессе сборки. Затем происходит проверка на наличие компилятора в recipes. Если он есть, результатом конвертации будет:

Листинг 5: Ожидаемый результат работы конвертера

```
add_custom_target(command COMMAND " + [commandName] + ")
```

Если его нет - будет:

Листинг 6: Ожидаемый результат работы конвертера

```
add_custom_command(OUTPUT " + [outputName] + " COMMAND " +
[commandName] ARGS [argName]))
```

Теперь проверяются таргеты, которые не являются командами, на вид зависимости. В случае, если таргет зависит от исходных файлов, тогда таргет будет конвертирован в:

Листинг 7: Ожидаемый результат работы конвертера

```
add_library([objects] [files])
```

Листинг 8: Ожидаемый результат работы конвертера

```
add_executable([final_object] $<TARGET_OBJECTS> object1 ...
$<TARGET_OBJECTS> objectN)
```

И, наконец, в случае, если у нас зависимость и от исходных файлов, и от объектов, команда будет выглядеть так:

Листинг 9: Ожидаемый результат работы конвертера

```
add_executable([object] $<TARGET_OBJECTS> object1 ... $<TARGET_OBJECTS>
objectN source1.cpp … sourceN.cpp)
```

Последняя проверка строки выполняется на свою причастность к переменной. Так, если в make переменные выглядят следующим образом:

Листинг 10: Переменные в make

```
a = b
```

То результатом конвертации будет:

Листинг 11: Ожидаемый результат работы конвертера

```
set(a b)
```

В противном случае, если мы никак не смогли определить строку, проверка вернёт токен **UNDEFINED**. Если мы его получили при выполнении конвертирующей функции, то она ничего не сделает.

4. Тестирование

В качестве тестирования было рассмотрено несколько типовых проектов:

Листинг 11: Проект на make, который собирает проект, состоящий из одного файла analysis.cpp

```
analysis: analysis.o
	g++ -o analysis analysis.o


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

Листинг 12: Результат работы конвертера

```
cmake_minimum_required(VERSION 3.02)
project(projectName)
add_library(analysis.o OBJECT  analysis.cpp)

add_executable(analysis  $<TARGET_OBJECTS:analysis.o>)
target_link_libraries(analysis  analysis.o)
```

Листинг 13: Проект на make, в котором присутствуют переменные

```
CC=g++
CFLAGS=-c -Wall
general : hello
hello: main.o factorial.o hello.o
	$(CC) main.o factorial.o hello.o -o hello
main.o: main.cpp
	$(CC) $(CFLAGS) main.cpp
factorial.o: factorial.cpp
	$(CC) $(CFLAGS) factorial.cpp
hello.o: hello.cpp
	$(CC) $(CFLAGS) hello.cpp
```

Листинг 14: Результат работы конвертера

```
cmake_minimum_required(VERSION 3.02)
project(projectName)
set(CMAKE_C_COMPILER g++)
set(CMAKE_CPP_COMPILER g++)
set(CMAKE_C_FLAGS -c -Wall)
set(CMAKE_CXX_GLAGS -c -Wall)
set(CC g++)
set(CFLAGS -c -Wall )
add_library(hello.o OBJECT  hello.cpp )
target_compile_options(hello.o PUBLIC  -Wall -c)
add_library(factorial.o OBJECT  factorial.cpp )
target_compile_options(factorial.o PUBLIC  -Wall -c)
add_library(main.o OBJECT  main.cpp )
target_compile_options(main.o PUBLIC  -Wall -c)
add_library(hello $<TARGET_OBJECTS:main.o>
$<TARGET_OBJECTS:factorial.o>
$<TARGET_OBJECTS:hello.o> )
add_executable(general   hello)
```

Листинг 15: Проект на make, в котором присутствует условная компиляция

```
bar = true
foo = bar
ifdef $(foo)
frobozz = yes
endif
```

Листинг 16: Результат работы конвертера

```
cmake_minimum_required(VERSION 3.02)
project(projectName)
set(bar   true)
set(foo   bar)
if(${foo} )
set(frobozz   yes)
endif()
```

Листинг 17: Проект на make, в котором присутствует многофайловость

Makefile:

```
general: hellomake

hellomake: hellofunc.o hellomake2.o
	g++ hellofunc.o hellomake2.o -o hellomake

hellofunc.o: hellofunc.cpp
	g++ -c hellofunc.cpp

include Makefile2
```

Makefile2:

```
hellomake2.o: hellomake2.cpp
		g++ -c hellomake2.cpp
```

Листинг 18: Результат работы конвертера

```
cmake_minimum_required(VERSION 3.02)
project(projectName)

add_custom_command(OUTPUT src.cpp COMMAND python ARGS
./generate_src.py)

add_library(src.o OBJECT  src.cpp)

add_library(src $<TARGET_OBJECTS:src.o> )

add_executable(general  src)
target_link_libraries(general  src)
```

Листинг 19: Проект на make, в котором присутствует компиляция исходных файлов в процессе сборки

```
general: src

src: src.o
	g++ src.o -o src

src.o: src.cpp
	g++ -c src.cpp

	src.cpp:
	python ./generate_src.py > src.cpp
```

Листинг 20: Результат работы конвертер

```
cmake_minimum_required(VERSION 3.02)
project(projectName)

add_custom_command(OUTPUT src.cpp COMMAND python ARGS
./generate_src.py)

add_library(src.o OBJECT  src.cpp)

add_library(src $<TARGET_OBJECTS:src.o> )

add_executable(general  src)
target_link_libraries(general  src)
```

Заключение

В рамках этой учебной практики были решены следующие задачи:

- Изучена логика работы make и Cmake
- Спроектирована логика преобразования make в Cmake
- Произведена техническая реализация преобразования

Репозиторий с кодом доступен по ссылке: https://github.com/lve-gh/make2cmake

Список литературы

[1] – Continuous Delivery. Reliable Software Releases Through Build, Test, and Deployment Automation / Ed. by Jez Humble, David Farley – 2010. – July.
–URL:
https://www.google.ru/books/edition/Continuous_Delivery/6ADDuzere-YC?hl=ru&gbpv=0 (дата обращения: 18.06.2024)
[2] – Building Automation / Ed. by Stephan D.Ritchie – 2010. – January. – URL:
https://www.researchgate.net/publication/302314513_Build_Automation (дата обращения: 18.06.2024)
[3] – Installing and using CMake. / Ed. by Matthew John Yee-King – 2024. – April. – URL:
https://www.researchgate.net/publication/380138251_Installing_and_using_CMake (дата обращения: 18.06.2024)