

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ  
ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«МУРМАНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра ЦТМиЭ

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ  
К ЛАБОРАТОРНЫМ, КОНТРОЛЬНЫМ И САМОСТОЯТЕЛЬНЫМ РАБОТАМ  
СТУДЕНТОВ**

По дисциплине: Использование инструментальных библиотек при разработке  
программного обеспечения

название дисциплины

для направления (специальности) 09.03.01

код направления (специальности)

---

«Информатика и вычислительная техника»

наименование направления подготовки

Мурманск  
2021

Составитель – Ершов Павел Сергеевич, старший преподаватель кафедры ЦТМиЭ

Методические указания к выполнению контрольной работы рассмотрены и одобрены на заседании кафедры-разработчика:

Цифровых технологий, математики и экономики

название кафедры

21.06.2021, протокол № 12.

дата

Рецензент – Романовская Ю.В., доцент кафедры цифровых технологий, математики и экономики

## ОГЛАВЛЕНИЕ

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ .....	4
ЗАДАНИЕ ДЛЯ САМОСТОЯТЕЛЬНО РАБОТЫ.....	16
ЗАДАНИЕ ДЛЯ КОНТРОЛЬНОЙ РАБОТЫ .....	16
ТРЕБОВАНИЯ К ОТЧЕТУ О ВЫПОЛНЕНИИ РАБОТЫ.....	16
СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ И ИНТЕРНЕТ-РЕСУРСОВ.....	17
Приложение 1. ОБРАЗЕЦ ТИТУЛЬНОГО ЛИСТА .....	18

## КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

CMake (от англ. cross-platform make) — это кроссплатформенная система автоматизации сборки программного обеспечения из исходного кода. CMake не занимается непосредственно сборкой, а лишь генерирует файлы управления сборкой из файлов CMakeLists.txt:

Makefile в системах Unix для сборки с помощью make;  
файлы projects/solutions (.vcxproj/.vcproj/.sln) в Windows для сборки с помощью Visual C++;  
проекты XCode в Mac OS X.  
Запуск CMake

Ниже приведены примеры использования языка CMake, по которым Вам следует попрактиковаться. Экспериментируйте с исходным кодом, меняя существующие команды и добавляя новые. Чтобы запустить данные примеры, установите CMake с официального сайта.

### Принцип работы

Система сборки CMake представляет из себя оболочку над другими платформенно зависимыми утилитами (например, Ninja или Make). Таким образом, в самом процессе сборки, как бы парадоксально это ни звучало, она непосредственного участия не принимает.

Система сборки CMake принимает на вход файл CMakeLists.txt с описанием правил сборки на формальном языке CMake, а затем генерирует промежуточные и нативные файлы сборки в том же каталоге, принятых на Вашей платформе.

Сгенерированные файлы будут содержать конкретные названия системных утилит, директорий и компиляторов, в то время как команды CMake орудуют лишь абстрактным понятием компилятора и не привязаны к платформенно зависимым инструментам, сильно различающихся на разных операционных системах.

### Проверка версии CMake

Команда `cmake_minimum_required` проверяет запущенную версию CMake: если она меньше указанного минимума, то CMake завершает свою работу фатальной ошибкой. Пример, демонстрирующий типичное использование данной команды в начале любого CMake-файла:

```
# Задать третью минимальную версию CMake:  
cmake_minimum_required(VERSION 3.0)
```

Как подметили в комментариях, команда `cmake_minimum_required` выставляет все флаги совместимости (смотреть `cmake_policy`). Некоторые разработчики намеренно выставляют низкую версию CMake, а затем корректируют функционал вручную. Это позволяет

одновременно поддерживать древние версии CMake и местами использовать новые возможности.

## Оформление проекта

В начале любого CMakeLists.txt следует задать характеристики проекта командой `project` для лучшего оформления интегрированными средами и прочими инструментами разработки.

```
# Задать характеристики проекта "MyProject":  
project(MyProject VERSION 1.2.3.4 LANGUAGES C CXX)
```

Стоит отметить, что если ключевое слово `LANGUAGES` опущено, то по умолчанию задаются языки `C CXX`. Вы также можете отключить указание любых языков путём написания ключевого слова `NONE` в качестве списка языков или просто оставить пустой список.

## Запуск скриптовых файлов

Команда `include` заменяет строку своего вызова кодом заданного файла, действуя аналогично препроцессорной команде `include` языков C/C++. Этот пример запускает скриптовый файл `MyCMakeScript.cmake` описанной командой:

```
message("TEST_VARIABLE' is equal to [${TEST_VARIABLE}]")
```

```
# Запустить скрипт `MyCMakeScript.cmake` на выполнение:  
include(MyCMakeScript.cmake)
```

```
message("TEST_VARIABLE' is equal to [${TEST_VARIABLE}]")
```

В данном примере, первое сообщение уведомит о том, что переменная `TEST_VARIABLE` ещё не определена, однако если скрипт `MyCMakeScript.cmake` определит данную переменную, то второе сообщение уже будет информировать о новом значении тестовой переменной. Таким образом, скриптовый файл, включаемый командой `include`, не создаёт собственной области видимости, о чём упомянули в комментариях к предыдущей статье.

## Компиляция исполняемых файлов

Команда `add_executable` компилирует исполняемый файл с заданным именем из списка исходников. Важно отметить, что окончательное имя файла зависит от целевой платформы (например, `<ExecutableName>.exe` или просто `<ExecutableName>`). Типичный пример вызова данной команды:

```
# Скомпилировать исполняемый файл "MyExecutable" из  
# исходников "ObjectHandler.c", "TimeManager.c" и "MessageGenerator.c":
```

```
add_executable(MyExecutable ObjectHandler.c TimeManager.c MessageGenerator.c)
```

### Компиляция библиотек

Команда `add_library` компилирует библиотеку с указанным видом и именем из исходников. Важно отметить, что окончательное имя библиотеки зависит от целевой платформы (например, `lib<LibraryName>.a` или `<LibraryName>.lib`). Типичный пример вызова данной команды:

```
# Скомпилировать статическую библиотеку "MyLibrary" из  
# исходников "ObjectHandler.c", "TimeManager.c" и "MessageConsumer.c":  
add_library(MyLibrary STATIC ObjectHandler.c TimeManager.c MessageConsumer.c)
```

Статические библиотеки задаются ключевым словом `STATIC` вторым аргументом и представляют из себя архивы объектных файлов, связываемых с исполняемыми файлами и другими библиотеками во время компиляции;

Динамические библиотеки задаются ключевым словом `SHARED` вторым аргументом и представляют из себя двоичные библиотеки, загружаемые операционной системой во время выполнения программы;

Модульные библиотеки задаются ключевым словом `MODULE` вторым аргументом и представляют из себя двоичные библиотеки, загружаемые посредством техник выполнения самим исполняемым файлом;

Объектные библиотеки задаются ключевым словом `OBJECT` вторым аргументом и представляют из себя набор объектных файлов, связываемых с исполняемыми файлами и другими библиотеками во время компиляции.

### Добавление исходников к цели

Бывают случаи, требующие многократного добавления исходных файлов к цели. Для этого предусмотрена команда `target_sources`, способная добавлять исходники к цели множество раз.

Первым аргументом команда `target_sources` принимает название цели, ранее указанной с помощью команд `add_library` или `add_executable`, а последующие аргументы являются списком добавляемых исходных файлов.

Повторяющиеся вызовы команды `target_sources` добавляют исходные файлы к цели в том порядке, в каком они были вызваны, поэтому нижние два блока кода являются функционально эквивалентными:

```
# Задать исполняемый файл "MyExecutable" из исходников  
# "ObjectPrinter.c" и "SystemEvaluator.c":  
add_executable(MyExecutable ObjectPrinter.c SystemEvaluator.c)
```

```
# Добавить к цели "MyExecutable" исходник "MessageConsumer.c":  
target_sources(MyExecutable MessageConsumer.c)  
# Добавить к цели "MyExecutable" исходник "ResultHandler.c":
```

```
target_sources(MyExecutable ResultHandler.c)
```

```
# Задать исполняемый файл "MyExecutable" из исходников  
# "ObjectPrinter.c", "SystemEvaluator.c", "MessageConsumer.c" и "ResultHandler.c":  
add_executable(MyExecutable ObjectPrinter.c SystemEvaluator.c MessageConsumer.c  
ResultHandler.c)
```

Генерируемые файлы

Местоположение выходных файлов, сгенерированных командами `add_executable` и `add_library`, определяется только на стадии генерации, однако данное правило можно изменить несколькими переменными, определяющими конечное местоположение двоичных файлов:

Переменные `RUNTIME_OUTPUT_DIRECTORY` и `RUNTIME_OUTPUT_NAME` определяют местоположение целей выполнения;  
Переменные `LIBRARY_OUTPUT_DIRECTORY` и `LIBRARY_OUTPUT_NAME` определяют местоположение библиотек;  
Переменные `ARCHIVE_OUTPUT_DIRECTORY` и `ARCHIVE_OUTPUT_NAME` определяют местоположение архивов.

Исполняемые файлы всегда рассматриваются целями выполнения, статические библиотеки — архивными целями, а модульные библиотеки — библиотечными целями. Для "не-DLL" платформ динамические библиотеки рассматриваются библиотечными целями, а для "DLL-платформ" — целями выполнения. Для объектных библиотек таких переменных не предусмотрено, поскольку такой вид библиотек генерируется в недрах каталога `CMakeFiles`.

Важно подметить, что "DLL-платформами" считаются все платформы, основанные на Windows, в том числе и Cygwin.

Компоновка с библиотеками

Команда `target_link_libraries` компоует библиотеку или исполняемый файл с другими предоставляемыми библиотеками. Первым аргументом данная команда принимает название цели, сгенерированной с помощью команд `add_executable` или `add_library`, а последующие аргументы представляют собой названия целей библиотек или полные пути к библиотекам. Пример:

```
# Скомпоновать исполняемый файл "MyExecutable" с  
# библиотеками "JsonParser", "SocketFactory" и "BrowserInvoker":  
target_link_libraries(MyExecutable JsonParser SocketFactory BrowserInvoker)
```

Стоит отметить, что модульные библиотеки не подлежат компоновке с исполняемыми файлами или другими библиотеками, так как они предназначены исключительно для загрузки техниками выполнения.

## Работа с целями

Как упомянули в комментариях, цели в CMake тоже подвержены ручному манипулированию, однако весьма ограниченному.

Имеется возможность управления свойствами целей, предназначенных для задания процесса сборки проекта. Команда `get_target_property` присваивает предоставленной переменной значение свойства цели. Данный пример выводит значение свойства `C_STANDARD` цели `MyTarget` на экран:

```
# Присвоить переменной "VALUE" значение свойства "C_STANDARD":  
get_target_property(VALUE MyTarget C_STANDARD)
```

```
# Вывести значение полученного свойства на экран:  
message("C_STANDARD' property is equal to [${VALUE}]")
```

Команда `set_target_properties` устанавливает указанные свойства целей заданными значениями. Данная команда принимает список целей, для которых будут установлены значения свойств, а затем ключевое слово `PROPERTIES`, после которого следует список вида `<название свойства> <новое значение>`:

```
# Установить свойству 'C_STANDARD' значение "11",  
# а свойству 'C_STANDARD_REQUIRED' значение "ON":  
set_target_properties(MyTarget PROPERTIES C_STANDARD 11  
C_STANDARD_REQUIRED ON)
```

Пример выше задал цели `MyTarget` свойства, влияющие на процесс компиляции, а именно: при компиляции цели `MyTarget` CMake затребует компилятора о использовании стандарта C11. Все известные именованные свойства целей перечисляются на этой странице.

Также имеется возможность проверки ранее определённых целей с помощью конструкции `if(TARGET <TargetName>)`:

```
# Выведет "The target was defined!" если цель "MyTarget" уже определена,  
# а иначе выведет "The target was not defined!":  
if(TARGET MyTarget)  
    message("The target was defined!")  
else()  
    message("The target was not defined!")  
endif()
```

## Добавление подпроектов

Команда `add_subdirectory` побуждает CMake к незамедлительной обработке указанного файла подпроекта. Пример ниже демонстрирует применение описанного механизма:



```
# Добавить каталог "subLibrary" в сборку основного проекта,  
# а генерируемые файлы расположить в каталоге "subLibrary/build":  
add_subdirectory(subLibrary subLibrary/build)
```

В данном примере первым аргументом команды `add_subdirectory` выступает подпроект `subLibrary`, а второй аргумент необязателен и информирует CMake о папке, предназначенной для генерируемых файлов включаемого подпроекта (например, `CMakeCache.txt` и `cmake_install.cmake`).

Стоит отметить, что все переменные из родительской области видимости унаследуются добавленным каталогом, а все переменные, определённые и переопределённые в данном каталоге, будут видимы лишь ему (если ключевое слово `PARENT_SCOPE` не было определено аргументом команды `set`). Данную особенность упомянули в комментариях к предыдущей статье.

## Поиск пакетов

Команда `find_package` находит и загружает настройки внешнего проекта. В большинстве случаев она применяется для последующей линковки внешних библиотек, таких как Boost и GSL. Данный пример вызывает описанную команду для поиска библиотеки GSL и последующей линковки:

```
# Загрузить настройки пакета библиотеки "GSL":  
find_package(GSL 2.5 REQUIRED)  
  
# Скомпоновать исполняемый файл с библиотекой "GSL":  
target_link_libraries(MyExecutable GSL::gsl)  
  
# Уведомить компилятор о каталоге заголовков "GSL":  
target_include_directories(MyExecutable ${GSL_INCLUDE_DIRS})
```

В приведённом выше примере команда `find_package` первым аргументом принимает наименование пакета, а затем требуемую версию. Опция `REQUIRED` требует печати фатальной ошибки и завершения работы CMake, если требуемый пакет не найден. Противоположность — это опция `QUIET`, требующая CMake продолжать свою работу, даже если пакет не был найден.

Далее исполняемый файл `MyExecutable` линкуется с библиотекой GSL командой `target_link_libraries` с помощью переменной `GSL::gsl`, инкапсулирующей расположение уже скомпилированной GSL.

В конце вызывается команда `target_include_directories`, информирующая компилятора о расположении заголовочных файлов библиотеки GSL. Обратите внимание на то, что

используется переменная `GSL_INCLUDE_DIRS`, хранящая местоположение описанных мною заголовков (это пример импортированных настроек пакета).

Вам, вероятно, захочется проверить результат поиска пакета, если Вы указали опцию `QUIET`. Это можно сделать путём проверки переменной `<PackageName>_FOUND`, автоматически определяемой после завершения команды `find_package`. Например, в случае успешного импортирования настроек `GSL` в Ваш проект, переменная `GSL_FOUND` обратится в истину.

В общем случае, команда `find_package` имеет две разновидности запуска: модульную и конфигурационную. Пример выше применял модульную форму. Это означает, что во время вызова команды `CMake` ищет скриптовый файл вида `Find<PackageName>.cmake` в директории `CMAKE_MODULE_PATH`, а затем запускает его и импортирует все необходимые настройки (в данном случае `CMake` запустила стандартный файл `FindGSL.cmake`).

#### Способы включения заголовков

Информировать компилятора о расположении включаемых заголовков можно посредством двух команд: `include_directories` и `target_include_directories`. Вы решаете, какую из них использовать, однако стоит учесть некоторые различия между ними (идея предложена в комментариях).

Команда `include_directories` влияет на область каталога. Это означает, что все директории заголовков, указанные данной командой, будут применяться для всех целей текущего `CMakeLists.txt`, а также для обрабатываемых подпроектов (смотреть `add_subdirectory`).

Команда `target_include_directories` влияет лишь на указанную первым аргументом цель, а на другие цели никакого воздействия не оказывается. Пример ниже демонстрирует разницу между этими двумя командами:

```
add_executable(RequestGenerator RequestGenerator.c)
add_executable(ResponseGenerator ResponseGenerator.c)
```

```
# Применяется лишь для цели "RequestGenerator":
target_include_directories(RequestGenerator headers/specific)
```

```
# Применяется для целей "RequestGenerator" и "ResponseGenerator":
include_directories(headers)
```

В комментариях упомянуто, что в современных проектах применение команд `include_directories` и `link_libraries` является нежелательным. Альтернатива — это команды `target_include_directories` и `target_link_libraries`, действующие лишь на конкретные цели, а не на всю текущую область видимости.

## Установка проекта

Команда `install` генерирует установочные правила для Вашего проекта. Данная команда способна работать с целями, файлами, папками и многим другим. Сперва рассмотрим установку целей.

Для установки целей необходимо первым аргументом описанной функции передать ключевое слово `TARGETS`, за которым должен следовать список устанавливаемых целей, а затем ключевое слово `DESTINATION` с расположением каталога, в который установятся указанные цели. Данный пример демонстрирует типичную установку целей:

```
# Установить цели "TimePrinter" и "DataScanner" в директорию "bin":  
install(TARGETS TimePrinter DataScanner DESTINATION bin)
```

Процесс описания установки файлов аналогичен, за тем исключением, что вместо ключевого слова `TARGETS` следует указать `FILES`. Пример, демонстрирующий установку файлов:

```
# Установить файлы "DataCache.txt" и "MessageLog.txt" в директорию "~/":  
install(FILES DataCache.txt MessageLog.txt DESTINATION ~/)
```

Процесс описания установки папок аналогичен, за тем исключением, что вместо ключевого слова `FILES` следует указать `DIRECTORY`. Важно подметить, что при установке будет копироваться всё содержимое папки, а не только её название. Пример установки папок выглядит следующим образом:

```
# Установить каталоги "MessageCollection" и "CoreFiles" в директорию "~/":  
install(DIRECTORY MessageCollection CoreFiles DESTINATION ~/)
```

После завершения обработки `CMake` всех Ваших файлов Вы можете выполнить установку всех описанных объектов командой `sudo checkinstall` (если `CMake` генерирует `Makefile`), или же выполнить данное действие интегрированной средой разработки, поддерживающей `CMake`.

## Наглядный пример проекта

Данное руководство было бы неполным без демонстрации реального примера использования системы сборки `CMake`. Рассмотрим схему простого проекта, использующего `CMake` в качестве единственной системы сборки:

```
+ MyProject  
  - CMakeLists.txt  
  - Defines.h  
  - StartProgram.c
```

```
+ core
  - CMakeLists.txt
  - Core.h
  - ProcessInvoker.c
  - SystemManager.c
```

Главный файл сборки CMakeLists.txt описывает компиляцию всей программы: сперва происходит вызов команды `add_executable`, компилирующей исполняемый файл, затем вызывается команда `add_subdirectory`, побуждающая обработку подпроекта, и наконец, исполняемый файл линкуется с собранной библиотекой:

```
# Задать минимальную версию CMake:
cmake_minimum_required(VERSION 3.0)

# Указать характеристики проекта:
project(MyProgram VERSION 1.0.0 LANGUAGES C)

# Добавить в сборку исполняемый файл "MyProgram":
add_executable(MyProgram StartProgram.c)

# Требовать обработку файла "core/CMakeFiles.txt":
add_subdirectory(core)

# Скомпоновать исполняемый файл "MyProgram" со
# скомпилированной статической библиотекой "MyProgramCore":
target_link_libraries(MyProgram MyProgramCore)

# Установить исполняемый файл "MyProgram" в директорию "bin":
install(TARGETS MyProgram DESTINATION bin)
```

Файл `core/CMakeLists.txt` вызывается главным файлом сборки и компилирует статическую библиотеку `MyProgramCore`, предназначенную для линковки с исполняемым файлом:

```
# Задать минимальную версию CMake:
cmake_minimum_required(VERSION 3.0)

# Добавить в сборку статическую библиотеку "MyProgramCore":
add_library(MyProgramCore STATIC ProcessInvoker.c SystemManager.c)
```

После череды команд `cmake . && make && sudo checkinstall` работа системы сборки CMake завершается успешно. Первая команда запускает обработку файла `CMakeLists.txt` в корневом каталоге проекта, вторая команда окончательно компилирует необходимые двоичные файлы, а третья команда устанавливает скомпонованный исполняемый файл `MyProgram` в систему.

## Python PIP

`pip` — система управления пакетами, которая используется для установки и управления программными пакетами, написанными на Python. Много пакетов можно найти в Python Package Index (PyPI).[4]

Начиная с версии Python 2.7.9 и Python 3.4, они содержат пакет pip (или pip3 для Python 3) по умолчанию.[5]

Когда вы в первый раз начинаете работу в качестве программиста в Python, вы, скорее всего, не задумываетесь о том, как именно вам нужно установить внешний пакет или модуль. Но когда эта нужда возникает, вам захочется сделать это как можно быстрее! Пакеты Python можно легко найти в интернете. Большая часть популярных пакетов может быть найдена в PyPI (Python Package Index). Также множество пакетов Python можно найти на github, и bitbucket, а также в Google Code. В данной статье мы рассмотрим следующие методы установки пакетов Python:

Установка из источника

easy\_install

pip

Другие способы

Установка из источника

Это отличный навык, которому стоит научиться. Существуют более простые способы, которые мы рассмотрим позже в этой статье. Тем не менее, существует ряд пакетов, которые нужно установить именно этим способом. Например, чтобы использовать easy\_install, вам сначала нужно установить setuptools. Чтобы сделать это, вам нужно скачать tar или zip файл с PyPI (<https://pypi.python.org/pypi/setuptools>), и извлечь его в вашей системе. Далее, обратите внимание на файл setup.py. Откройте сессию терминала и измените каталог на папку, содержащую файл setup. После этого запустите следующую команду:

Python

```
python setup.py install
```

```
1
```

```
python setup.py install
```

Если Python не расположен в пути вашей системы, вы получите сообщение об ошибке, указывающее на то, что команда не была найдена, или приложение неизвестно. Вы также можете вызвать эту команду, применив весь путь к Python. Для пользователей Windows это выглядит так:

Python

```
c:\python34\python.exe setup.py install
```

```
1
```

```
c:\python34\python.exe setup.py install
```

Этот метод особенно удобен, если у вас несколько установленных версий Python и вам нужно установить пакет на разные версии. Все что вам нужно, это указать полный путь к конкретной версии Python и установить пакет. Некоторые пакеты содержат в себе C код, например, заглавные файлы C, которые должны быть скомпилированы под пакет для корректной установки. В случае с Linux, у вас должен быть установленный C/C++ компилятор, так что вы можете установить пакет без головной боли. Возвращаясь к Windows, вам нужна правильная версия Visual Studio, для корректной компиляции пакета. Некоторые люди упоминают также и MingW, который можно использовать для этих целей, но я не знаю, как сделать так, чтобы это работало. Если в пакете присутствует установщик Windows, используйте его. В таком случае вы можете забыть о компиляции в принципе.

## Применение easy\_install

После установки `setuptools`, вы можете использовать `easy_install`. Вы можете найти его в папке с установочными скриптами Python. Не забудьте добавить папку со скриптами в путь вашей системы, чтобы вы в дальнейшем смогли вызывать `easy_install` в командной строке, без указания его полного пути. Попробуйте выполнить запустить следующую команду, чтобы узнать больше об опциях `easy_install`:

Python

```
easy_install -h
```

```
1
```

```
easy_install -h
```

Если вам нужно начать установку пакета при помощи `easy_install`, вам нужно сделать следующее:

Python

```
easy_install package_name
```

```
1
```

```
easy_install package_name
```

`easy_install` попытается скачать пакет с PyPI, скомпилировать его (если нужно) и установить его. Если вы зайдете в свою директорию `site-packages`, вы найдете файл под названием `easy-install.pth`, который содержит доступ ко всем пакетам, установленным через `easy_install`. Python использует этот файл, чтобы помочь в импорте модуля или пакета. Вы также можете указать `easy_install` на установку через URL или через путь на вашем компьютере. `easy_install` также может выполнить установку напрямую из файла `tar`. Вы можете использовать `easy_install` для обновления пакета, воспользовавшись функцией `upgrade` (или `-U`). И наконец, вы можете использовать `easy_install` для установки файла `egg` файлов. Вы можете найти эти файлы в PyPI, или в других источниках. Файлы `egg` – это особые `zip` файлы. На самом деле, если вы измените расширение на `.zip`, вы можете разархивировать файл `egg`.

Мы собрали ТОП Книг для Python программиста которые помогут быстро изучить язык программирования Python. Список книг: Книги по Python  
Вот несколько примеров:

Python

```
easy_install -U SQLAlchemy
```

```
easy_install http://example.com/path/to/MyPackage-1.2.3.tgz
```

```
easy_install /path/to/downloaded/package
```

```
1
```

```
2
```

```
3
```

```
easy_install -U SQLAlchemy
```

```
easy_install http://example.com/path/to/MyPackage-1.2.3.tgz
```

```
easy_install /path/to/downloaded/package
```

Существует несколько проблем с `easy_install`. Он может попробовать установить пакет, пока он еще загружается. При помощи `easy_install` нельзя деинсталлировать пакет. Вам придется удалить пакет вручную и обновить файл `easy-install.pth`, удалив доступ к пакету. По этой, и многим другим причинам, в сообществе Python создали `pip`.

## Использование pip

Программа `pip` вышла вместе с Python 3.4. Если у вас старая версия Python, вам нужно установить `pip` вручную. Установка `pip` немного отличается от того, что мы обсуждали ранее. Вам нужно зайти в PyPI, но вместо того, чтобы скачать пакет и запустить его скрипт `setup.py`, вам предложат скачать один скрипт под названием `get-pip.py`. После этого, вам нужно будет выполнить его следующим образом:

Python

```
python get-pip.py
```

```
1
```

```
python get-pip.py
```

Таким образом `setuptools` (или какая-либо альтернатива `setuptools`) будет установлена, если вы не сделали этого ранее. Также будет установлен и `pip`. Он работает с версиями Python 2.6, 2.7, 3.1, 3.2, 3.3, 3.4 и с `pyru`. Вы можете использовать `pip` для установки всего, что может установить `easy_install`, но сам процесс несколько отличается. Чтобы установить пакет, выполните следующее:

Python

```
pip install package_name
```

```
1
```

```
pip install package_name
```

А для обновления пакета, вам нужно сделать это:

Python

```
pip install -U PackageName
```

```
1
```

```
pip install -U PackageName
```

Вам может понадобиться вызвать `-h`, чтобы получить полный список всего, что `pip` может сделать. В отличие от `easy_install`, `pip` может устанавливать формат `wheel`. Формат `wheel` – это ZIP архив, с именем файла, имеющим особый формат, и расширением. `Whl`. Вы также можете установить `wheel`, при помощи его собственной командной строки. С другой стороны, `pip` не может выполнять установку файлов `egg`. Если вам нужно установить `egg`, вам придется воспользоваться `easy_install`.

### Кое-что о зависимостях

Преимущества в использовании `easy_install` и `pip` в том, что если пакет имеет зависимости от скрипта `setup.py`, то и `easy_install`, и `pip` попытаются установить и скачать этот скрипт. Это поможет вам избежать головной боли в той ситуации, когда вы пробуете новый пакет, и еще не понимаете, что пакет А зависит от пакетов В, С и D. При помощи `easy_install` или `pip`, вам не нужно об этом беспокоиться вообще.

### Подведем итоги

С этого момента, вы можете использовать любой пакет, который вам может понадобиться, конечно, если этот пакет поддерживается вашей версией Python. Существует множество инструментов, доступных программисту Python. Пока что пакеты – достаточно запутанная тема, тем не менее, вы знаете, как использовать необходимые инструменты, которые вы можете легко и быстро получить, для установки необходимого пакета.

## **ЗАДАНИЕ ДЛЯ САМОСТОЯТЕЛЬНО РАБОТЫ**

Необходимо самостоятельно разобраться в дополнительных возможностях системы Stake. Дополнительно изучить возможности интеграции и взаимодействия библиотек, написанных на различных языках с помощью соглашения FFI. Попытаться интегрировать самостоятельно написанную библиотеку на языке C в Python приложение.

## **ЗАДАНИЕ ДЛЯ КОНТРОЛЬНОЙ РАБОТЫ**

Целью контрольной работы является получение опыта в оценке качества графического пользовательского интерфейса программного средства.

### **Задание:**

Разработать приложение на предлагаемом языке разработки ПО (C++, Java, Python), которое использует свободно распространяемые библиотеки (OpenGL, Zlib, PThreads, LibPng, Curl, SQLite и тд). Сформировать поддерживаемую архитектуру для дальнейшего предполагаемого расширения функционала приложения

## **ТРЕБОВАНИЯ К ОТЧЕТУ О ВЫПОЛНЕНИИ КОНТРОЛЬНОЙ РАБОТЫ**

Отчет по выполнению контрольной работы должен содержать:

1. Описание функционала рассматриваемого программного средства в виде списка функций.
2. Демонстрация работающего программного продукта



## СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ И ИНТЕРНЕТ-РЕСУРСОВ

1. <https://cmake.org/>
2. <https://www.sqlite.org/index.html>
3. <https://www.zlib.net/>
4. <http://www.libpng.org/pub/png/libpng.html>
5. <http://man7.org/linux/man-pages/man7/pthreads.7.html>
6. <https://www.php.net/manual/ru/book.pthreads.php>
7. <https://curl.haxx.se/libcurl/>

**Приложение 1. ОБРАЗЕЦ ТИТУЛЬНОГО ЛИСТА**

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ  
ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«МУРМАНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра  
Математики, информационных  
систем и программного  
обеспечения

О Т Ч Е Т  
о выполнении работы  
по дисциплине

«»

Выполнил:  
студент группы \_\_\_\_\_  
Фамилия И.О.

Проверил:  
Старший преподаватель  
кафедры МИС и ПО  
Фамилия И.О.

Мурманск

20\_\_