Основы автоматизации сборки проектов на C++.

Инженер каф. 40 Курова А.С.

Зачем нужна автоматизация сборки?

Если ваша программа состоит из одного файла

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

а если их становится несколько

```
g++ main.cpp mylib.cpp mysublib.cpp -o main
```

Если же файлов исходников становится сильно больше (а это неизбежно наступает, или же ваш код со временем превращается в простыню на десятки тысяч строк, где невозможно ориентироваться),

например:

```
MBP-Anastasia-2:Root EvaRage$ ls
BkgFitSyst.cpp                CalculationFakeRate2dim.cpp   Config.cpp                    FakeRate.cpp                  FitPlotter.cpp                TreeReader.cpp
Calculation.cpp               CalculationLeak.cpp           ControlPlots.cpp              FakeRatePlotter.cpp           IsolationPlotter.cpp          TreeReader2dim.cpp
Calculation2dim.cpp           CalculationRData.cpp          DirectoryReader.cpp           FakeRateReader.cpp            LeakReader.cpp                WenuReader.cpp
CalculationEToGam.cpp         CalculationWenu.cpp           EToGamReader.cpp              FakeRateReader2dim.cpp        Plotter.cpp
CalculationFakeRate.cpp       Common.cpp                    EventDuplicateCheckerTool.cpp FakeRateSyst.cpp              RDataReader.cpp
```

то такой формат компиляции становится неудобным.

Тут приходит на помощь автоматизация сборки.

Makefile - традиционный способ

команда make будет искать в текущем каталоге Makefile, в котором содержатся инструкции для компиляции

Makefile состоит из блоков

```
цель: зависимости

[tab] команда
```

Простейший Makefile в для примера, приведенного выше:

```
all:
    g++ main.cpp mylib.cpp mysublib.cpp -o main
```

Удобно использовать цель по умолчанию `all` в любом вашем Makefile, т.к. при вызове make не потребуется писать имя цели.

Повторяет нашу командную строку. Теперь запустить компиляции можно так:

```
make
```

Согласитесь, проще?
Усложним...

Makefile - традиционный способ

Следующий makefile сделает всё то же самое, но определяет зависимости между частями проекта. Это позволяет не компилировать большой проект целиком, а только ту его часть которая изменилась, что может существенно ускорить процесс

```
all: main

main: main.o mylib.o mysublib.o
        g++ main.o mylib.o mysublib.o -o main

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

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

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

clean:
        rm -rf *.o main
```

Каждый блок можно вызывать отдельно: make цель
Например, make mylib.o, make main, make clean

При выполнении команды make блок clean не будет выполняться, почему?
Как нужно изменить Makefile, чтобы блок clean выполнялся?

Makefile - традиционный способ

Чтобы ещё ~~усложнить~~ улучшить ваш Makefile можно использовать комментарии, шаблоны, переменные и пр.

```
#это комментарий и его команда make будет игнорировать
```

полезный шаблон `'*': *.cpp =` все файлы с расширением cpp в каталоге

Переменные задаются синтаксисом `NAME=VALUE` и вызываются как `$(NAME)`, удобны замены повторяющихся частей, что позволяет избежать ошибок при дальнейшем изменении Makefile (шаблон '*' в переменной <u>не будет</u> работать)

Попробуйте оптимизировать имеющийся Makefile с использованием этих средств

А теперь грабли Makefile!

при изменении заголовочных файлов проект <u>не будет пересобираться</u>, т.к. они не прописываются в зависимостях. Поэтому чтобы быть уверенным в правильности сборки необходимо использовать make clean перед.

Для дальнейшего изучения:
http://rus-linux.net/nlib.php?name=/MyLDP/algol/gnu_make/gnu_make_3-79_russian_manual.html#SEC1

cmake - ещё более автоматизированная сборка

cmake - надстройка над make, которая генерирует Makefiles (в случае Unix), имеет возможность работать с большими сложными проектами, с подпроектами, искать и устанавливать связи с библиотеками, создавать библиотеки статические/динамические, может работать со сложной иерархией директорий и много чего ещё.

Постоянно обновляется и расширяется.

Директивы cmake записываются в файле CMakeLists.txt для каждого отдельного проекта в его папке

Удобно проводить сборку в папке отдельной от вашего проекта, например, build. Сборка из папки build будет производится как

cmake ../source

make

Если проект содержит несколько компонентов или зависит от других пакетов, то удобно иметь папку source, где будут храниться папки с зависимыми пакетами. Обычно это source.

CMakeLists.txt для нашего проекта

```
cmake_minimum_required(VERSION 2.8)      # Проверка версии CMake.
# Если версия установленой программы
# старее указаной, произайдёт аварийный выход.

project(EventChecker)                     # Название проекта

set(SOURCE_EXE main.cpp)                 # Установка переменной со списком исходников для
исполняемого файла

file (GLOB_RECURSE CPP_FILES *.cpp )    #  установка переменной для исходников с
расширением cpp (рекурсивный поиск по всем подпапкам)

set(SOURCE_LIB ${CPP_FILES})             # Тоже самое, но для библиотеки

add_library(MyLib STATIC ${SOURCE_LIB})   # Создание статической библиотеки с именем MyLib

add_executable(main ${SOURCE_EXE})      # Создает исполняемый файл с именем main

target_link_libraries(main MyLib)               # Линковка программы с библиотекой
```

Пример CMakeLists.txt для нескольких связанных проектов

выделим библиотеку MyLib как отдельный проект в подпапке Mylib. В ней же создадим отдельный файл CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)      # Проверка версии CMake.
project(Mylib)                     # Название проекта
file (GLOB_RECURSE CPP_FILES *.cpp )    #  установка переменной для исходников с расширением
cpp (рекурсивный поиск по всем подпапкам)
set(SOURCE_LIB ${CPP_FILES})             # Тоже самое, но для библиотеки
add_library(MyLib STATIC ${SOURCE_LIB})   # Создание статической библиотеки с именем MyLib
```

При этом основной файл в файл CMakeLists.txt поменяется

```
cmake_minimum_required(VERSION 2.8)      # Проверка версии CMake.
project(EventChecker)                    # Название проекта
set(SOURCE_EXE main.cpp)                # Установка переменной со списком исходников для
исполняемого файла
include_directories(Mylib)
add_executable(main ${SOURCE_EXE})      # Создает исполняемый файл с именем main
add_subdirectory(Mylib)
target_link_libraries(main MyLib)               # Линковка программы с библиотекой
```

Пример CMakeLists.txt для проекта с поиском сторонних библиотек

```
cmake_minimum_required(VERSION 2.8)      # Проверка версии CMake.
                                                                    # Если версия установленой программы
                                                                    # старее указаной, произайдёт аварийный вых

project(ZGamma)                    # Название проекта

list(APPEND CMAKE_PREFIX_PATH $ENV{ROOTSYS})
find_package(ROOT REQUIRED COMPONENTS RIO Net)

find_package(Boost 1.68.0)

if(Boost_FOUND)
  include_directories(${Boost_INCLUDE_DIRS})
endif()

#---Define useful ROOT functions and macros (e.g. ROOT_GENERATE_DICTIONARY)
include(${ROOT_USE_FILE})

include_directories("${CMAKE_CURRENT_SOURCE_DIR}/include"  "ZGamma" "$ENV{HOME}/atlasstyle-00-04-02/")

set(SOURCE_EXE main.cpp)                # Установка переменной со списком исходников для исполняемого файла

file (GLOB_RECURSE CPP_FILES Root/*.cpp $ENV{HOME}/atlasstyle-00-04-02/*.C)

set(SOURCE_LIB ${CPP_FILES})             # Тоже самое, но для библиотеки

add_library(Zgam STATIC ${SOURCE_LIB})   # Создание статической библиотеки с именем Zgam

add_executable(main ${SOURCE_EXE})      # Создает исполняемый файл с именем main

if(Boost_FOUND)
  target_link_libraries(main Zgam ${ROOT_LIBRARIES} ${Boost_LIBRARIES})              # Линковка программы с библиотекой
else()
  target_link_libraries(main Zgam ${ROOT_LIBRARIES})              # Линковка программы с библиотекой
endif()
```

при добавлении сторонней библиотеки можно потребовать, чтобы в ней присутствовали некоторые специальные её компоненты и проверить версию

CMakeLists.txt также можно использовать условия if else

Упражнение

1) Добавить к нашему проекту библиотеку Boost и ROOT (Boost можно скачать тут https://www.boost.org/users/download/)
чтобы подключить root на lxfarm с конфигурацией для cmake, необходимы следующие команды

```
export ATLAS_LOCAL_ROOT_BASE=/cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase
source ${ATLAS_LOCAL_ROOT_BASE}/user/atlasLocalSetup.sh
lsetup 'root 6.14.04-x86_64-slc6-gcc62-opt'
lsetup cmake
```

2) С помощью функции split (#include "boost/algorithm/string/split.hpp") из библиотеки Boost разделить вводимую строку на слова и вывести.
Также пригодится #include <boost/algorithm/string/classification.hpp>

Д.з. использовать вместо std::string TString (“TString.h”)