CMake: управление проектом v2

Содержание

Полезные ссылки

- Каталог ссылок
- CGold: The Hitchhiker's Guide to the CMake

Инструментарий

В таблице приведён перечень используемых программ.

Таблица 1. Программы

Программа	**Назначение**
GCC	Компилятор C/C++
LLVM	Компилятор C/C++ и средства статического анализа
GDB	Отладчик
gcov	Анализатор покрытия кода
lcov	Генератор отчётов для gcov
Qt Creator	Среда разработки
uncrustify	Форматирование исходных текстов на языке C/C++
git	Система контроля версий
pre-commit	Управление хуками git
CMake	Система управления проектом
cmake-format	Форматирование исходных текстов для CMake
Doxygen	Автоматическая генерация документации

В дистрибутиве Astra Linux Orel 2.12 может присутствовать проблема с загрузкой файлов из-за устаревшей версии пакета с корневыми сертификатами. Решить её можно так:

```
wget http://ftp.ru.debian.org/debian/pool/main/c/ca-certificates/ca-
certificates_20211016_all.deb
sudo dpkg -i ca-certificates_20211016_all.deb
```

При работе в совместимой с Debian операционной системе (Ubuntu, Astra Linux) требуемые программы можно установить командами:

```
sudo apt-get install build-essential gdb clang clang-tidy clang-tools clazy qtcreator
sudo apt-get install qt5-default qttools5-dev qtbase5-private-dev qttools5-dev-tools
sudo apt-get install cmake doxygen lcov git
sudo apt-get install cmake-format uncrustify
```

В дистрибутиве Astra Linux Orel 2.12 нет пакетов `cmake-format` и `uncrustify`. Их можно установить так:

```
wget -A ".deb" -c -q -np -nd -r -l 1
https://deb.246060.ru/astra/orel212/pool/inside/c/cmake-format/
wget -A ".deb" -c -q -np -nd -r -l 1
https://deb.246060.ru/astra/orel212/pool/inside/u/uncrustify/
sudo dpkg -i cmake-format*deb uncrustify*deb
sudo apt-get -f install
```

Для дальней работы потребуется установка пакетов `myx-dev`, в котором находятся скрипты для выполнения типовых команд, и `myx-cmake` с библиотекой MyxCMake, содержащей дополнительные функции для проекта на CMake.

```
sudo apt-get install myx-dev myx-cmake
```

Если пакеты не доступны для `apt-get`, то установить их можно так:

```
wget -A ".deb" -c -q -np -nd -r -l 1
https://deb.246060.ru/ubuntu/focal/pool/main/m/myx-cmake/
wget -A ".deb" -c -q -np -nd -r -l 1
https://deb.246060.ru/ubuntu/focal/pool/main/m/myx-dev/
sudo dpkg -i myx-cmake*deb myx-dev*deb
sudo apt-get -f install
```

Действия, приведённые в данном руководстве, необходимо выполнять в собственном репозитории. Для каждого раздела из этого документа созданы эталонные репозитории, с которыми можно сравнивать свой проект.

Структура каталогов проекта

Файлы проекта и результаты компиляции размещаются в каталогах:

```
└──── project
    ├──── _build
    │    ├──── debug
    │    │    ├──── bin
    │    │    ├──── etc
    │    │    ├──── include
    │    │    ├──── lib
    │    │    ├──── log
    │    │    ├──── share
    │    │    └──── var
    │    └──── release
    ├──── .git
    ├──── cmake
    ├──── doc
    ├──── files
    │    ├──── etc
    │    ├──── log
    │    ├──── share
    │    └──── var
    ├──── l10n
    ├──── src
    │    ├──── app
    │    └──── lib
    ├──── thirdparty
    └──── tools
```

Назначение каталогов приведено в таблице.

Таблица 2. Назначение каталогов

Каталог	**Назначение**
`_build`	Результаты компиляции
`_build/debug`	Результаты компиляции в режиме отладки
`_build/debug/bin`	Исполняемые файлы
`_build/debug/etc`	Символическая ссылка на каталог `files/etc`
`_build/debug/include`	Заголовочные файлы копируемые и генерируемые во время сборки
`_build/debug/lib`	Статические и динамические библиотеки
`_build/debug/log`	Символическая ссылка на каталог `files/log`
`_build/debug/share`	Символическая ссылка на каталог `files/share`
`_build/debug/var`	Символическая ссылка на каталог `files/var`
`_build/release`	Результаты компиляции в режиме выпуска (иерархия аналогична `debug`)
`.git`	Системные файлы репозитория Git

Каталог	Назначение
`cmake`	Файлы с дополнительными функциями для CMake
`doc`	Документация для проекта
`files`	Каталог для дополнительных файлов
`files/etc`	Каталог для файлов настроек проекта
`files/log`	Каталог для журналов
`files/share`	Каталог для неизменяемых файлов
`files/var`	Каталог для изменяемых файлов
`l10n`	Файлы переводов
`src`	Исходные тексты
`src/app`	Исходные тексты программы
`src/lib`	Исходные тексты библиотеки
`thirdparty`	Исходные тексты дополнительных и сторонних проектов
`tools`	Дополнительные утилиты

Каталог `_build` создаётся, чтобы избежать попадания создаваемых во время сборки файлов в иерархию основного проекта. Запись результатов сборки проекта внутрь иерархии каталогов с исходными текстами приводит к засорению формируемыми на этапе сборки файлами, которые затрудняют разработку, поиск в оригинальных файлах и мешают ориентироваться в проекте. Компиляция проекта в отдельном каталоге обеспечивает возможность одновременной работы с несколькими типами сборки, например, отладка и выпуск.

Qt Creator

Настройка программы Qt Creator для работы с описываемыми проектами приведена здесь.

Некоторые описываемые ниже проекты требуют указания ключей для CMake. Это можно сделать в командной строке, добавив к ключу флаг `-D`. Например, чтобы активировать ключ `MYX_CMAKE_CODE_COVERAGE`, нужно к списку параметров командной строки добавить `-DMYX_CMAKE_CODE_COVERAGE=ON`. Также его можно включить в Qt Creator в режиме **Проекты**.

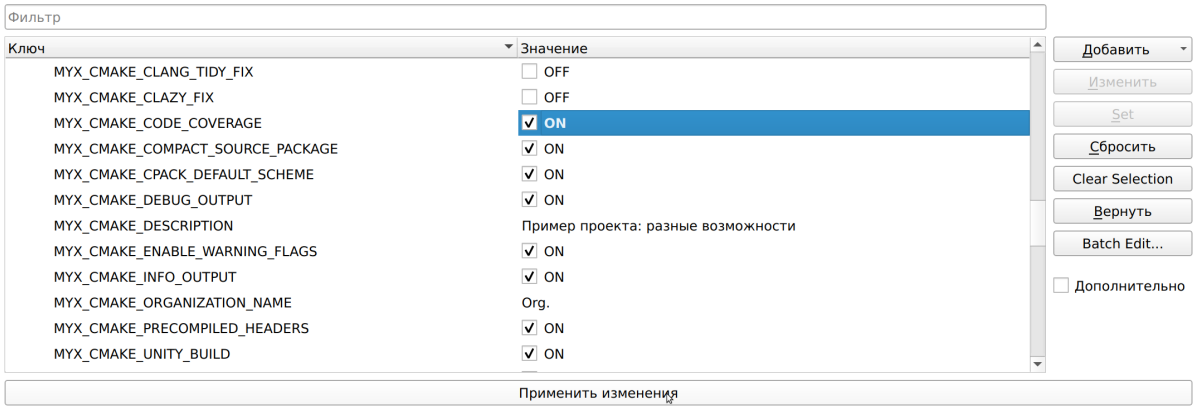

Рисунок 1. Активация ключа `MYX_CMAKE_CODE_COVERAGE` в Qt Creator

Базовый проект

Проект, в котором выполнены приведённые в данном разделе действия, можно посмотреть здесь или сделать его копию командой:

```
git clone --recursive https://git.246060.ru/f1x1t/myx-cmake-example-base
```

Инициализация проекта

Проект создаётся командой `myx-dev-project`. Обязательно должен быть указан параметр `-i` для инициализации проекта, также можно указать git-сервер, с которого будут загружены типовые файлы, и имя каталога для проекта. Например:

```
myx-dev-project -i -s git.246060.ru myx-cmake-example-base
```

Во время инициализации проекта создаются некоторые типовые каталоги:

Файлы `.gitkeep` позволяют защитить каталоги от удаления (будет выводиться дополнительное предупреждение, что каталог не пуст) и обеспечивают возможность помещения каталогов с систему контроля версий Git, в которой пустые каталоги недопустимы (это правильно!).

Эти команды выполнять не нужно.

```
mkdir -p files/{etc,log,share,var}
touch files/{etc,log,share,var}/.gitkeep
```

Автоматически создаётся типовой минимальный файл `CMakeLists.txt`, загружаются файл для форматирования исходных текстов автоматической сборки проекта на сервере GitLab, файл `.clang-tidy` с правилами для анализатора исходных текстов `.clang-tidy` и файлы

сценариев для выполнения автоматических действий в репозитории, которые устанавливаются в каталог `.git/hooks`.

Базовые инструкции в CMake

В корневом каталоге проекта после выполнения команды `myx-dev-project` будет создан файл `CMakeLists.txt`:

```
# Минимальная версия CMake
cmake_minimum_required(VERSION 3.3)

# Предпочтительно следовать стандартам принятым в указанном диапазоне версий
cmake_policy(VERSION 3.0.2..3.7)

# Название и версия проекта и используемые языки программирования
project(myx-cmake-example-base VERSION 0.2.0 LANGUAGES C CXX)
```

Значение версии проекта следует формировать согласно правилам семантического версионирования.

Для подключения функций для CMake из библиотеки MyxCMake, нужно отредактировать в файле `CMakeLists.txt` строки, содержащие обязательные переменные:

```
###
# Обязательные переменные для MyxCMake
###
# Название организации
set(MYX_CMAKE_ORGANIZATION_NAME "Org." CACHE STRING "")

# Имя автора
set(MYX_CMAKE_AUTHOR_NAME "John Doe" CACHE STRING "")

# Почта автора
set(MYX_CMAKE_AUTHOR_EMAIL "mail@johndoe.com" CACHE STRING "")

# Краткое описание проекта
set(MYX_CMAKE_DESCRIPTION "Пример проекта: начало" CACHE STRING "")

find_package(MyxCMake 0.4.1 REQUIRED)
```

Значения обязательных переменных, используемых библиотекой MyxCMake для архивирования исходных текстов, автоматического создания пакетов, генерации документации, следует отредактировать, после чего произведённые изменения можно зафиксировать:

```
git commit -m "Подключение библиотеки MyxCMake" CMakeLists.txt
```

Чтобы проверить корректность подключения МухCMake, можно выполнить команду (пути обязательно без пробелов!):

```
(cmake -B_build -H. && cmake --build _build && echo OK)
```

Если используется версия CMake новее 3.15, можно использовать официально задокументированный набор аргументов для каталогов сборки и корня проекта:

```
(cmake -B _build -S . && cmake --build _build && echo OK)
```

Если последней строкой вывода будет OK, то CMake успешно проверил окружение для сборки

Проверка программного окружения

Поиск программ, библиотек и заголовочных файлов, установленных в системе, можно выполнять с помощью программы pkg-config или функции CMake find_package. В любом случае для указания того, что наличие искомого объекта обязательно для сборки проекта, используется параметр REQUIRED. Если требуемый компонент не будет найден, настройка проекта завершится с ошибкой и для продолжения работы будет необходимо установить недостающие пакеты. Кроме того, можно указывать требования к версии необходимого пакета.

Поиск с помощью программы pkg-config

Программа pkg-config хранит базу данных параметров (обычно в каталогах /usr/share/pkgconfig, /usr/lib/pkgconfig и /usr/lib/x86_64-linux-gnu/pkgconfig), содержащую флаги компиляции для поиска заголовочных файлов и компоновки библиотек, установленных в систему. Для использования в CMake сначала необходимо выполнить проверку наличия программы pkg-config в системе и подключить определённую в модуле PkgConfig функцию pkg_check_modules. Например, для поиска библиотек gsl, fftw3 и udev можно написать в файле CMakeLists.txt:

```
# Поиск библиотек с помощью pkgconfig
find_package(PkgConfig REQUIRED)
pkg_check_modules(GSL gsl REQUIRED)
pkg_check_modules(FFTW3 fftw3>=3.3.2 REQUIRED)
pkg_check_modules(UDEV udev)
```

Если настройка проекта завершается с ошибкой, то нужно установить пакеты:

```
sudo apt-get install pkg-config libgsl-dev libfftw3-dev
```

Поиск с помощью функции `find_package`

Если системная библиотека поставляется без файла описания для `pkg-config` или необходимо произвести более сложный поиск, например, включающий поиск исполняемого файла, то может быть написан специальный модуль для CMake, который вызывается функцией `find_package`. Примеры вызова функции:

```
# Поиск с помощью функции find_package
find_package(LibXml2)
find_package(CURL REQUIRED)
find_package(Boost 1.55.0 REQUIRED)
```

Если настройка проекта завершается с ошибкой, то нужно установить пакеты:

```
sudo apt-get install libxml2-dev curl libcurl4-openssl-dev libboost-all-dev
```

Автоматически генерируемый заголовочный файл

На этапе конфигурирования проекта можно генерировать файлы, в которые будут записаны собранные значения параметров. Функция `myx_cmake_generate_private_config_header()`, из библиотеки MyxCMake создаёт файл `${CMAKE_BINARY_DIR}/include/myx_cmake_private_config_p.hpp`, в который записывается информация о имени и версии проекта, дате и типе сборки.

```
# Автоматически генерируемый заголовочный файл
myx_cmake_generate_private_config_header()
```

Автоматически генерируемый файл о состоянии проекта

Функция `myx_cmake_generate_git_info_header()` библиотеки MyxCMake предоставляет возможность генерировать при каждой сборке проекта файл `${CMAKE_BINARY_DIR}/include/myx_cmake_git_info_p.hpp`, в который записывается информация о теге, текущей ветки и последнем коммите в ней.

```
# Автоматически генерируемый файл с информацией о репозитории
myx_cmake_generate_git_info_header()
```

Примеры библиотек и приложений

Базовая библиотека

Проект с базовой библиотекой реализован на основе базового проекта. Исходные тексты содержат комментарии, объясняющие назначение используемых функций. Проект можно посмотреть здесь или сделать его копию командой:

```
git clone --recursive https://git.246060.ru/f1x1t/myx-cmake-example-library
```

В находящийся в корневом каталоге проекта файл `CMakeLists.txt` нужно записать:

```
# Минимальная версия CMake
cmake_minimum_required(VERSION 3.3)

# Предпочтительно следовать стандартам принятым в указанном диапазоне версий
cmake_policy(VERSION 3.0.2..3.7)

# Название и версия проекта и используемые языки программирования
project(myx-cmake-example-library VERSION 0.2.0 LANGUAGES C CXX)

###
# Обязательные переменные для MyxCMake
###
# Название организации
set(MYX_CMAKE_ORGANIZATION_NAME "Org." CACHE STRING "")

# Имя автора
set(MYX_CMAKE_AUTHOR_NAME "John Doe" CACHE STRING "")

# Почта автора
set(MYX_CMAKE_AUTHOR_EMAIL "mail@johndoe.com" CACHE STRING "")

# Краткое описание проекта
set(MYX_CMAKE_DESCRIPTION "Пример проекта: библиотека" CACHE STRING "")

find_package(MyxCMake 0.4.1 REQUIRED)

# Автоматически генерируемый заголовочный файл
myx_cmake_generate_private_config_header()

# Автоматически генерируемый файл с информацией о репозитории
myx_cmake_generate_git_info_header()

# Исходные тексты библиотеки
add_subdirectory(src/myx-cmake-example-library)
```

В подкаталоге `src/myx-cmake-example-library` нужно создать файл `CMakeLists.txt`:

```
# Название основной цели и имя библиотеки в текущем каталоге
set(TRGT myx-cmake-example-library)

# Список файлов исходных текстов
set(TRGT_cpp ${CMAKE_CURRENT_SOURCE_DIR}/init.cpp)

# Список заголовочных файлов (используется для установки)
set(TRGT_hpp ${CMAKE_CURRENT_SOURCE_DIR}/init.hpp)

# Функция для создания цели, результатом которой будет сборка библиотеки
# Обязательно использовать тип OBJECT
add_library(${TRGT} OBJECT ${TRGT_cpp} ${TRGT_hpp})

# Автоматическая установка значений свойств для цели
myx_cmake_common_target_properties(${TRGT})

# Создание разделяемой библиотеки
myx_cmake_add_shared_library(${TRGT})

# Создание статической библиотеки
myx_cmake_add_static_library(${TRGT})

# Установка заголовочных файлов
install(FILES ${TRGT_hpp} COMPONENT dev DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}/${
TRGT})

# Установка файла для pkg-config
myx_cmake_generate_pkgconfig(${TRGT} COMPONENT dev INSTALL_LIBRARY true)
```

файл `init.hpp`:

```
#ifndef MYX_CMAKE_EXAMPLE_LIBRARY_INIT_HPP_
#define MYX_CMAKE_EXAMPLE_LIBRARY_INIT_HPP_

#pragma once

#include <cstdint>

int32_t init( int32_t v );

#endif // MYX_CMAKE_EXAMPLE_LIBRARY_INIT_HPP_
```

и файл `init.cpp`:

```
#include <myx-cmake-example-library/init.hpp>

#include <cmath>

int32_t init( int32_t v = 0 )
{
    int32_t s = 0;
    for ( auto i = std::abs( v ); i > 0; i-- )
    {
        s += i;
    }
    return( s );
}
```

Базовое приложение

Проект с базовым приложением реализован на основе базового проекта. Исходные тексты содержат комментарии, объясняющие назначение используемых функций. Проект можно посмотреть здесь или сделать его копию командой:

```
git clone --recursive https://git.246060.ru/f1x1t/myx-cmake-example-app
```

В находящийся в корневом каталоге проекта файл `CMakeLists.txt` нужно записать:

```
# Минимальная версия CMake
cmake_minimum_required(VERSION 3.3)

# Предпочтительно следовать стандартам принятым в указанном диапазоне версий
cmake_policy(VERSION 3.0.2..3.7)

# Название и версия проекта и используемые языки программирования
project(myx-cmake-example-app VERSION 0.2.0 LANGUAGES C CXX)

###
# Обязательные переменные для MyxCMake
###
# Название организации
set(MYX_CMAKE_ORGANIZATION_NAME "Org." CACHE STRING "")

# Имя автора
set(MYX_CMAKE_AUTHOR_NAME "John Doe" CACHE STRING "")

# Почта автора
set(MYX_CMAKE_AUTHOR_EMAIL "mail@johndoe.com" CACHE STRING "")

# Краткое описание проекта
set(MYX_CMAKE_DESCRIPTION "Пример проекта: программа" CACHE STRING "")

find_package(MyxCMake 0.4.1 REQUIRED)

# Boost
set(Boost_USE_STATIC_LIBS ON)
set(Boost_USE_MULTITHREADED OFF)
set(Boost_USE_STATIC_RUNTIME ON)
find_package(Boost 1.55.0 REQUIRED)

# Автоматически генерируемый заголовочный файл
myx_cmake_generate_private_config_header()

# Автоматически генерируемый файл с информацией о репозитории
myx_cmake_generate_git_info_header()

# Исходные тексты программы
add_subdirectory(src/myx-cmake-example-app)
```

В подкаталоге `src/myx-cmake-example-app` нужно создать файл `CMakeLists.txt`:

```
# Название основной цели и имени программы в текущем каталоге
set(TRGT myx-cmake-example-app)

# Список файлов исходных текстов
set(TRGT_cpp ${CMAKE_CURRENT_SOURCE_DIR}/main.cpp)

# Функция для создания цели, результатом которой будет сборка приложения
add_executable(${TRGT} ${TRGT_cpp})
myx_cmake_common_target_properties(${TRGT})

# Добавление к пути поиска заголовочных файлов
target_include_directories(${TRGT} SYSTEM PUBLIC ${Boost_INCLUDE_DIRS})

# Правила для установки
install(TARGETS ${TRGT} COMPONENT main RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR})
```

и файл `main.cpp`:

```
#include <myx_cmake_git_info_p.hpp>
#include <myx_cmake_private_config_p.hpp>

#include <iostream>
#include <boost/range/counting_range.hpp>

int32_t nsum( int32_t i = 0 )
{
    int32_t s = 0;
    for ( auto r: boost::counting_range( 1, i ) )
    {
        s += r;
    }
    return( s );
}


int main( int argc, char* argv[] )
{
    // Значение из myx_cmake_private_config.hpp
    std::cout << "Build type:   " << MYX_CMAKE_BUILD_TYPE << std::endl;
    // Значение из myx_cmake_git_info.hpp
    std::cout << "Git revision: " << MYX_CMAKE_EXAMPLE_APP_GIT_REV << std::endl;

    auto s = nsum( argc );
    std::cout << s << std::endl;

    return ( s );
}
```

Подключение внешнего проекта

Проект, использующий для сборки внешний проект, реализован на основе проектов базовой библиотеки и базового приложения. Исходные тексты содержат комментарии, объясняющие назначение используемых функций. Проект можно посмотреть здесь или сделать его копию командой:

```
git clone --recursive https://git.246060.ru/f1x1t/myx-cmake-example-app-ext
```

Для подключения проекта базовой библиотеки нужно выполнить:

```
git submodule add https://git.246060.ru/f1x1t/myx-cmake-example-library
thirdparty/myx-cmake-example-library
git submodule update --init --recursive
```

В находящийся в корневом каталоге проекта файл `CMakeLists.txt` нужно записать:

```
# Минимальная версия CMake
cmake_minimum_required(VERSION 3.3)

# Предпочтительно следовать стандартам принятым в указанном диапазоне версий
cmake_policy(VERSION 3.0.2..3.7)

# Название и версия проекта и используемые языки программирования
project(myx-cmake-example-app-ext VERSION 0.2.0 LANGUAGES C CXX)

###
# Обязательные переменные для MyxCMake
###
# Название организации
set(MYX_CMAKE_ORGANIZATION_NAME "Org." CACHE STRING "")

# Имя автора
set(MYX_CMAKE_AUTHOR_NAME "John Doe" CACHE STRING "")

# Почта автора
set(MYX_CMAKE_AUTHOR_EMAIL "mail@johndoe.com" CACHE STRING "")

# Краткое описание проекта
set(MYX_CMAKE_DESCRIPTION "Пример проекта: программа с поключенной библиотекой" CACHE
STRING "")

find_package(MyxCMake 0.4.1 REQUIRED)

# Подключение внешних проектов
include(ExternalProject)

ExternalProject_Add(extlib ①
  EXCLUDE_FROM_ALL  TRUE
  SOURCE_DIR        ${CMAKE_SOURCE_DIR}/thirdparty/myx-cmake-example-library ②
  INSTALL_DIR       ${CMAKE_BINARY_DIR} ③
  DOWNLOAD_COMMAND  ""
  BUILD_BYPRODUCTS  <INSTALL_DIR>/lib/libmyx-cmake-example-library.a
  CMAKE_ARGS        ${MYX_CMAKE_EXTERNAL_PROJECT_ARGS}
-DBUILD_MYX_CMAKE_EXAMPLE_LIBRARY_SHARED=OFF ④
  )

# Исходные тексты программы
add_subdirectory(src/myx-cmake-example-app-ext)
```

① Название цели, от которой будет зависеть основная программа

② Каталог с внешней библиотекой

③ Каталог, в который устаналиваются результаты сборки внешней библиотеки

④ Аргументы для сборки внешней библиотеки

В результате будет создана цель `extlib`, являющаяся результатом сборки подключённой библиотеки. Все функции `ExternalProject_Add` необходимо располагать перед функциям `add_subdirectories`, чтобы в указанных подкаталогах можно было использовать добавленные цели для определения зависимостей.

В файле `src/myx-cmake-example-app-ext/CMakeLists.txt` после создания цели `${TRGT}` нужно подключить внешний проект `extlib`:

```
# Название основной цели и имени программы в текущем каталоге
set(TRGT myx-cmake-example-app-ext)

# Список файлов исходных текстов
set(TRGT_cpp ${CMAKE_CURRENT_SOURCE_DIR}/main.cpp)

# Функция для создания цели, результатом которой будет сборка приложения
add_executable(${TRGT} ${TRGT_cpp})
myx_cmake_common_target_properties(${TRGT})

# Зависимость от библиотеки из внешнего проекта
add_dependencies(${TRGT} extlib)

# Компоновка с библиотекой из внешнего проекта
target_link_libraries(${TRGT} myx-cmake-example-library)
```

Для проверки работоспособности в файле `src/myx-cmake-example-app-ext/main.cpp` нужно вызвать функцию `init` из библиотеки, предоставляемой внешним проектом. Например:

```
#include <myx-cmake-example-library/init.hpp>

#include <iostream>

int main( int argc, char* argv[] )
{
    auto s = init( argc );
    std::cout << "Value: " << s << std::endl;

    return ( 0 );
}
```

Qt5

В данном разделе будут приведены примеры создания консольного и графического приложений, а также подключения локализации, вызовы препроцессоров `moc`, `uic` и `rcc`.

Консольное приложение

Пример консольного приложения на Qt5 с поддержкой локализации основан на проекте базового приложения. Исходные тексты содержат комментарии, объясняющие назначение используемых функций. Проект можно посмотреть здесь или сделать его копию командой:

```
git clone --recursive https://git.246060.ru/f1x1t/myx-cmake-example-qt5-console
```

В находящийся в корневом каталоге проекта файл `CMakeLists.txt` нужно записать:

```
# Минимальная версия CMake
cmake_minimum_required(VERSION 3.3)

# Предпочтительно следовать стандартам принятым в указанном диапазоне версий
cmake_policy(VERSION 3.0.2..3.7)

# Название и версия проекта и используемые языки программирования
project(myx-cmake-example-qt5-console VERSION 0.2.0 LANGUAGES C CXX)

###
# Обязательные переменные для MyxCMake
###
# Название организации
set(MYX_CMAKE_ORGANIZATION_NAME "Org." CACHE STRING "")

# Имя автора
set(MYX_CMAKE_AUTHOR_NAME "John Doe" CACHE STRING "")

# Почта автора
set(MYX_CMAKE_AUTHOR_EMAIL "mail@johndoe.com" CACHE STRING "")

# Краткое описание проекта
set(MYX_CMAKE_DESCRIPTION "Пример проекта: консольная программа Qt5" CACHE STRING "")

find_package(MyxCMake 0.4.1 REQUIRED)

# Qt5
find_package(Qt5 COMPONENTS Core REQUIRED) ①
find_package(Qt5Core COMPONENTS Private REQUIRED) ②

# Исходные тексты программы
add_subdirectory(src/myx-cmake-example-qt5-console)
```

① Поиск необходимых компонентов Qt5

② Поиск приватных заголовочных файлов из пакета `qtbase5-private-dev`

В подкаталоге `src/myx-cmake-example-qt5-console` нужно создать файл `CMakeLists.txt`:

```
# Название основной цели и имени программы в текущем каталоге
set(TRGT myx-cmake-example-qt5-console)

# Список файлов исходных текстов
set(TRGT_cpp ${CMAKE_CURRENT_SOURCE_DIR}/main.cpp)

# Функция для создания цели, результатом которой будет сборка приложения
add_executable(${TRGT} ${TRGT_cpp})
myx_cmake_common_target_properties(${TRGT})

# Qt5: подключение заголовочных файлов
target_include_directories(${TRGT} SYSTEM PUBLIC ${Qt5Core_INCLUDE_DIRS})
target_include_directories(${TRGT} SYSTEM PUBLIC ${Qt5Core_PRIVATE_INCLUDE_DIRS})

# Qt5: подключение библиотек
target_link_libraries(${TRGT} Qt5::Core)
```

Фалй `myx-cmake-example-qt5-console/main.cpp`:

```
#include <QCoreApplication>
#include <QDebug>
#include <QtCore/private/minimum-linux_p.h>

int main( int argc, char** argv )
{
    QCoreApplication app( argc, argv );

    qDebug() << "Qt5";
    qDebug() << "Min Linux: " << MINLINUX_MAJOR << "." << MINLINUX_MINOR << "." <<
MINLINUX_PATCH;

    return( 0 );
}
```

Графическое приложение, файлы описания ресурсов и интерфейсов

Пример приложения на Qt5 с использованием графического интерфейса основан на проекте консольного приложения для Qt5. Исходные тексты содержат комментарии, объясняющие назначение используемых функций. Проект можно посмотреть здесь или сделать его копию командой:

```
git clone --recursive https://git.246060.ru/f1x1t/myx-cmake-example-qt5-gui
```

В находящийся в корневом каталоге проекта файл `CMakeLists.txt` нужно записать:

```
# Минимальная версия CMake
cmake_minimum_required(VERSION 3.3)

# Предпочтительно следовать стандартам принятым в указанном диапазоне версий
cmake_policy(VERSION 3.0.2..3.7)

# Название и версия проекта и используемые языки программирования
project(myx-cmake-example-qt5-gui VERSION 0.2.0 LANGUAGES CXX)

###
# Обязательные переменные для MyxCMake
###
# Название организации
set(MYX_CMAKE_ORGANIZATION_NAME "Org." CACHE STRING "")

# Имя автора
set(MYX_CMAKE_AUTHOR_NAME "John Doe" CACHE STRING "")

# Почта автора
set(MYX_CMAKE_AUTHOR_EMAIL "mail@johndoe.com" CACHE STRING "")

# Краткое описание проекта
set(MYX_CMAKE_DESCRIPTION "Пример проекта: графическая программа Qt5" CACHE STRING "")

find_package(MyxCMake 0.4.1 REQUIRED)

# Qt5
find_package(Qt5Core COMPONENTS Private REQUIRED)
find_package(Qt5 COMPONENTS Core Gui Widgets REQUIRED)

# Автоматически генерируемый заголовочный файл
myx_cmake_generate_private_config_header()

# Исходные тексты программы
add_subdirectory(src/myx-cmake-example-qt5-gui)
```

В подкаталоге `src/myx-cmake-example-qt5-gui` нужно создать файл `CMakeLists.txt`:

```
# Название основной цели и имени программы в текущем каталоге
set(TRGT myx-cmake-example-qt5-gui)

# cmake-format: off
###
# Списки файлов проекта
###
# Исходные коды
set(TRGT_cpp
  ${CMAKE_CURRENT_SOURCE_DIR}/main.cpp
  ${CMAKE_CURRENT_SOURCE_DIR}/test_window.cpp)
```

```
# Заголовочные файлы, для которых необходима обработка препроцессором moc
# (содержат класс, унаследованный от QObject, использующий сигналы и/или слоты)
set(TRGT_moc_hpp
  ${CMAKE_CURRENT_SOURCE_DIR}/test_window.hpp)

# Другие заголовочные файлы
set(TRGT_hpp)

# Файлы с описанием графического интерфейса для Qt
set(TRGT_ui
  ${CMAKE_CURRENT_SOURCE_DIR}/test_window.ui)

# Файлы описания ресурсов, включаемых в исполняемый файл
set(TRGT_qrc
  ${CMAKE_SOURCE_DIR}/files/share/icon.qrc)
###
# Конец списков файлов
###
# cmake-format: on

set(TRGT_headers ${TRGT_hpp} ${TRGT_moc_hpp})

# Правило для автоматической генерации препроцессором uic
qt5_wrap_ui(TRGT_ui_h ${TRGT_ui})

# Правило для автоматической генерации препроцессором moc
qt5_wrap_cpp(TRGT_moc_cpp ${TRGT_moc_hpp})

# Поиск строк для локализации в файлах, перечисленных в ${TRGT_cpp} ${TRGT_ui}
# Создание и обновление файлов переводов в каталоге ${CMAKE_SOURCE_DIR}/l10n
# Интеграция переводов в исполняемый файл для подключения классом QTranslator
myx_cmake_qt5_translation(TRGT_qrc_cpp
  OUTPUT_DIR ${CMAKE_SOURCE_DIR}/l10n
  BASE_NAME ${TRGT}
  SOURCES ${TRGT_cpp} ${TRGT_ui}
  LANGUAGES ru_RU)

# Правило для автоматической генерации препроцессором qrc
# (обязательно после вызова функции qt5_translation, если она есть,
# так как она инициализирует переменную со списком ресурсов)
qt5_add_resources(TRGT_qrc_cpp ${TRGT_qrc})

# Функция для создания цели, результатом которой будет сборка приложения
add_executable(${TRGT} ${TRGT_headers} ${TRGT_ui_h} ${TRGT_moc_cpp} ${TRGT_qrc_cpp}
${TRGT_cpp})
myx_cmake_common_target_properties(${TRGT})

# Qt5: подключение заголовочных файлов
target_include_directories(${TRGT} SYSTEM PUBLIC ${Qt5Core_INCLUDE_DIRS})
target_include_directories(${TRGT} SYSTEM PUBLIC ${Qt5Gui_INCLUDE_DIRS})
```

```
target_include_directories(${TRGT} SYSTEM PUBLIC ${Qt5Widgets_INCLUDE_DIRS})

# Qt5: подключение библиотек
target_link_libraries(${TRGT} Qt5::Core Qt5::Gui Qt5::Widgets)
```

В каталоге `files/share` создать файл описания включаемых ресурсов `icon.qrc`:

```
<RCC>
  <qresource prefix="/icon">
    <file alias="icon.png">icon.png</file>
  </qresource>
</RCC>
```

и загрузить файл иконки:

```
wget https://git.246060.ru/f1x1t/myx-cmake-example-qt5-
gui/raw/branch/master/files/share/icon.png
```

Для графического приложения нужно создать файл описания интерфейса `src/myx-cmake-example-qt5-gui/test_window.ui`:

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>TestWindow</class>
 <widget class="QMainWindow" name="TestWindow">
  <property name="geometry">
   <rect><x>0</x><y>0</y><width>413</width><height>253</height></rect>
  </property>
  <property name="windowTitle">
   <string>Test Window</string>
  </property>
  <widget class="QWidget" name="centralwidget">
   <widget class="QPushButton" name="exitButton">
    <property name="geometry">
     <rect><x>170</x><y>30</y><width>80</width><height>26</height></rect>
    </property>
    <property name="text">
     <string>Press me</string>
    </property>
   </widget>
  </widget>
 </widget>
 <resources/>
 <connections/>
</ui>
```

заголовочный файл src/myx-cmake-example-qt5-gui/test_window.hpp:

```
#ifndef TEST_WINDOW_HPP_
#define TEST_WINDOW_HPP_

#pragma once

#include "ui_test_window.h"

#include <QMainWindow>

class TestWindow : public QMainWindow, private Ui::TestWindow
{
    Q_OBJECT
public:
    TestWindow( QMainWindow* parent = nullptr );
    virtual ~TestWindow();
};

#endif /* TEST_WINDOW_HPP_ */
```

и файл с реализацией конструктора, в котором проводится инициализация графических элементов, src/myx-cmake-example-qt5-gui/test_window.cpp:

```
#include "test_window.hpp"

TestWindow::TestWindow( QMainWindow* parent ) :
    QMainWindow   ( parent ),
    Ui::TestWindow()
{
    setupUi( this );
}

TestWindow::~TestWindow() = default;
```

Для отображения графического окна нужно создать файл `src/myx-cmake-example-qt5-gui/main.cpp`:

```
#include "myx_cmake_private_config_p.hpp"
#include "test_window.hpp"

#include <QApplication>
#include <QIcon>
#include <QTranslator>

int main( int argc, char** argv )
{
    QApplication app( argc, argv );

    // Подключение переводов в зависимости от текущей локали
    auto* translator = new QTranslator( QApplication::instance() );
    if ( translator->load( QLocale(), MYX_CMAKE_PROJECT_NAME, QStringLiteral( "_" ),
QStringLiteral( ":/qm" ) ) )
    {
        QApplication::installTranslator( translator );
    }

    // Установка иконки для программы
    QApplication::setWindowIcon( QIcon( ":/icon/icon.png" ) );

    // Создание и отображение главного окна
    auto* w = new TestWindow();
    w->show();
    return( QApplication::exec() );
}
```

Для работы с файлами переводов создаётся цель `l10n`, которую нужно вызывать при появлении новых строк. После первого выполнения команды `make l10n` в корневом каталоге проекта будет создан подкаталог `l10n`, в котором появится файл с расширением `.ts`. Его можно открыть и отредактировать программой `linguist`, входящей в состав пакета `qttools5-dev-tools`. При следующей сборке программы переводы будут интегрированы в исполняемый файл. В примере показано, как экземпляр класса `QTranslator` загружает файл переводов и подключает его для отображения строк в программе.

Дополнительные возможности

Библиотека MyxCMake содержит шаблонные функции для использования в программных проектах. Цели для автоматического форматирования и статического анализа исходных текстов создаются автоматически при вызове функции `myx_cmake_common_target_properties()` из библиотеки MyxCMake. Такие функции как анализ покрытия, динамический анализ, сборка из единого компилируемого файла можно подключить только после перечисления подключаемых библиотек. Для решения этой задачи используется функция `myx_cmake_common_target_properties_post_link()`, которую необходимо вызывать после всех

вызовов функции `target_link_libraries()`.

Пример проекта, демонстрирующего перечисленные ниже возможности можно посмотреть здесь или сделать его копию командой:

```
git clone --recursive https://git.246060.ru/f1x1t/myx-cmake-example-features
```

Форматирование исходных текстов

Функция `myx_cmake_common_target_properties()` создаёт для файлов, формирующих цель, дополнительные цели `${target}-format-sources-uncrustify` для форматирования исходных текстов на языке C++ в соответствии с правилами, перечисленными в файле `.uncrustify.cfg`, находящимся в корне проекта, а также `${target}-format-sources-dos2unix` для преобразования переводов строк в файлах к стандарту, принятому в Unix. Для проекта создаётся цель `myx-cmake-format-sources`, которая объединяет все цели, выполняющие форматирование исходных текстов.

Настройка правил форматирования помогает другим разработчикам придерживаться вашего стиля программирования и отправлять изменения в ваш проект в формате, который удобен вам. Проявите заботу о своих коллегах и своём проекте!

Чтобы выполнить форматирование, нужно в каталоге `${CMAKE_BINARY_DIR}` выполнить команду:

```
make myx-cmake-format-sources
```

При создании типового проекта командой `myx-dev-project` к локальному репозиторию подключается обработчик, который автоматически проверяет исходные тексты на соответствие стандарту форматирования перед выполнением фиксации (`pre-commit`). Таким образом в репозитории будут сохраняться исходные тексты, соответствующие принятым правилам форматирования.

Статический анализ исходных кодов

Для работы с программами на языке C++ используются утилиты, выполняющие статический анализ кода и генерирующие отчёты, помогающие программисту находить и устранять ошибки. Эти программы применяют методы, позволяющие в синтаксически корректном коде находить недостатки или ошибки, которые пропускает компилятор, ценой продолжительного анализа исходных текстов.

Библиотека MyxCMake поддерживает анализаторы clazy, Clang Tidy, Clang Static Analyzer и PVS-Studio.

clazy

Функция `myx_cmake_common_target_properties()` создаёт для файлов исходных текстов, формирующих цель, дополнительную цель `${target}-analyze-clazy` для проверки исходных текстов анализатором `clang`. Для всего проекта создаётся цель `myx-cmake-analyze-clazy`, которая выполняет все цели для анализатора.

Чтобы выполнить статический анализ, нужно в каталоге `${CMAKE_BINARY_DIR}` выполнить команду:

```
make myx-cmake-analyze-clazy
```

Clang Static Analyzer

Функция `myx_cmake_common_target_properties()` создаёт для файлов исходных текстов, формирующих цель, дополнительную цель `${target}-analyze-clang-check` для проверки исходных текстов анализатором `clang-check`. Для всего проекта создаётся цель `myx-cmake-analyze-clang-check`, которая выполняет все цели для анализатора.

Чтобы выполнить статический анализ, нужно в каталоге `${CMAKE_BINARY_DIR}` выполнить команду:

```
make myx-cmake-analyze-clang-check
```

Clang Tidy

Функция `myx_cmake_common_target_properties()` создаёт для файлов исходных текстов, формирующих цель, дополнительную цель `${target}-analyze-clang-tidy` для проверки исходных текстов анализатором `clang-tidy`. Для всего проекта создаётся цель `myx-cmake-analyze-clang-tidy`, которая выполняет все цели для анализатора.

Чтобы выполнить статический анализ, нужно в каталоге `${CMAKE_BINARY_DIR}` выполнить команду:

```
make myx-cmake-analyze-clang-tidy
```

PVS-Studio

Функция `myx_cmake_common_target_properties()` создаёт для всего проекта цель `myx-cmake-analyze-clang-tidy` для проверки исходных текстов анализатором `pvs-studio-analyzer`.

Чтобы выполнить статический анализ, нужно в каталоге `${CMAKE_BINARY_DIR}` выполнить команду:

```
make myx-cmake-analyze-pvs-studio
```

Автоматическое исправление кода

Редактирование кода в автоматическом режиме может приводить к его неработоспособности, хотя это и маловероятно. Перед выполнением действий, приведённых в данном раздела, желательно фиксировать текущее состояние в репозитории или делать резервную копию.

clazy

Программа clazy может преобразовывать в программах, использующих Qt, подключения сигналов и слотов старого типа, производить замену старых ключевых слов, подставлять оптимизированные способы для инициализации строк, исправлять циклы и передачу аргументов в функции для избежания лишних копирований.

Для включения автоматического исправления нужно в настройках сборки проекта **Проекты › Настройки сборки** выбрать цель `myx-cmake-analyze-clazy`:

Рисунок 2. Выбор цели

Затем в перечне опций включить `MYX_CMAKE_CLAZY_FIX`, нажать кнопку **[Применить изменения]**, а затем скомпилировать проект `Ctrl` + `B`:

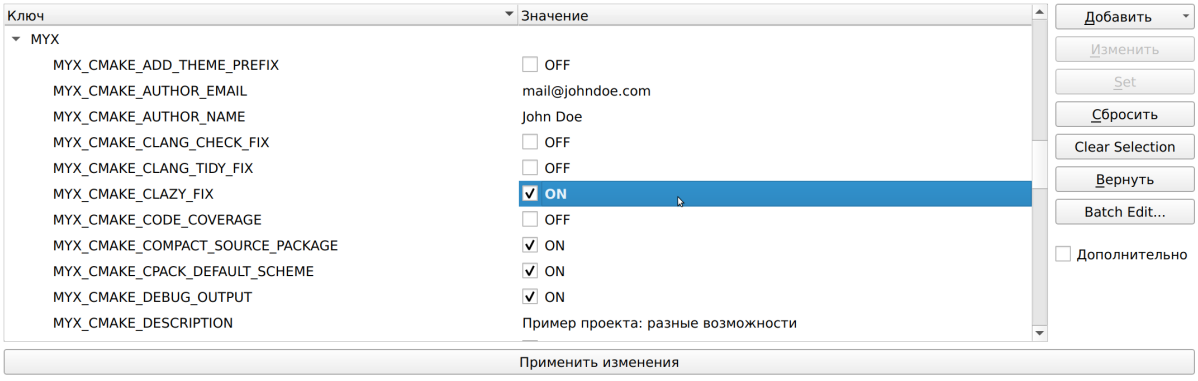

Рисунок 3. Разрешение автозамены

Результат автоматической правки исходных текстов можно посмотреть с помощью git (`git diff`).

Clang-Tidy

Анализатор Clang-Tidy предоставляет более широкие возможности по автоматической правке кода. В проектах, использующих Qt, желательно использовать Clang-Tidy после clazy.

Для включения автоматического исправления нужно в настройках сборки проекта **Проекты › Настройки сборки** выбрать цель `myx-cmake-analyze-clang-tidy`:

Рисунок 4. Выбор цели

Затем в перечне опций включить `MYX_CMAKE_CLANG_TIDY_FIX`, нажать кнопку **[Применить изменения]**, а затем скомпилировать проект `Ctrl` + `B`:

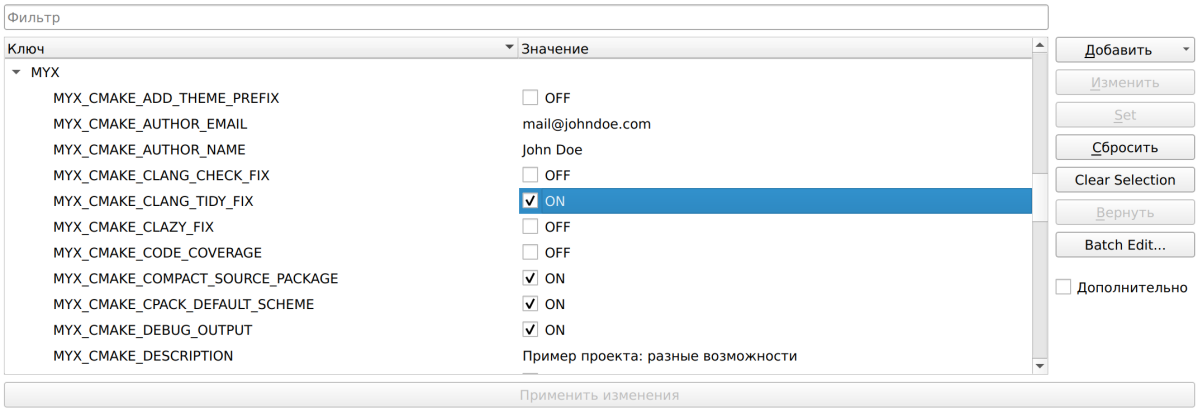

Рисунок 5. Разрешение автозамены

Результат автоматической правки исходных текстов можно посмотреть с помощью git (`git diff`).

Динамический анализ программы

Динамический анализ программы позволяет ценой значительного замедления скорости работы получить дополнительную информацию о ходе её выполнения. Современные компиляторы делают вставку инструкций в определённые точки программы, во время работы программы в них собирается необходимая информация, а по её завершению предоставляется отчёт. Основная информация о работе таких анализаторов находится здесь.

Для обеспечения возможности подключения динамического анализа к проекту нужно выполнить функцию `myx_cmake_common_target_properties_post_link()` (обязательно после подключения всех библиотек):

```
# Дополнительные функции для цели ${TRGT}.
# Вызов обязательно после всех функций target_link_libraries
myx_cmake_common_target_properties_post_link(${TRGT})
```

Подключение анализатора осуществляется включением опций при запуске CMake для генерации сборочных файлов. Некоторые из опций между собой несовместимы, в случае попытки совместного использования будет выведено сообщение об ошибке.

Таблица 3. Назначение опций для динамического анализа

Опция	**Назначение**
`SANITIZE_ADDRESS`	Определение ошибок при работе с памятью: использование после освобождения, использование за пределами области видимости, переполнения буферов в стеке, на куче, в общей памяти, утечки памяти, нарушение порядка инициализации
`SANITIZE_CFI`	Определение нарушений путей исполнения инструкций программы
`SANITIZE_LEAK`	Определение утечек памяти
`SANITIZE_LINK_STATIC`	Статическая компоновка анализатора с программой
`SANITIZE_MEMORY`	Определение попыток доступа к неинициализированным областям памяти
`SANITIZE_SS`	Определение переполнения буфера стека
`SANITIZE_THREAD`	Определение состояние гонок
`SANITIZE_UNDEFINED`	Определение невыровненных и нулевых указателей, переполнения знаковых целых, преобразования типов с плавающей точкой, ведущих к переполнению результирующей переменной

Для проверки возможности динамической отладки можно в перечне опций включить `SANITIZE_ADDRESS`, нажать кнопку **[Применить изменения]**, а затем скомпилировать проект `Ctrl` + `B`. После запуска программы `myx-cmake-example-features` будет выведено сообщение об утечке памяти, показывающее, что объект типа `QFile` не был удалён.

```
==322360==ERROR: LeakSanitizer: detected memory leaks

Indirect leak of 256 byte(s) in 1 object(s) allocated from:
    #0 0x7fbe8558c947 in operator new(unsigned long) (/usr/lib/x86_64-linux-
gnu/libasan.so.5+0x10f947)
    #1 0x7fbe850efeb4 in QFile::QFile() (/usr/lib/x86_64-linux-
gnu/libQt5Core.so.5+0x1bbeb4)
    #2 0x7fbe84a1d0b2 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x270b2)

Indirect leak of 16 byte(s) in 1 object(s) allocated from:
    #0 0x7fbe8558c947 in operator new(unsigned long) (/usr/lib/x86_64-linux-
gnu/libasan.so.5+0x10f947)
    #1 0x56089352a542 in main ../../src/myx-cmake-example-features/main.cpp:41
    #2 0x7fbe84a1d0b2 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x270b2)

SUMMARY: AddressSanitizer: 272 byte(s) leaked in 2 allocation(s).
```

Анализ покрытия кода

Для сбора информации о точном количестве исполнений для каждого оператора в программе используется программа Gcov, входящая в состав компилятора GCC. Для генерирования отчётов в виде HTML-страниц используется программа `lcov`.

Для обеспечения возможности подключения анализа покрытия кода к проекту нужно выполнить функцию `myx_cmake_common_target_properties_post_link()` (обязательно после подключения всех библиотек):

```
# Дополнительные функции для цели ${TRGT}.
# Вызов обязательно после всех функций target_link_libraries
myx_cmake_common_target_properties_post_link(${TRGT})
```

Анализ покрытия кода работает только для файлов, скомпилированных GCC. Во избежание ошибок его можно явно включить, используя при вызове CMake флаги `-DCMAKE_C_COMPILER=gcc` и `-DCMAKE_CXX_COMPILER=g++`.

Подключение осуществляется включением опции `MYX_CMAKE_CODE_COVERAGE` при запуске CMake для генерации сборочных файлов. В результате будут созданы две дополнительные цели `${TRGT}-coverage` для сбора статистики после работы программы и `${TRGT}-coverage-report` для её вывода в виде HTML-страниц.

Пример анализа покрытия кода для проекта `myx-cmake-example-features`:

```
cmake -B_build/cov -H. -DMYX_CMAKE_CODE_COVERAGE=ON -DCMAKE_BUILD_TYPE=Debug
cd _build/cov
make
bin/myx-cmake-example-features
make myx-cmake-example-features-coverage
make myx-cmake-example-features-coverage-report
```

После выполнения этих команд в каталоге `myx-cmake-example-features-coverage-html` будет сформирован отчёт в виде HTML-страниц, в котором будет показано, что функция `int unused(int)` не вызывалась.

Профилирование кода

Библиотека MyxCMake предоставляет вариант сборки для профилирования кода, для которого можно сгенерировать сборочные файлы, присвоив переменной `CMAKE_BUILD_TYPE` значение `Profile`:

```
cmake -B_build/profile -H. -DCMAKE_BUILD_TYPE=Profile
cd _build/profile
make
bin/myx-cmake-example-features
```

По окончании работы исполняемого файла будет сгенерирован файл `gmon.out`, по данным из которого можно строить отчёты утилитой `gprof`. Например:

```
gprof -b bin/myx-cmake-example-features gmon.out > profiling-tree.txt
gprof -b -p bin/myx-cmake-example-features gmon.out > profiling-flat.txt
```

Результаты профилирования будут записаны в файлы `profiling-tree.txt` и `profiling-flat.txt`.

Ускорение компиляции

Для ускорения компиляции используется сторонний модуль cotire, который автоматизирует использование предварительно откомпилированных заголовочных файлов и организует пакетный режим обработки исходных файлов. Аналогичные функции встроены в CMake, начиная с версии 3.16.

Для обеспечения возможностей, предоставляемых модулем cotire, нужно выполнить функцию `myx_cmake_common_target_properties_post_link()` (обязательно после подключения всех библиотек):

```
# Дополнительные функции для цели ${TRGT}.
# Вызов обязательно после всех функций target_link_libraries
myx_cmake_common_target_properties_post_link(${TRGT})
```

FIXME

Документирование кода

Для документирования кода используются блоки комментариев, оформленные для обработки программой Doxygen. Установка программы:

```
sudo apt-get install doxygen
```

Пример комментариев для исходных текстов можно посмотреть в проекте `myx-cmake-example-features`. Поддержка автоматической генерации документации реализована в функции библиотеки MyxCMake `myx_cmake_doc_doxygen()`, которую необходимо вызвать в основном файле `CMakeLists.txt` проекта.

```
# Документация
myx_cmake_doc_doxygen(LATEX YES HTML YES)
```

В результате будет добавлена цель `myx-cmake-doc-doxygen`, которую можно использовать после конфигурирования проекта:

```
make myx-cmake-doc-doxygen
```

Шаблоны для комментирования файлов, классов и функций можно автоматически расставить в файлах исходных кодов с помощью цели `myx-cmake-doc-doxygen-append-comments`, которая доступна при наличии установленной программы `uncrustify`:

```
make myx-cmake-doc-doxygen-append-comments
```

Удаление установленных файлов

В библиотеку MyxCMake добавлена цель `uninstall`, позволяющая удалить файлы, которые могут быть установлены в результате выполнения цели `install`:

```
cmake -B_build -H. -DCMAKE_INSTALL_PREFIX=_output ①
cmake --build _build -t install ②
cmake --build _build -t uninstall ③
```

① Конфигурирование проекта (каталог сборки `_build`, каталог для установки `_output`)

② Компиляция проекта и установка в каталог `_output`

③ Удаление файлов, установленных в каталог `_output`, а также пустых каталогов

Архивирование проекта и создание пакетов

Стандартный модуль `CPack` предназначен для архивирования исходных текстов проекта и создания пакетов для установки в целевую систему. Переменные, необходимые для создания пакетов и архивов, устанавливаются в начале файла `CMakeLists.txt` см. выше.

Библиотека MyxCMake предоставляет возможность стандартного разбиения на пакеты в соответствии с критериями, приведёнными в таблице.

Таблица 4. Критерии разбиения на пакеты

Компонент	Имя пакета	Назначение
`main`	`proj`	Основные файла проекта (исполняемые файлы, файлы данных, настроек, ресурсов)
`dev`	`libproj-dev`	Заголовочные файлы, дополнительные файлы необходимые для разработки
`static`	`libproj-static-dev`	Статические библиотеки
`doc`	`proj-doc`	Документация

Принадлежность устанавливаемого файла к компоненту определяется с помощью

параметра `COMPONENT` функции `install()`. В компоненте `main` необходимо перечислить файлы для установки на целевую систему (исполняемые файлы, файлы настроек, файлы данных, разделяемые библиотеки), в компонентах `dev` и `static` --- для установки на систему для разработки (заголовочные файлы, статические библиотеки). Пример включения устанавливаемых файлов в компоненты можно посмотреть здесь. Имена компонентов обязательно должны быть в нижнем регистре.

По умолчанию цель для упаковки исходных текстов называется `package_source`, цель для создания общего архива скомпилированного проекта --- `package`, цель для создания пакетов в формате Debian --- `deb`.

Пример работы с архивированием на основе репозитория библиотеки MyXLib:

```
git clone --recursive https://git.246060.ru/f1x1t/myxlib -b v2
```

В корневом каталоге проекта нужно выполнить команды:

```
cmake -B_build/release -H. -DMYXLIB_BUILD_LIBRARIES=ON -DMYXLIB_BUILD_EXAMPLES=ON
-DCMAKE_INSTALL_PREFIX=_output ①
cmake --build _build/release -t install -- -j4 ②
cmake --build _build/release -t doc-doxygen ③
cmake --build _build/release -t package ④
cmake --build _build/release -t package_source ⑤
cmake --build _build/release -t deb ⑥
```

① Создание конфигурации для сборки проекта

② Компиляция проекта

③ Генерирование документации

④ Создание архива с результатами компиляции `myx_amd64_0.4.1.tar.xz`

⑤ Создание архива с исходными текстами `myx-0.4.1.tar.xz`

⑥ Создание пакетов в формате Debian