

сmake

Практикум, 3 курс

Рассказывает:

Подымов Владислав Васильевич

Осень 2016

Вступление

Основная задача stake:

Собрать проект

(build project)

А что такое “проект”?

Как минимум, весь исходный код, лежащий в заданной папке, который хочется скомпилировать

А что такое “собрать”?

Например, скомпилировать *(или сделать что-то нетривиальное, что хочется сделать с исходным кодом)*

Как это можно сделать?

- Вручную *(набирать в консоли нужные команды)*
- make *(один раз написать нужные команды с зависимостями и запускать их с помощью утилиты make)*

А можно ли лучше?

IDE^{*}

“хочу, чтобы было всё и сразу,
в том числе не писать никаких инструкций по сборке”

IDE – это приложение, позволяющее удобно и просто писать программные проекты

Обычно **IDE** включает в себя

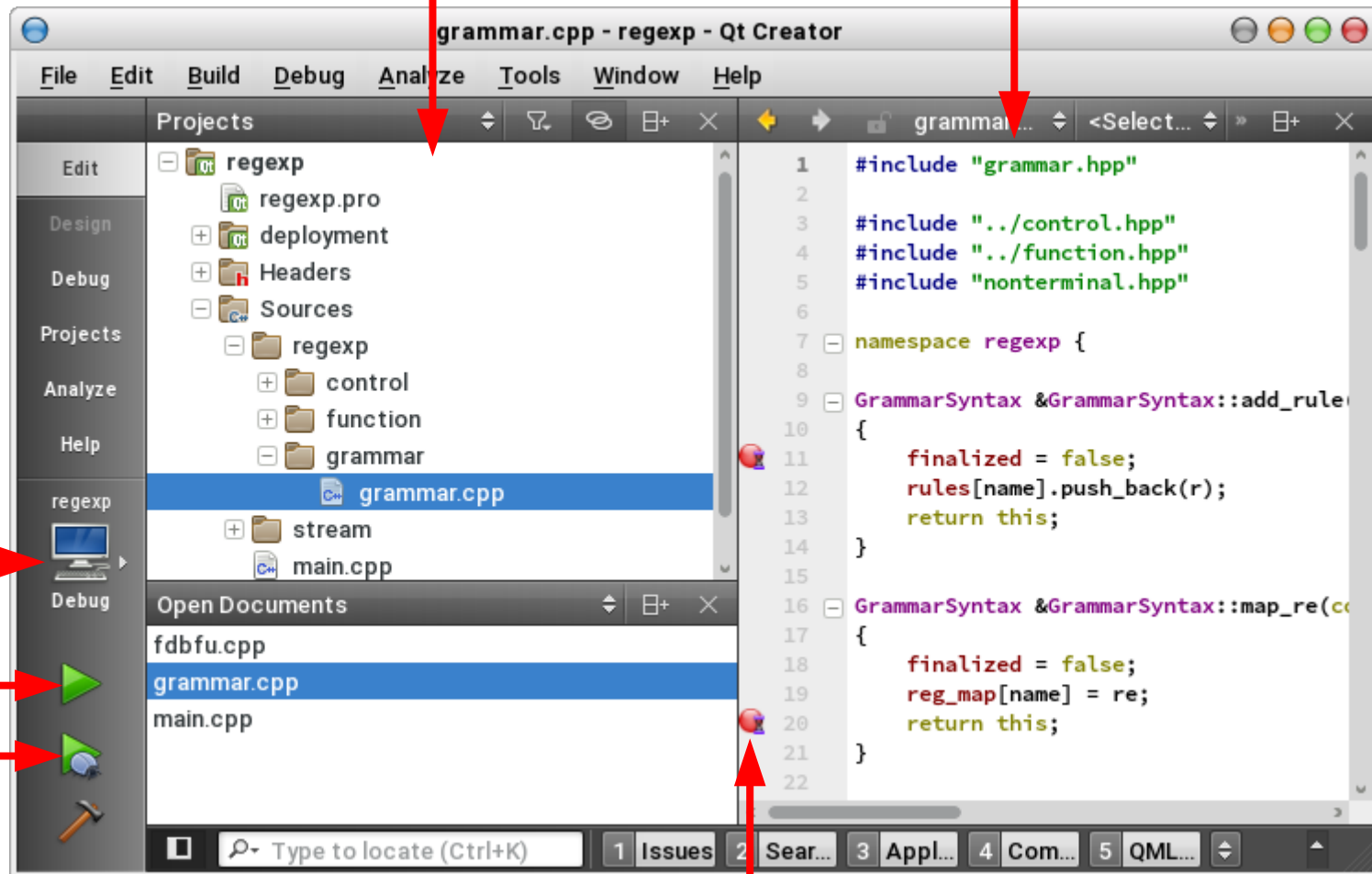
- Редактор исходных файлов *(с кучей “фишек”, облегчающих написание кода)*
- Визуальные средства для автоматической сборки исходных файлов
(в окошке выставил флаги и пути, нажал кнопку – и оно собралось)
- Отладчик *(общающийся с текстовым редактором и сборщиком)*

* интегрированная среда разработки
(Integrated Development Environment)

IDE

Как это может выглядеть:

Редактор



Сборщик

Отладчик

IDE

И много ли разных IDE бывает?

Например, для C/C++:

Название	Платформа	Лицензия
Microsoft Visual Studio	Windows	Проприетарное
Eclipse	Windows, Linux, OS X	Свободное (EPL)
QT Creator *	Windows, Linux, OS X	Свободное (GPL)
Code::Blocks	Windows, Linux, OS X	Свободное (GPL)
Anjuta	Linux	Свободное (GPL)
NetBeans	Windows, Linux, OS X	Свободное (CDDL, GPL)
C++Builder	Windows, OS X	Проприетарное
... (место кончилось)		

* он был на предыдущем слайде

IDE

Плюсы:

- Быстро пишется
- Легко собирается
- Нужно думать только над самим кодом

Минусы:

- **У каждого IDE свои служебные файлы и свой способ сборки**

Если проект пишется группой, то всем работать в одном IDE?

А как это сочетать с git-репозиторием?

(никто не говорил, что git вообще должен поддерживаться)

А если хочется выложить исходники “в мир”, то как это собирать пользователям?

- **Каждое IDE умеет только то, что умеет**

Умеет ли оно генерировать файлы для сборки?

Не факт

Умеет ли оно собирать больше одного бинарного файла?

Не факт

Умеет ли оно собирать что-то кроме бинарных файлов?

Не факт

(документацию, тестовые входные наборы, вспомогательные скрипты)

Надстройки над make

Второй способ, как можно упростить сборку проекта, - использовать средства, умеющие **генерировать makefile'ы** нужного вида

- Пишем исходный код в любом текстовом редакторе
- Пишем команды для системы сборки
- Запускаем утилиту, генерирующую makefile'ы

(или не обязательно их – просто то, что собирает проект)

То есть всё сводится к make? Может, писать сразу makefile'ы?

Каждое средство умеет решать свой класс задач

Когда решается конкретный класс задач, не нужны **все** возможности make

Каждую конкретную задачу проще решать в системе,
предназначенной для этой задачи

сmake

cmake.org/documentation/

Что это такое:

- Консольная утилита

```
terminal>  
terminal> cmake ..  
-- The C compiler identification  
-- The CXX compiler identificati
```

*Она генерирует makefile'ы,
собирающие проект*

- Файлы, которые читает и исполняет эта утилита



*В make было "makefile",
а здесь – "CMakeLists.txt"*

- Скриптовый язык, на котором пишутся эти файлы

```
CMakeLists.txt  
1 cmake_minimum_required(VERSION 3.3)  
2 add_subdirectory("lib")  
3 add_executable(main main.cpp)  
4 target_link_libraries(main lib)  
5
```

Пока что очень похоже на make. Так насколько это лучше?

сmake

- С помощью cmake можно собирать и в Windows
(не факт что сгенерируются именномakefile'ы – проверьте, если хотите)
- Утилита специализирована: основное назначение – собирать проекты
(поэтому и язык специализирован и более естественен)
- В файлах, читаемых утилитой, описываются
 - Набор исходных файлов
 - Их взаимосвязь *(эти файлы используются здесь, эти – там)*
 - Особые параметры сборки
 - Флаги компилятора
 - Используемые нестандартные библиотеки
 - Необходимые для сборки требования
 - ...

Hello, World!

Пишем программу "Hello, World!"

```
main.cpp
1 #include <iostream>
2 int main(int argc, char ** argv) {
3     std::cout << "Hello, World!" << std::endl;
4 }
5
```

Пишем CMakeLists.txt

```
CMakeLists.txt
1 cmake_minimum_required(VERSION 3.3)
2 add_executable(main main.cpp)
3
```

Собери бинарный файл

Файл называется main

Файл собирается из main.cpp

Запускаем утилиту cmake

```
terminal> ls
CMakeLists.txt main.cpp
terminal> cmake .
        Всякий разный вывод в консоль
```

Сгенерировался makefile

```
terminal> ls
CMakeCache.txt cmake_install.cmake main.cpp
CMakeFiles      CMakeLists.txt Makefile
```

Запускаем утилиту make

```
terminal> make
Scanning dependencies of target main
[ 50%] Building CXX object CMakeFiles/main.dir/main.cpp.o
[100%] Linking CXX executable main
[100%] Built target main
```

Бинарный файл готов

```
terminal> ls
CMakeCache.txt cmake_install.cmake main      Makefile
CMakeFiles      CMakeLists.txt  main.cpp
terminal> ./main
Hello, World!
terminal>
```

Цели makefile'a

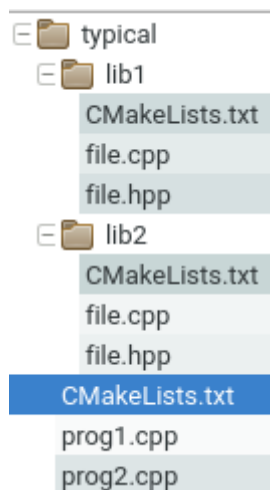
А какие цели создаёт make в makefile'е?

- **all** - “собрать все бинарные файлы”, цель по умолчанию
- цель для каждого бинарного файла
- **clean** - “стереть все объектные и бинарные файлы”
- дополнительные цели – например, “собрать объектный файл”
- служебные цели – об этом немного позже

(эти цели лучше не собирать просто так)

Типовое применение

CmakeLists.txt в каждой подпапке проекта



```
1 add_library(lib1 file.cpp file.hpp)
```

Собери **статическую библиотеку** lib1 из исходных файлов file.hpp, file.cpp

```
1 add_library(lib2 file.cpp file.hpp)
```

Собери **статическую библиотеку** lib2 из исходных файлов file.hpp, file.cpp

```
1 cmake_minimum_required(VERSION 3.3)
2 project(my_superproject)
3 add_subdirectory("lib1")
4 add_subdirectory("lib2")
5 add_executable(prog1 prog1.cpp)
6 target_link_libraries(prog1 lib1)
7 add_executable(prog2 prog2.cpp)
8 target_link_libraries(prog2 lib2)
```

Мой проект называется вот так

Добавь к проекту папки lib1, lib2 и собери их согласно написанным в них cmake-файлам

Добавь к проекту бинарник prog2; он будет собираться из исходного файла prog2.cpp

Когда будешь собирать prog2, слинкуй его с библиотекой lib2

Создаём и заходим в специальную папку build

Собираем здесь проект из исходников, лежащих одной папкой выше

```
terminal> ls
CMakeLists.txt lib1 lib2 prog1.cpp prog2.cpp
terminal> mkdir build
terminal> cd build
terminal> cmake ..
```

Типовое применение: stake+git

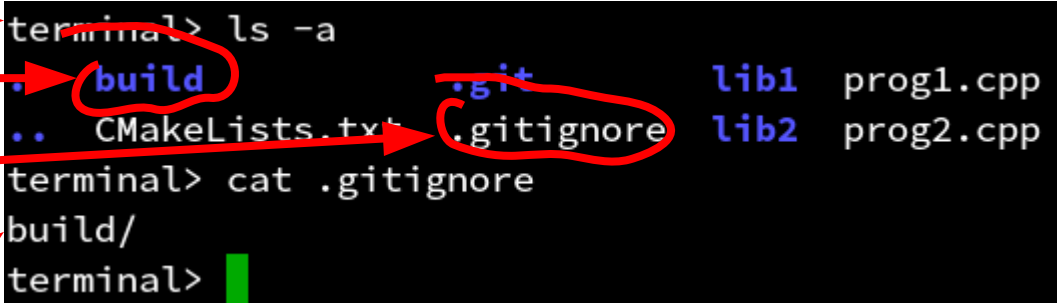
Настраиваем обычным образом git-репозиторий

Сейчас находимся в его корне

Создаём папку build

Создаём файл .gitignore, если его ещё нет

Дописываем в .gitignore строку "build/"



```
terminal> ls -a
. build .git lib1 prog1.cpp
.. CMakeLists.txt .gitignore lib2 prog2.cpp
terminal> cat .gitignore
build/
terminal>
```

The terminal screenshot shows the following sequence of commands and outputs: `terminal> ls -a` outputs `. build .git lib1 prog1.cpp` and `.. CMakeLists.txt .gitignore lib2 prog2.cpp`. `terminal> cat .gitignore` outputs `build/`. Red arrows point from the text annotations to the `build` directory in the `ls` output, the `.gitignore` file in the `ls` output, and the `build/` line in the `cat` output.

А дальше всё как на предыдущем слайде

И что это значит?

- В файле “.gitignore” прописываются (по одному на строку) объекты (файлы и папки), которые git будет игнорировать – при учёте файлов репозитория действовать так, будто их нет
- В папке build можно спокойно собирать проект, копить мусор и в целом делать всё, что заблагорассудится, и git этого не увидит

Ленивая сборка

В `make` есть две степени “ленивости” при сборке проекта:

- Он генерирует `makefile`’ы, и они (по своей природе) делают ленивую сборку
- Он генерирует “хитрые” `makefile`’ы:
 - В `makefile`’ах отслеживается изменение не только исходных файлов, но и всех используемых `CMakeLists.txt`
 - Если файлы `CMakeFiles.txt` изменились, команда `make`
 - вызовет `make`, чтобы пересобрать `makefile`’ы, которые нужно изменить
 - вызовет требуемый `make` с изменёнными `makefile`’ами

Не нужно при каждом изменении файлов `CMakeLists.txt` вызывать команду `make`: достаточно вызвать `make` так, будто `make` уже был вызван

Основные концепты

- В cmake есть точно такие же переменные, как в make
(только вместо $\$(...)$ пишется $\${...}$)
- Кроме того, в cmake можно создавать разные **объекты**:*

Создать объект “библиотека”	→	<code>add_library(lib file.hpp file.cpp)</code>
Создать объект “бинарный файл”	→	<code>add_executable(main main.cpp)</code>
Слинковать один объект с другим	→	<code>target_link_libraries(main lib)</code>
		<code>4 </code>

А что такое этот “объект”?

Обычно это **имя цели** в makefile’е,

а значения, используемые при создании объекта - **зависимости**

Например, в генерируемых makefile’ах будут

- цель **lib** с зависимостями **file.hpp**, **file.cpp**
- цель **main** с зависимостями **main.cpp**, **lib**

* в документации к CMake нет понятия “объект”, там говорится сразу “цель”;

здесь я пишу “объект”, чтобы подчеркнуть, что это не совсем та же цель, что и в make

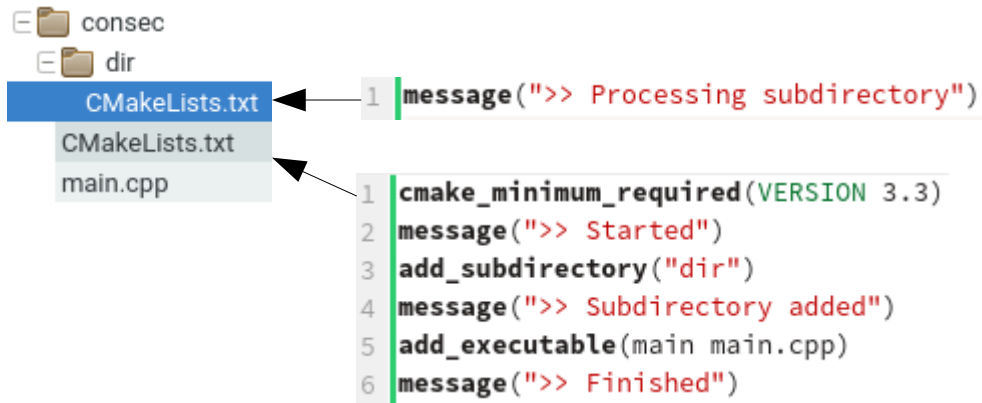
Исполнение команд

Команды стайке исполняются **последовательно** сверху вниз

Это значит, в числе прочего, что перед использованием переменные надо определять
(*всё немного хитрее, но лучше придерживаться этого правила*)

Команду вызова обработки другого файла CMakeLists.txt

можно расценивать как **вызов последовательной процедуры**



“message” - команда вывода в консоль

```
terminal> cmake .
-- The C compiler identification is GNU 4.8.5
-- The CXX compiler identification is GNU 4.8.5
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - failed
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - failed
>> Started
>> Processing subdirectory
>> Subdirectory added
>> Finished
-- Configuring done
-- Generating done
-- Build files have been written to:
terminal>
```


Некоторые команды

cmake_minimum_required(VERSION *num*)

указывается в начале главного cmake-файла; *num* – номер версии cmake, начиная с которого написанные команды точно заработают

add_executable(*name src_1 src_2 ...*)

собрать бинарный файл *name* из исходников *src_i*; функция `main` – в *src_1*

add_dependencies(*name dep_1 dep_2 ...*)

добавить зависимости *dep_i* к объекту (цели) *name*

add_subdirectory(*name*)

добавить к проекту папку *name* и обработать в ней CMakeLists.txt

add_library(*name src_1 src_2 ...*)

собрать статическую библиотеку *name* из исходников *src_i*

target_link_libraries(*targ lib_1 lib_2 ...*)

слинковать объект *targ* с библиотеками *lib_i*

target_...(*targ ...*)

есть и другие команды, настраивающие отдельные объекты (*цели*, *target'ы*)

Некоторые команды

project(*name*)

- сохраняет имя *name* в переменную **PROJECT_NAME**
- сохраняет путь к исходным файлам в переменные **PROJECT_SOURCE_DIR** и **<name>_SOURCE_DIR**
- сохраняет путь к собираемым файлам в переменные **PROJECT_BINARY_DIR** и **<name>_BINARY_DIR**
- заводит переменные, специфичные для языка исходного кода
 - например, **CMAKE_CXX_COMPILER** – переменная для компилятора C/C++
 - не просто так упоминалось, что cmake умеет нечто большее, чем компилировать исходники C/C++;
например, он поддерживает и другие языки программирования
 - языки по умолчанию – C и CXX (то есть C++)

project(*name* LANGUAGES *lang1 lang2 ...*)

то же самое, но с явным указанием языков исходного кода *lang_i*

enable_language(*lang*)

заводит переменные, специфичные для языка *lang*

Некоторые команды

set(*name val*)

установить в переменную *name* значение *val*

find_package(*name*)

найти пакет *name*, поддерживающий сборку системой *stake*,
и загрузить его объекты и переменные, и результат поиска (да/нет) в **<name>_FOUND**

В дополнительных опциях этой команды можно указать

- конкретные объекты пакета
- пути, по которым его требуется искать (*помимо стандартных*)
- минимальную требуемую версию пакета
- ... (*смотрите документацию к stake*)

Если удастся таким образом подключить пакет, то можно будет использовать,
в числе прочего, переменные со значениями

- папки с исходным кодом пакета
- папки с бинарными файлами пакета
- папки с header'ами пакета

Пример использования: **find_package**(Boost 1.49.0) загружает всё,
что известно об установленном в системе пакете Boost версии не ниже 1.49.0

Некоторые команды

find_library(*var name*)

найти библиотеку *name*; если найдена, сохранить её в переменную *var*;
если не найдена, записать в неё значение **<var>-NOTFOUND**

Обращение к библиотеке происходит по значению *var*:

target_link_libraries(*targ* $\${var}$)

find_...(...)

есть еще объекты, которые *stake* умеет искать (*смотрите документацию*)

include_directories(*dir_1 dir_2 ...*)

добавить папки *dir_i* к списку мест, в которых *stake* будет искать файлы,
необходимые для `#include ...` в C/C++ и аналогичных конструкций других языков

message([*mode*] "*message*")

вывести в консоль сообщение *message*, опционально пометив его как
предупреждение/ошибку/... опцией *mode* (*смотрите документацию к команде*)

Некоторые команды

```
foreach(var val_1 val_2 ...)  
  command_1  
  command_2  
  ...  
endforeach(var)
```

Записывая последовательно
в переменную *var* значения *val_i*,
выполнить команды *command_i*

```
if(expr_1)  
  commands  
elseif(expr_2)  
  commands  
elseif(expr_3)  
  commands  
...  
else  
  commands  
endif
```

Обычная условная инструкция
Выражения *expr_i* могут содержать:

- константы
- строки (некоторые строки равны **true** или **false**)
- булевы операции
- строковые операции
- проверки наличия переменных
- проверки соответствия строки заданному формату
- ...

```
while(expr)  
  commands  
endwhile(expr)
```

Обычный цикл while

```
break(), continue()
```

Имеют такое же значение, как в C/C++

Некоторые переменные

Есть ряд переменных, которые cmake использует для конкретных целей, например:

CMAKE_BINARY_DIR	корневая папка, в которой собирается проект
CMAKE_SOURCE_DIR	корневая папка, в которой лежат исходные файлы, собираемые cmake'ом
CMAKE_INCLUDE_PATH	набор нестандартных папок, в которых cmake будет искать заголовочные файлы (см. include_directories)
CMAKE_CURRENT_BINARY_DIR	папка сборки проекта, в которой cmake находится сейчас
CMAKE_CURRENT_SOURCE_DIR	папка с исходниками, в которой cmake находится сейчас

Некоторые переменные

Есть ряд переменных, которые cmake использует для конкретных целей, например:

Переменные проекта

см. команду **project**

CMAKE_LIBRARY_PATH

нестандартные папки,
в которых cmake должен искать библиотеки
(см. **find_library**)

CMAKE_CXX_STANDARD

используемый в исходном коде
стандарт языка C++
(**98, 11, 14**)

Например, если код – в стандарте 11:

set(CMAKE_CXX_STANDARD 11)

CMAKE_CXX_FLAGS

флаги компилятора C++

Например, если хочется подключить флаг “выводить все предупреждения”:

set(CMAKE_CXX_FLAGS \${CMAKE_CXX_FLAGS} “-Wall”)

Заключение



```
terminal> ls
CMakeLists.txt lib1 lib2 prog1.cpp prog2.cpp
terminal> mkdir build
terminal> cd build
terminal> cmake ..
```

В языке стাকে есть много команд

Этими командами можно производить очень тонкую настройку принципов сборки

Настолько тонкую, что в общем-то, он может собрать что угодно из чего угодно

Самые простые команды и переменные были выписаны на слайдах (на всякий случай)

Этих команд намного больше, и все команды можно тонко настраивать

На каждое примитивное действие по сборке проекта имеется своя команда

Но чем проще проект, тем выше вероятность, что выглядеть всё будет как в типовом случае