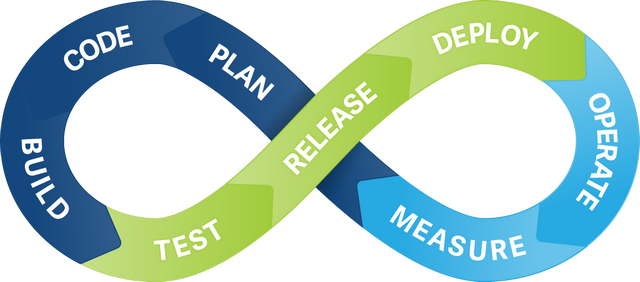
Непрерывная интеграция (CI) для GitHub проектов на С/C++ с CMake-сборкой

* [Тестирование IT-систем](https://habr.com/ru/hub/it_testing/),
* [Программирование](https://habr.com/ru/hub/programming/),
* [C++](https://habr.com/ru/hub/cpp/),
* [GitHub](https://habr.com/ru/hub/github/)
* Tutorial



Про непрерывную интеграцию и её целебные свойства слышали, наверное, все — не буду повторять написанное в многочисленных вводных статьях и обзорах и рассказывать, что же такое непрерывная интеграция, и как именно она упрощает жизнь разработчикам, релиз-инженерам и менеджерам. Предполагается, что читатель и сам прекрасно понимает, что CI — вне зависимости от применяемых в проекте языков программирования — это *стопроцентно правильный подход*, но… Но вот до практического его внедрения руки в силу разных причин пока не дошли.

Данная статья — инструкция по прикручиванию *базовой непрерывной интеграции* (build-test-deploy) к гитхабовским C/C++ проектам с CMake сборкой — по непонятным причинам, на Хабре такого до сих пор не проскальзывало. Впрочем, если моё гугл-фу подвело меня, и таки проскальзывало — не беда. Лишний туториал, описывающий всё под несколько другим углом и предостерегающий от неповторимого набора набитых автором шишек, совершенно точно не повредит.

Выбираем сервисы

Критериев выбора у меня было немного (как я наивно полагал).

* Поддержка основных десктопных платформ:
  + Windows
  + Linux
  + Mac OS X
* Облачный хостинг — чтобы не нужно было поднимать и настраивать свой сервер
* Из-коробочная интеграция с GitHub
* Бесплатность
* Размер и активность пользовательской базы

Вопреки моим ожиданиям, сервиса, в одиночку покрывающего все эти требования, не нашлось. Странно. Ну да ничего страшного — следующей пары сервисов оказывается вполне достаточно:

1) [Travis CI](https://travis-ci.org/) — для Linux и Mac OS X  
2) [AppVeyor](https://appveyor.com/) — для Windows

Общим планом

И Travis CI, и AppVeyor позволяют запускать виртуальные машины с предустановленным инструментарием, на которых будет производиться процесс сборки, тестирования и развёртывания.

Docker-контейнеры в Travis

Как именно собирать, тестировать и развёртывать проект, описывается нами в специальном конфигурационном файле на языке **YAML**. Этот файл должен лежать в корне репозитория и иметь имя .travis.yml или appveyor.yml (допускается .appveyor.yml) — для Travis CI и AppVeyor соответственно. Мы хотим использовать оба сервиса одновременно, а значит и файлов будет два.

Предостережение для тех, кто, никогда не использовал YAML ранее. YAML это язык с синтаксическим структурированием с помощью *отступов* (как и, например, Питон), но при этом *не разрешается использование табуляции*. Поэтому придётся либо настроить ваш текстовый редактор на expand-tabs-to-spaces для .yml файлов, либо не использовать табы вообще и вбивать отступы пробелами.

После того, как YAML файлы добавлены в репозиторий, нужно будет включить непрерывную интеграцию для заданного проекта на сайтах Travis и AppVeyor. Делается это очень просто. Заходим на [https://travis-ci.org](https://travis-ci.org/) под своим GitHub аккаунтом, соглашаемся с доступом, который запрашивает Travis CI (ему нужно будет получать уведомления о новых коммитах), синхронизуем список своих проектов, выбираем нужный и щёлкаем на включатель. Готово. Далее повторяем аналогичный процесс на сайте [https://ci.appveyor.com](https://ci.appveyor.com/)

Начиная с этого момента каждый git push в ваш репозиторий будет запускать процесс непрерывной интеграции: сервисы Travis и AppVeyor поднимут виртуальную машину, настроят среду, установят зависимости, скачают ваш проект, соберут и протестируют его, а также, при желании, выложат инсталляторы, архивы с исходниками и документацию — всё согласно спецификации в YAML-файлах.

Собственно, в создании YAML-файлов и будет заключаться наша основная работа.

Структура .travis.yml

Рассмотрим пример файла .travis.yml с комментариями:

dist: trusty # используем Ubuntu 14.04 Trusty Tahr (а не 12.02 Precise Pangolin)

sudo: required # используем Virtual Machine (а не Docker container)

language: cpp # на практике разницы между специфичным для C++ окружением

# и, скажем, python -- никакой. Но пусть будет cpp.

os:

# будем прогонять CI и на Linux, и на Mac OS X...

- linux

- osx

compiler:

# ... и с помощью GCC, и Clang

- gcc

- clang

env:

# список переменных окружения, влияющих на матрицу сборки:

- TARGET\_CPU=amd64 BUILD\_CONFIGURATION=Debug

- TARGET\_CPU=amd64 BUILD\_CONFIGURATION=Release

- TARGET\_CPU=x86 BUILD\_CONFIGURATION=Debug

- TARGET\_CPU=x86 BUILD\_CONFIGURATION=Release

matrix:

exclude:

# вручную исключим ненужные элементы из матрицы сборки:

- os: osx

compiler: gcc

- os: osx

env: TARGET\_CPU=x86 BUILD\_CONFIGURATION=Debug

- os: osx

env: TARGET\_CPU=x86 BUILD\_CONFIGURATION=Release

install:

# скрипт настройки среды и установки зависимостей:

- source ci/travis/install-$TRAVIS\_OS\_NAME.sh

script:

# скрипт сборки и тестирования проекта:

- mkdir build

- cd build

- cmake .. -DCMAKE\_BUILD\_TYPE=$BUILD\_CONFIGURATION -DTARGET\_CPU=$TARGET\_CPU

- cmake --build .

- ctest --output-on-failure

deploy:

# выкладываем tagged-коммиты на GitHub Releases:

provider: releases

file: <package-file>

skip\_cleanup: true

overwrite: true

api\_key:

secure: <encrypted-github-token>

on:

tags: true

Скрипты этапов установки и развёртывания требуют пояснений — они будут даны далее. Сначала же остановимся на одном из ключевых понятий непрерывной интеграции — **матрице сборки** (build matrix).

Все мы знакомы с ситуацией, когда мы внесли минимальные изменения в проект, собрали, запустили, проверили — всё работает. Сделали релиз, и — БАМ! Он тут же упал на целевой машине. Проверяем у себя ещё раз — работает, собака! В английском языке есть специальный акроним на этот счёт:

**IWOMM — It works on my machine!**

Разница, приводящая к проблеме, может таиться где угодно — разные версии компилятора у разработчика и на сборочной машине, отличающаяся версия библиотеки на целевой, тестировали на одной операционке, а запускали на другой — да мало ли что ещё!

Способ борьбы с этим пакостным феноменом может быть только один — собирать и тестировать с использованием *обширной матрицы сборки*, то есть сразу на *множестве возможных конфигураций*. Это, вкупе с достаточным покрытием исходного кода *автоматическими тестами*, позволяет значительно снизить вероятность IWOMM.

C/C++ проект на Travis CI имеет смысл гонять, как минимум, под следующей матрицей:

* OS
  + Linux
  + Mac OS X
* Compiler
  + GCC
  + Clang
* Target CPU
  + amd64
  + x86
* Configuration
  + Debug
  + Release

К сожалению, Тревис предлагает на выбор всего два дистрибутива Linux — **Ubuntu 12.04 Precise Pangolin** (по умолчанию) и **Ubuntu 14.04 Trusty Tahr**. Даже если отвлечься от отсутствия каких-либо альтернатив Ubuntu, напомню, что текущая LTS (long-term-support) версия Ubuntu — это **Ubuntu 16.04 Yakkety Yak**, а текущая не-LTS версия — **Ubuntu 17.04 Zesty Zapus**. Иными словами, в мае 2017 года нам по умолчанию предлагается версия Ubuntu, выпущенная в 2012 году и отстающая на 5 мажорных релизов. Мда. Ну что ж, будем играть картами, что есть на руках.

Как бы то ни было, данная матрица содержит 16 конфигураций. Выбрасываем отсюда x86 под Mac OS X — целевая машина там однозначно будет 64-битная. Также выбрасываем GCC под Mac OS X — GCC под Маком это не более чем адаптер для GCC-специфичных опций командной строки, реальный же компилятор всё равно будет Clang. Исключение элементов из матрицы сборки осуществляется в секции

matrix:

exclude:

Предостережение: если исключаемые элементы описываются с помощью переменных окружения (как в случае с TARGET\_CPU), то придётся скопировать всю строку целиком:

env: TARGET\_CPU=x86 BUILD\_CONFIGURATION=Debug

Кажущегося логичным env: TARGET\_CPU=x86 будет недостаточно.

Итого, остаётся **12 конфигураций** на каждый git push. Согласитесь, это значительно лучше, чем одна-две, как обычно бывает при тестировании вручную, без использования непрерывной интеграции?

В терминологии Travis CI всё это звучит так. Каждый *коммит* (commit) приводит к запуску *сборки* (build), которая состоит из множества *заданий* (jobs) — по одному на каждую конфигурацию в *матрице сборки* (build matrix).

Структура .appveyor.yml

YAML-файл в AppVeyor выглядит очень похоже с поправкой на имена секций — всё тут называется немного по-другому:

image: Visual Studio 2015 # на этом VM-образе установлены все Visual Studio с 2008 по 2015

init:

# секция инициализации, исполняется до клонирования проекта

# скорее всего, строчка ниже необязательна (так должно быть по умолчанию),

# вместе с тем, она присутствует в официальных примерах, так что пусть будет:

- git config --global core.autocrlf input

clone\_folder: c:\projects\my-prj # выбираем локальную директорию для клонирования

shallow\_clone: true # копируем только последний коммит, без истории (git clone --depth 1)

matrix:

fast\_finish: false # не останавливаемся после возникновения ошибки в каком-то из заданий

platform:

# будем гонять CI на amd64 и x86...

- x64

- x86

configuration:

# ... и в дебажной, и в релизной конфигурациях ...

- Debug

- Release

environment:

matrix:

# ... на трёх студиях (2010, 2012, 2015)

- TOOLCHAIN: msvc10

- TOOLCHAIN: msvc12

- TOOLCHAIN: msvc14

install:

# скрипт настройки среды и установки зависимостей:

- call ci\appveyor\set-env.bat %TOOLCHAIN% %PLATFORM%

- appveyor DownloadFile <url> -FileName <local-file>

- 7z e -y <local-file> -o<local-dir>

- ...

build\_script:

# скрипт сборки проекта:

- mkdir build

- cd build

- cmake .. %CMAKE\_CONFIGURE\_FLAGS%

- cmake --build . %CMAKE\_BUILD\_FLAGS%

test\_script:

# скрипт тестирования проекта:

- ctest -C %CONFIGURATION% --output-on-failure

artifacts:

- path: <local-package-path>

name: package\_name

deploy:

# выкладываем tagged-коммиты на GitHub Releases:

description: '$(APPVEYOR\_REPO\_TAG\_NAME)'

provider: GitHub

auth\_token:

secure: <encrypted-github-token>

artifact: package\_name

force\_update: true

on:

appveyor\_repo\_tag: true

Концепция матрицы сборки в полной мере применима и здесь. Как минимум, стоит прогонять CI для двух архитектур x86/amd64, обе под Debug/Release. Помимо этого, я бы рекомендовал также добавить в матрицу сборки и версию Visual Studio — вы ведь не знаете, какую именно студию использует наугад взятый клонер?

Для использования Visual Studio 2017 нужно подключить отдельный VM образ — на котором, увы, отсутствует большинство библиотек и инструментов, доступных на образе для Visual Studio 2015. Если вам необходимо-таки прогонять CI на VS 2017, добавьте в матрицу сборки:

- TOOLCHAIN: msvc15

APPVEYOR\_BUILD\_WORKER\_IMAGE: Visual Studio 2017

Я же для себя решил ограничиться диапазоном VS 2010 — VS 2015. Возможно, со временем разработчики AppVeyor исправят ситуацию с VS 2017 в лучшую сторону.

* Toolchain
  + Visual Studio 2010
  + Visual Studio 2012
  + Visual Studio 2015
* Target CPU
  + amd64
  + x86
* Configuration
  + Debug
  + Release

В сумме это даёт нам симпатичную матрицу на **12 конфигураций**.

Обратите внимание, что в YAML файлах как для AppVeyor, так и для Travis скрипты install ссылаются на внешние shell-файлы (.sh/.bat) Возникает вопрос: нельзя ли обойтись одними YAML файлами, без вспомогательных скриптов? К сожалению, скорее всего, ответом будет "нет".

Вспомогательные скрипты

Проблема в том, что shell-команды, указанные в YAML файлах, выполняются *построчно*. Нужен if-then-else или for? Будьте добры уместить всё в одну строчку. Понятно, что в теории в одну строку можно утрамбовать всё что угодно. На практике же это применимо только к простейшим случаям. Всю нетривиальную многострочную логику остаётся выносить в отдельные shell-скрипты и вызывать из YAML уже их.

Посему, я для себя организовал всё следующим образом. В проекте имеется папка ci (continuous integration) с двумя подпапками travis и appveyor, в которых лежат shell-скрипты:

ci/

travis/

install-linux.sh

install-osx.sh

...

appveyor/

install.bat

set-env.bat

...

В YAML-файлах же используются только однострочные команды типа:

- source ci/travis/install.sh

Выделим три основных этапа непрерывной интеграции на сервисах Travis и AppVeyor и рассмотрим их более подробно.

* Install — настройка виртуальной среды, установка инструментов для сборки, библиотек и других зависимостей
* Build & Test — сборка и тестирование проекта (как правило, выполняются вместе)
* Deploy — выкладывание собранных пакетов/документации в заданные локации (для примеров я буду использовать GitHub Releases)

Install

Скрипт для установки зависимостей, конечно, будет сильно зависеть от требований конкретного проекта. Ниже даны только общие рекомендации.

Travis со своей доисторической Юбунтой предлагает из коробки **CMake 3.2** и **GCC 4.8.4** — скорее всего, для современных проектов этого окажется недостаточно. Для получения GCC, способного переваривать C++14, в install-linux.sh пишем:

sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test

sudo apt-get update -qq

sudo apt-get install -qq g++-5

sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-5 90

Адекватную версию CMake придётся качать и устанавливать вручную. Добавляем в тот же install-linux.sh:

CMAKE\_VERSION=3.3.2

CMAKE\_VERSION\_DIR=v3.3

CMAKE\_OS=Linux-x86\_64

CMAKE\_TAR=cmake-$CMAKE\_VERSION-$CMAKE\_OS.tar.gz

CMAKE\_URL=http://www.cmake.org/files/$CMAKE\_VERSION\_DIR/$CMAKE\_TAR

CMAKE\_DIR=$(pwd)/cmake-$CMAKE\_VERSION

wget --quiet $CMAKE\_URL

mkdir -p $CMAKE\_DIR

tar --strip-components=1 -xzf $CMAKE\_TAR -C $CMAKE\_DIR

export PATH=$CMAKE\_DIR/bin:$PATH

Для моих проектов CMake версии 3.3 хватает, а вот доступной по умолчанию 3.2 — нет (в 3.3 было сразу несколько критических исправлений). Если для ваших проектов нужно что-то посвежее — соответственным образом поменяйте CMAKE\_VERSION и CMAKE\_VERSION\_DIR.

Для поддержки кросс-компиляции amd64->x86 необходимо также установить multilib:

if [ "$TARGET\_CPU" == "x86" ]; then

sudo dpkg --add-architecture i386

sudo apt-get -qq update

*# устанавливаем 32-битные версии необходимых проекту библиотек*

sudo apt-get install -y liblua5.2-dev:i386

sudo apt-get install -y libusb-1.0:i386

*# ...*

*# g++-multilib ставим в самом конце, после i386-пакетов!*

sudo apt-get install -y g++-5-multilib

fi

И Clang, и CMake под Маком не такие древние, как под Ubuntu, поэтому install-osx.sh у меня состоит только из установки специфичных для проекта зависимостей. Осуществляется это с помощью homebrew. Например, чтобы поставить ragel, добавляем в install-osx.sh:

brew update

brew install ragel

А вот install-скрипт в AppVeyor будет выглядеть более устрашающе — как минимум, нужно правильно настроить CMake-генератор в соответствии с версией Visual Studio и целевой платформой.

Настройка CMake-генератора

Если требуется скачать и установить специфичные для платформы зависимости, это делается с помощью утилиты appveyor, доступной на всех виртуальных машинах сервиса AppVeyor. Например, чтобы установить ragel, необходимо добавить в ci/appveyor/install.bat:

set DOWNLOAD\_DIR=c:\downloads

mkdir %DOWNLOAD\_DIR%

set RAGEL\_DOWNLOAD\_URL=http://downloads.yorickpeterse.com/files/ragel-68-visualstudio2012.7z

mkdir %DOWNLOAD\_DIR%\ragel

appveyor DownloadFile %RAGEL\_DOWNLOAD\_URL% -FileName %DOWNLOAD\_DIR%\ragel\ragel.7z

7z e -y %DOWNLOAD\_DIR%\ragel\ragel.7z -o%DOWNLOAD\_DIR%\ragel

set PATH=%DOWNLOAD\_DIR%\ragel;%PATH%

Также я рекомендую удалить следующие файлы — они не нужны для C++ проектов, но приводят к большому количеству мусорных сообщений в логах сборки:

del "c:\Program Files (x86)\MSBuild\14.0\Microsoft.Common.targets\ImportAfter\Xamarin.Common.targets"

del "c:\Program Files (x86)\MSBuild\4.0\Microsoft.Common.targets\ImportAfter\Xamarin.Common.targets"

Build & Test

Этот этап значительно проще предыдущего — для него, скорее всего, даже не потребуется выделенного shell-скрипта. Единственная, пожалуй, хитрость состоит в настройке кросс-компиляции под Linux на Travis CI. Для поддержки кросс-компиляции amd64->x86 добавьте в корневой CMakeLists.txt следующее:

if ("${TARGET\_CPU}" STREQUAL "amd64")

set (CMAKE\_SIZEOF\_VOID\_P 8)

set\_property (GLOBAL PROPERTY FIND\_LIBRARY\_USE\_LIB64\_PATHS TRUE)

set\_property (GLOBAL PROPERTY FIND\_LIBRARY\_USE\_LIB32\_PATHS FALSE)

elseif ("${TARGET\_CPU}" STREQUAL "x86")

set (CMAKE\_SIZEOF\_VOID\_P 4)

set\_property (GLOBAL PROPERTY FIND\_LIBRARY\_USE\_LIB64\_PATHS FALSE)

set\_property (GLOBAL PROPERTY FIND\_LIBRARY\_USE\_LIB32\_PATHS TRUE)

if (GCC)

set (CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -m32")

set (CMAKE\_C\_FLAGS "${CMAKE\_C\_FLAGS} -m32")

endif ()

else ()

message (FATAL\_ERROR "Unsupported CPU: ${TARGET\_CPU}")

endif ()

Далее действуем, как обычно. Сборка CMake-проектов осуществляется в выделенной папке build в два этапа: *Configure* и *Build*. Внутри travis.yml это будет выглядеть так:

script:

- mkdir build

- cd build

- cmake .. -DCMAKE\_BUILD\_TYPE=$BUILD\_CONFIGURATION -DTARGET\_CPU=$TARGET\_CPU

- cmake --build .

Тестируем проект с использованием **CTest** сразу по окончанию сборки — конечно, данный шаг имеет смысл только в том случае, если ваш проект содержит тесты, добавленные с помощью CMake-команд enable\_testing () и add\_test (...):

- ctest --output-on-failure

Я рекомендую запускать ctest с флажком --output-on-failure — в этом случае при провале теста в логе сборки будет виден выхлоп тестовой программы.

Под AppVeyor всё аналогично, со следующими незначительными отличиями. Во-первых, кросс-компиляция осуществляется не с помощью флажков C++ компилятора, а с помощью выбора нужного CMake-генератора (мы обрабатываем это в set-env.bat). Во вторых, Visual Studio генераторы, в отличие от Unix Makefiles, являются т.н. multi-configuration-генераторами, а это значит, что конфигурация Debug/Release указывается не на этапе CMake-configure, а на этапе Build. Впрочем, всё по-прежнему укладывается в те же лаконичные 4 строчки:

build\_script:

- mkdir build

- cd build

- cmake .. %CMAKE\_CONFIGURE\_FLAGS% # содержит нужный CMake-генератор

- cmake --build . %CMAKE\_BUILD\_FLAGS% # задаёт конфигурацию Debug/Release

# и дополнительные параметры для msbuild.exe

Для тестирования добавляем секцию test\_script:

test\_script:

- ctest -C %CONFIGURATION% --output-on-failure

Обратите внимание, что нужно явно указать конфигурацию Debug/Release с помощью ключа -C.

Ура! Проект собирается и проходит тестирование под **24-мя конфигурациями** на каждый git-push. Скорее бежим добавлять статус-бэйджики в README-файл!

Заходим на страничку проекта на сайте Travis CI по адресу:

https://travis-ci.org/<user-name>/<project-name>

Кликаем на бэйджик build|passing на страничке логов сборки — откроется окошко с линком на динамическое изображение. Копируем линк и вставляем его в README.rst:

.. image:: https://travis-ci.org/<user-name>/<project-name>.svg?branch=master

:target: https://travis-ci.org/<user-name>/<project-name>

Мне послышалось, или кто-то сказал README.md? Ни-ког-да! **reStructuredText** FTW!

Аналогичным образом поступаем с бэйджиком AppVeyor. Линки на бэйджики доступны по адресу:

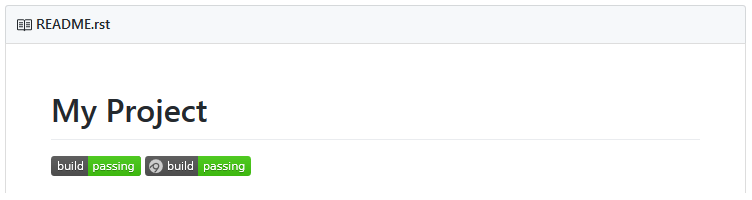
https://ci.appveyor.com/project/<user-name>/<project-name>/settings/badges

Копируем нужную строчку и вставляем рядышком:

.. image:: https://ci.appveyor.com/api/projects/status/<gibberish-code>?svg=true

:target: https://ci.appveyor.com/project/<user-name>/<project-name>

Делаем коммит для обновлённого README.rst и наслаждаемся результатом:



Deploy

Покажем, как задействовать механизмы развёртывания на примере [GitHub Releases](https://help.github.com/articles/creating-releases).

Прежде всего, необходимо авторизовать сервисы Travis и AppVeyor, чтобы они могли создавать релизы в ваших GitHub проектах. Логинимся на GitHub, заходим в Profile->Settings->Developer settings->Personal access tokens. Кликаем на Generate new token, называем токен, например, CI Deploy Token и выставляем единственный флажок public\_repo (разрешить носителю токена доступ к публичным репозиториям). Копируем сгенерированный токен в какой-нибудь временную локацию (вскоре его можно будет удалить).

С использованием этого токена можно как создавать релизы, так и пушить во *все ваши публичные репозитории*. Капитан Очевидность намекает, что выкладывать такой токен во всеобщий доступ в открытом виде нежелательно. Надо зашифровать его так, чтобы ключ для расшифровки был *только у сервисов Travis CI и AppVeyor*.

Чтобы зашифровать токен для Travis, необходимо установить Ruby-утилиту travis:

gem install travis

После этого запускаете в папке проекта:

travis encrypt <github-token>

Утилита зашифрует токен с использованием асимметричного шифрования, так что приватный ключ для расшифровки — созданный в момент подключения проекта на сервисе Travis CI — будет иметься только у самого Travis CI. Выход утилиты можно смело вставлять в .travis.yml.

Аналогичным образом токен асимметрично шифруется и для AppVeyor. Тут, однако, устанавливать никакие вспомогательные утилиты не надо — всё можно сделать прямо с сайта [https://ci.appveyor.com](https://ci.appveyor.com/) через Account->Tools->Encrypt Data.

Далее, надо определиться, какие именно файлы будут выкладываться для всеобщего доступа. В терминах GitHub Releases такие файлы называются **артефактами релиза** (release artifacts). Это могут быть инсталляторы, архивы или же просто исполняемые файлы. Наиболее логичным подходом к генерации архивов или инсталляторов для проекта с CMake-сборкой выглядит использование **CPack**, но, очевидно, что и набор релиз-артефактов, и способ их генерации может разниться от проекта к проекту.

Так или иначе, допустим, в процессе сборки у нас образовались выходные файлы file1.pkg, file2.pkg, и т.д. и мы хотим прикрепить их в качестве release artifcats к релизам нашего GitHub проекта.

В .travis.yml создаём секцию deploy:

deploy:

provider: releases # выкладываем на GitHub Releases

api\_key:

secure: encrypted-github-token # токен, зашифрованный с помощью `travis encrypt`

file:

# список артефактов

- file1.pkg

- file2.pkg

- ...

skip\_cleanup: true # не удалять сгенерированные во время сборки файлы перед развёртыванием

on:

tags: true # выполнять deploy только для tagged-commit

В .appveyor.yml аналогичная функциональность достигается так:

artifacts:

# файлы-артефакты нужно предварительно явно указать в данной секции

# в терминологии AppVeyor это называется "push artifacts"

- path: file1.pkg

- path: file2.pkg

- ...

deploy:

provider: GitHub # выкладываем на GitHub Releases

description: '$(APPVEYOR\_REPO\_TAG\_NAME)' # tag как имя релиза (на Travis так по умолчанию)

auth\_token:

secure: <secret-github-token> # токен, зашифрованный на сайте appveyor.ci

artifact: /.\*\.pkg/ # регулярное выражение, описывающее все артефакты данного релиза

force\_update: true # перезаписывать артефакты, если файл с таким именем уже имеется

on:

appveyor\_repo\_tag: true # выполнять deploy только для tagged-commit

После этого создаём tagged-commit для нового релиза и пушим его:

git add --all

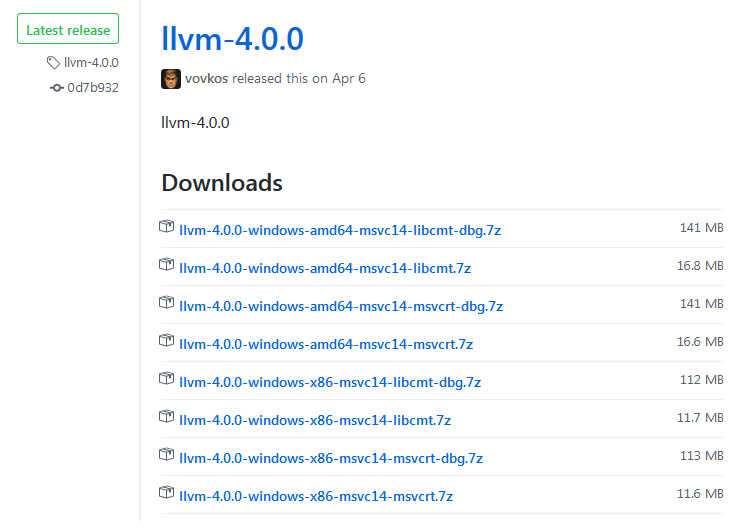
git commit --message "release 1.0.0"

git tag project-name-1.0.0

git push

git push origin project-name-1.0.0

Если сборка, тестирование и развёртывание прошло удачно, то на страничке проекта на GitHub вы должны будете увидеть новый релиз с прикреплёнными артефактами:



Заключение

В данной статье рассмотрен процесс подключения базовой непрерывной интеграции к C++/CMake проектам на GitHub. Разумеется, в рамках короткого туториала невозможно охватить всего множества нюансов и деталей, поэтому считаю своим долгом направить заинтересованных читателей к подробной документации сервисов Travis CI & AppVeyor:

[https://docs.travis-ci.com](https://docs.travis-ci.com/)  
<https://www.appveyor.com/docs>

Также привожу ссылки на собственные проекты, которые демонстрируют всё вышеописанное на практике, и на которых, собственно, и был получен опыт настройки непрерывной интеграции:

<https://github.com/vovkos/jancy>  
<https://github.com/vovkos/doxyrest>  
<https://github.com/vovkos/llvm-package-travis>  
<https://github.com/vovkos/llvm-package-windows>

Буду рад ответить на любые вопросы в комментариях.

**Теги:**

* [cmake](https://habr.com/ru/search/?q=%5Bcmake%5D&target_type=posts)
* [c](https://habr.com/ru/search/?q=%5Bc%5D&target_type=posts)
* [c++](https://habr.com/ru/search/?q=%5Bc%2B%2B%5D&target_type=posts)
* [github](https://habr.com/ru/search/?q=%5Bgithub%5D&target_type=posts)
* [continuous integration](https://habr.com/ru/search/?q=%5Bcontinuous%20integration%5D&target_type=posts)
* [travis ci](https://habr.com/ru/search/?q=%5Btravis%20ci%5D&target_type=posts)
* [appveyor](https://habr.com/ru/search/?q=%5Bappveyor%5D&target_type=posts)

[Реклама](https://tmtm.ru/services/advertising/)

ЧИТАЮТ СЕЙЧАС