CMake: управление проектом

Содержание

Полезные ссылки . . . 1
Структура каталогов проекта . . . 1
Базовый проект . . . 3
- Инициализация подмодулей . . . 4
- Базовые инструкции в CMake . . . 6

Поиск системных библиотек . . . 7
- Поиск с помощью программы `pkg-config` . . . 8
- Поиск с помощью функции `find_package` . . . 8

Автоматически генерируемый заголовочный файл . . . 8
Удаление установленных файлов . . . 9
Архивирование проекта и создание пакетов . . . 9
Примеры библиотек и приложений . . . 10
- Базовая библиотека . . . 10
- Базовое приложение . . . 13
- Подключение внешнего проекта . . . 15
- Qt5 . . . 17

Дополнительные возможности . . . 25
- Форматирование исходных текстов . . . 25
- Статический анализ исходных кодов . . . 26
- Автоматическое исправление кода . . . 27
- Динамический анализ программы . . . 29
- Анализ покрытия кода . . . 30
- Профилирование кода . . . 31
- Ускорение компиляции . . . 31
- Документирование кода . . . 32

Полезные ссылки

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

Структура каталогов проекта

Файлы проекта и результаты компиляции размещаются в каталогах:

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

Назначение каталогов приведено в таблице.

Таблица 1. 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 `cmex/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 `cmex/files/log`
`_build/debug/share`	Символическая ссылка на каталог `cmex/files/share`
`_build/debug/var`	Символическая ссылка на каталог `cmex/files/var`

Каталог	Назначение
`_build/release`	Результаты компиляции в режиме выпуска (иерархия аналогична `debug`)
`.git`	Системные файлы репозитория Git
`cmake`	Файлы с дополнительными функциями для CMake
`cmake/cmlib`	Библиотека функций для CMake
`cmake/etc`	Файлы настроек, используемые в CMake
`cmake/etc/uncrustify`	Файл настройки для программы автоматического форматирования исходных текстов
`cmake/find`	Модули CMake для поиска внешних программ и библиотек
`cmake/generators`	Генераторы проектов
`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. При работе с несколькими типами сборки, например, отладка и выпуск, появляется необходимость корректного полного удаления результатов предыдущего тип сборки.

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/cmlib-example-base
```

Инициализация подмодулей

Для начала нужно создать каталог для проекта, перейти в него и инициализировать репозиторий:

```
mkdir cmlib-example-base
cd cmlib-example-base
git init
```

Для подключения основных подмодулей, содержащих дополнительные функции для работы с проектом, и фиксации произведённого изменения нужно выполнить:

```
git submodule add https://git.246060.ru/f1x1t/cmlib.git cmake/cmlib
git submodule add https://git.246060.ru/f1x1t/cmake-find.git cmake/find
git submodule add https://git.246060.ru/f1x1t/cmake-generators.git cmake/generators
git commit -a -m "Начало проекта"
```

Отправить изменения в проекте на сервер и сделать ветку `master` основной (можно пропустить):

```
git remote add origin АДРЕС_РЕПОЗИТОРИЯ_НА_СЕРВЕРЕ
git push -u origin master
```

После отправки файлов на сервер отредактировать файл `.gitmodules`, так чтобы он содержал только относительные пути к подмодулям, чтобы система автоматической сборки проекта могла их загружать без дополнительной авторизации. Например, заменить:

```
[submodule "cmake/cmlib"]
    path = cmake/cmlib
    url = git@git.246060.ru:f1x1t/cmlib.git
[submodule "cmake/find"]
    path = cmake/find
    url = git@git.246060.ru:f1x1t/cmake-find.git
[submodule "cmake/generators"]
    path = cmake/generators
    url = git@git.246060.ru:f1x1t/cmake-generators.git
```

на

```
[submodule "cmake/cmlib"]
    path = cmake/cmlib
    url = ../../f1x1t/cmlib.git
[submodule "cmake/find"]
    path = cmake/find
    url = ../../f1x1t/cmake-find.git
[submodule "cmake/generators"]
    path = cmake/generators
    url = ../../f1x1t/cmake-generators.git
```

Обновить пути и сохранить изменения:

```
git submodule sync --recursive
git add .gitmodules
git commit -m "Настройка путей к подмодулям"
```

Загрузить шаблоны для автоматической сборки проекта в разных вариантах программных окружений и зафиксировать изменения:

```
wget https://git.246060.ru/f1x1t/gitlab-ci/raw/branch/master/.gitlab-ci.yml
git add .gitlab-ci.yml
git commit -m "Настройка автосборки"
```

Загрузить файл настройки для анализатора Clang-Tidy:

```
wget https://git.246060.ru/f1x1t/clang-tidy-config/raw/branch/master/.clang-tidy
git add .clang-tidy
git commit -m "Настройка Clang-Tidy"
```

Загрузить файл настройки для программы cmake-format, используемой для форматирования файлов CMake:

```
wget https://git.246060.ru/f1x1t/cmake-format/raw/branch/master/.cmake-format.py
git add .cmake-format.py
git commit -m "Настройка cmake-format"
```

Создать стандартные файлы и каталоги:

```
mkdir -p doc/breathe
touch doc/breathe/index.md.in
mkdir -p files/{etc,log,share,var}
touch files/{etc,log,share,var}/.gitkeep
git add doc files
git commit -m "Стандартные файлы и 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!).

Создать файл `.gitignore` для исключения каталогов и файлов из-под контроля Git:

```
wget https://git.246060.ru/f1x1t/cmlib-gitignore/raw/branch/master/.gitignore
git add .gitignore
git commit -m "Шаблон для игнорирования каталогов и файлов"
```

Базовые инструкции в CMake

В корневом каталоге проекта нужно создать 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(cmlib-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 CMLib, нужно добавить в файл `CMakeLists.txt` строки:

```
###
# Обязательные переменные для CMLib
###
# Название организации
set(CMLIB_ORGANIZATION_NAME "Org." CACHE STRING "")

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

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

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

# В каталоге cmake/cmlib находятся файлы с библиотечными функциями
if(IS_DIRECTORY ${CMAKE_SOURCE_DIR}/cmake/cmlib)
  list(INSERT CMAKE_MODULE_PATH 0 ${CMAKE_SOURCE_DIR}/cmake/cmlib)
else()
  message(FATAL_ERROR "CMake library directory does not exist")
endif()
list(APPEND CMAKE_MODULE_PATH ${CMAKE_SOURCE_DIR}/cmake/find)

include(CMLibCommon)
```

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

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

Чтобы проверить корректность подключения CMLib, можно выполнить команду:

```
(mkdir -p _build && cd _build && cmake .. && make && echo OK)
```

Если последней строкой вывода будет `OK`, то настройка завершена верно.

Поиск системных библиотек

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, 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 REQUIRED gsl)
pkg_check_modules(FFTW3 REQUIRED fftw3 >= 3.3.2)
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 curl libcurl-dev libboost-all-dev
```

Автоматически генерируемый заголовочный файл

На этапе конфигурирования проекта можно сгенерировать файл, в который будут записаны

собранные значения параметров. В библиотеке CMLib присутствует функция `cmlib_generate_private_config_hpp()`, создающая файл `${CMAKE_BINARY_DIR}/include/cmlib_private_config.hpp`, в который записывается информация о имени и версии проекта, дате и типе сборки.

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

Удаление установленных файлов

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

```
cd _build/debug
make install
make uninstall
```

Архивирование проекта и создание пакетов

Стандартный модуль `CPack` предназначен для архивирования исходных текстов проекта и создания пакетов для установки в целевую систему. Необходимые переменные устанавливаются в файле `cmake/etc/Variables.cmake` см. выше.

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

Таблица 2. Критерии разбиения на пакеты

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

Принадлежность устанавливаемого файла к компоненту определяется с помощью параметра `COMPONENT` функции `install`. В компонент `main` необходимо помещать файлы для установки на целевую систему (исполняемые файлы, файлы настроек, файлы данных, разделяемые библиотеки), в компоненты `base-dev` и `base-libs` --- для установки на систему для разработки (заголовочные файлы, статические библиотеки). Пример включения устанавливаемых файлов в компоненты можно посмотреть здесь.

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`.

Для примера работы с архивированием можно сделать копию репозитория библиотеки MyXLib:

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

Дальнейшие шаги:

```
cd myxlib ①
mkdir -p _build/debug ②
cd _build/debug ③
cmake ../.. -DMYXLIB_BUILD_LIBRARIES=ON -DMYXLIB_BUILD_EXAMPLES=ON ④
make -j4 ⑤
make doc-doxygen ⑥
make package ⑦
make package_source ⑧
make deb ⑨
```

① Перейти в каталог проекта

② Создать каталог для сборки проекта

③ Перейти в каталог для сборки проекта

④ Создать конфигурацию для сборки проекта

⑤ Скомпилировать проект

⑥ Сгенерировать документацию

⑦ Создать архив с результатами компиляции `myx_amd64_0.4.0.tar.xz`

⑧ Создать архив с исходными текстами `myx-0.4.0.tar.xz`

⑨ Создать пакеты в формате Debian

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/cmlib-example-library
```

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

```
# Поиск библиотеки 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
cmlib_generate_private_config_hpp()

# Каталог с исходными текстами библиотеки
add_subdirectory(src/cmlib-example)

# Документация
add_breathe_target(doc-breathe)
add_doxygen_target(doc-doxygen LATEX YES HTML YES)

# Создание вспомогательных символических ссылок
add_dependencies(cmlib-example create_auxilary_symlinks)
```

В подкаталоге `src/cmlib-example` нужно создать файл `CMakeLists.txt`:

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

# 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
add_common_library(${TRGT} SOURCES ${TRGT_cpp} ${TRGT_hpp})
common_target_properties(${TRGT})

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

# Цель, используемая только для установки
# заголовочных файлов без компиляции проекта
add_custom_target(${TRGT}-install-headers COMMAND "${CMAKE_COMMAND}"
-DCMAKE_INSTALL_COMPONENT=DEV -P

"${CMAKE_BINARY_DIR}/cmake_install.cmake")

# Установка статической библиотеки
install(TARGETS ${TRGT}_static COMPONENT DEV ARCHIVE DESTINATION
${CMAKE_INSTALL_LIBDIR})

# Установка динамической библиотеки
if(BUILD_SHARED_LIBS)
  install(TARGETS ${TRGT}_shared COMPONENT DEV LIBRARY DESTINATION
${CMAKE_INSTALL_LIBDIR})
endif()

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

# Установка файла для pkg-config
install(FILES ${CMAKE_BINARY_DIR}/${TRGT}.pc COMPONENT DEV DESTINATION
${CMAKE_INSTALL_LIBDIR}/pkgconfig)
```

файл `init.hpp`:

```
#ifndef CMLIB_EXAMPLE_HPP_
#define CMLIB_EXAMPLE_HPP_

#include <stdint.h>

int32_t cmlib_example_init(int32_t i);

#endif // CMLIB_EXAMPLE_HPP_
```

и файл `init.cpp`:

```
#include "init.hpp"

#include <boost/range/counting_range.hpp>

int32_t cmlib_example_init(int32_t i = 0)
{
    int32_t s = 0;
    for ( auto r : boost::counting_range( 1, i ) )
    {
        s += r;
    }
    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/cmlib-example-app
```

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

```
# 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
cmlib_generate_private_config_hpp()

# Приложение
add_subdirectory(src/cmlib-example)

# Документация
add_breathe_target(doc-breathe)
add_doxygen_target(doc-doxygen LATEX YES HTML YES)

# Создание вспомогательных символических ссылок
add_dependencies(cmlib-example create_auxilary_symlinks)
```

В подкаталоге `src/cmlib-example` нужно создать файл `CMakeLists.txt`:

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

# 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} ${TRGT_cpp})
common_target_properties(${TRGT})

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

# Имя целевого каталога и выходного файла для цели
set_target_properties(${TRGT} PROPERTIES OUTPUT_NAME ${TRGT})

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

и файл `main.cpp`:

```
#include "compiler_features.hpp"
#include "cmlib_private_config.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 compiler_features.hpp
    std::cout << CMLIB_EXAMPLE_APP_COMPILER_VERSION_MAJOR << std::endl;
    // Значение из cmlib_private_config.hpp
    std::cout << CMLIB_BUILD_TYPE << std::endl;
    // Значение из cmlib_private_config.hpp
    std::cout << CMLIB_BUILD_DATE << 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/cmlib-example-app-ext
```

Для подключения проекта базовой библиотеки нужно выполнить:

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

В файл `CMakeLists.txt`, находящийся в корневом каталоге проекта, перед функциями `add_subdirectories` нужно добавить:

```
# Подключение внешних проектов
include(ExternalProject)

ExternalProject_Add(ext-lib
  EXCLUDE_FROM_ALL  TRUE
  SOURCE_DIR        ${CMAKE_SOURCE_DIR}/thirdparty/cmlib-example-library
  INSTALL_DIR       ${CMAKE_BINARY_DIR}
  DOWNLOAD_COMMAND  ""
  BUILD_BYPRODUCTS  <INSTALL_DIR>/lib/libcmlib-example.a
  CMAKE_ARGS        -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
-DCMAKE_BUILD_TYPE=Release
  )
```

В результате будет создана цель `ext-lib`, 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/cmlib-example/CMakeLists.txt` после создания цели `${TRGT}` нужно подключить внешний проект `ext-lib`:

```
# Зависимость от библиотеки из внешнего проекта проекта
add_dependencies(${TRGT} ext-lib)

# Добавление каталога, в который устанавливаются заголовочные файлы
# от внешнего проекта, к списку путей для поиска
target_include_directories(${TRGT} PUBLIC
$<BUILD_INTERFACE:${CMAKE_BINARY_DIR}/include>)

# Компоновка с библиотекой из внешнего проекта
target_link_libraries(${TRGT} ${CMAKE_BINARY_DIR}/lib/libcmlib-example.a)
```

Для проверки работоспособности в файле `src/cmlib-example/main.cpp` нужно вызвать функцию `cmlib_example_init` из библиотеки, предоставляемой внешним проектом. Например, можно заменить его содержимое на:

```
#include <cmlib-example/init.hpp>

#include <iostream>

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

    return ( s );
}
```

Qt5

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

Консольное приложение и локализация

Пример консольного приложения на Qt5 с поддержкой локализации основан на проекте базового приложения и библиотеке MyXLib. 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/cmlib-example-app-qt5-con
```

Для подключения проекта библиотеки MyXLib нужно выполнить:

```
git submodule add https://git.246060.ru/f1x1t/myxlib thirdparty/myxlib
git submodule update --init --recursive
```

В файлах `CMakeLists.txt` и `src/cmlib-example/CMakeLists.txt` нужно заменить все строки `cmlib-example` на `cmlib-example-app-qt5-con`.

В файл CMakeLists.txt, находящийся в корневом каталоге проекта, перед функциями add_subdirectories нужно добавить:

```
# Подключение внешних проектов
include(ExternalProject)

ExternalProject_Add(
  myxlib
  SOURCE_DIR ${CMAKE_SOURCE_DIR}/thirdparty/myxlib
  INSTALL_DIR ${CMAKE_BINARY_DIR}
  DOWNLOAD_COMMAND ""
  CONFIGURE_COMMAND
    ${CMAKE_COMMAND} -G "${CMAKE_GENERATOR}" -DCMAKE_BUILD_TYPE=Debug
    -DCMAKE_C_COMPILER=${CMAKE_C_COMPILER}
    -DCMAKE_CXX_COMPILER=${CMAKE_CXX_COMPILER}
    -DCMAKE_INSTALL_PREFIX=${CMAKE_BINARY_DIR} <SOURCE_DIR>
  BUILD_COMMAND true)
```

В результате будет создана цель myxlib, 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/cmlib-example/CMakeLists.txt после создания цели ${TRGT} нужно подключить внешний проект myxlib:

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

# Добавление каталога, в который устанавливаются заголовочные файлы
# от внешнего проекта, к списку путей для поиска
target_include_directories(${TRGT} PUBLIC
$<BUILD_INTERFACE:${CMAKE_BINARY_DIR}/include>)
```

Для поиска необходимых компонентов Qt5 нужно в файле CMakeLists.txt, находящемся в корневом каталоге проекта, перед вызовом функции cmlib_generate_private_config_hpp() добавить строку:

```
# Используемые компоненты Qt5
find_package(Qt5 COMPONENTS Core REQUIRED)
```

Если в проекте используются приватные заголовочные файлы из пакета qtbase5-private-dev, то нужно добавить следующую инструкцию:

```
find_package(Qt5Core COMPONENTS Private REQUIRED)
find_package(Qt5 COMPONENTS Core REQUIRED)
```

В файл src/cmlib-example/CMakeLists.txt перед вызовом функции add_executable добавить строки:

```
# Правила для создания файла ресурсов с вложенными файлами переводов
qt5_translation(
    TRGT_qrc
    OUTPUT_DIR ${CMAKE_SOURCE_DIR}/l10n BASE_NAME ${TRGT}
    SOURCES ${TRGT_cpp} LANGUAGES ru_RU)

# Путь поиска библиотек созданных при компиляции проекта,
# включая библиотеки из подключённых внешних проектов, например MyXLib
# Функция link_directories обязательно должна находиться перед
# функцией add_executable, иначе компоновка не может быть выполнена
link_directories(${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
```

В вызове add_executable подключить использование файла ресурсов с переводами:

```
add_executable(${TRGT} ${TRGT_cpp} ${TRGT_qrc})
```

После чего добавить подключение Qt5 и MyXLib:

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

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

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

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

# Добавление каталога, в который устанавливаются заголовочные файлы
# от внешнего проекта, к списку путей для поиска
target_include_directories(${TRGT} PUBLIC
$<BUILD_INTERFACE:${CMAKE_BINARY_DIR}/include>)
```

Для проверки работоспособности подключения Qt5 файл `src/cmlib-example/main.cpp` нужно заменить на:

```
#include "cmlib_private_config.hpp"

#include <myx/qt/translators.hpp>

#include <QCoreApplication>
#include <QDebug>

namespace MQ = myx::qt;

int main( int argc, char** argv )
{
    QCoreApplication app( argc, argv );
    MQ::QTranslatorsList tl;

    qDebug() << QObject::tr( "No" );
    MQ::append_translators( tl, QStringLiteral( CMLIB_PROJECT_NAME ) );
    qDebug() << QObject::tr( "Yes" );

    return( 0 );
}
```

Для сбора списка строк из файлов исходных кодов и описаний интерфейса, подлежащих переводу, создаётся цель `l10n`. В результате выполнения в каталоге сборки команды `make l10n` в каталоге `l10n`, находящемся в корне проекта, появится файл `cmlib-example-app-qt5-con_ru_RU.ts`, в котором нужно отредактировать переводы с помощью программы `linguist`. После сохранения файла переводов проект нужно пересобрать, файл переводов в скомпилированном виде будет встроен в исполняемый файл `cmlib-example-app-qt5-con`, а доступ к нему будет осуществляться с помощью кода:

```
    MQ::QTranslatorsList tl;
    MQ::append_translators( tl, QStringLiteral( CMLIB_PROJECT_NAME ) );
```

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/cmlib-example-app-qt5-gui
```

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/cmlib-example-app-qt5-
gui/raw/branch/master/files/share/icon.png
```

Для графического приложения нужно создать файл описания интерфейса `src/cmlib-example/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/cmlib-example/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/cmlib-example/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 заменить файл `src/cmlib-examples/main.cpp` на:

```
#include "cmlib_private_config.hpp"
#include "test_window.hpp"

#include <myx/qt/translators.hpp>

#include <QApplication>
#include <QIcon>
#include <QDebug>

namespace MQ = myx::qt;


int main( int argc, char** argv )
{
    QApplication app( argc, argv );
    qDebug() << QObject::tr( "No" );

    // Подключение переводов
    MQ::QTranslatorsList tl;
    MQ::append_translators( tl, QStringLiteral( CMLIB_PROJECT_NAME ) );
    qDebug() << QObject::tr( "Yes" );

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

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

В файлах `CMakeLists.txt` и `src/cmlib-example/CMakeLists.txt` нужно заменить все строки `cmlib-example-app-qt5-con` на `cmlib-example-app-qt5-gui`.

Для поиска необходимых компонентов Qt5 нужно в файле `CMakeLists.txt`, находящемся в корневом каталоге проекта, добавить поиск компонентов `Gui` и `Widgets`:

```
# Используемые компоненты Qt5
find_package(Qt5 COMPONENTS Core Gui Widgets REQUIRED)
```

В файле `src/cmake-example/CMakeLists.txt` добавить новые файлы к списку файлов, используемых для компиляции:

```
###
# 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
###
```

Для обеспечения работы препроцессоров Qt необходимо создать правила преобразования файлов:

```
# 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})

# Правила для создания файла ресурсов с вложенными файлами переводов
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 добавляет свои файлы к списку ресурсов)
qt5_add_resources(TRGT_qrc_cpp ${TRGT_qrc})
```

Цель для создания исполняемого файла нужно изменить таким образом, чтобы она

зависела от файлов с исходными кодами и файлов, генерируемых препроцессорами:

```
add_executable(${TRGT} ${TRGT_ui_h} ${TRGT_moc_cpp} ${TRGT_qrc_cpp} ${TRGT_cpp}
${TRGT_hpp})
```

Подключение заголовочных файлов и библиотек Qt должно выглядеть так:

```
# 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 результате выполнения в каталоге сборки команды `make l10n` в каталоге `l10n`, находящемся в корне проекта, появится файл `cmlib-example-app-qt5-gui_ru_RU.ts`, в котором нужно отредактировать переводы с помощью программы `linguist`. После сохранения файла переводов проект нужно пересобрать, файл переводов в скомпилированном виде будет встроен в исполняемый файл.

Дополнительные возможности

Библиотека CMLib содержит шаблонные функции для использования в программных проектах. Пример проекта с примерами использования функций основан на проекте графического 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/cmlib-example-app-features
```

Форматирование исходных текстов

Функция `add_format_sources` генерирует цель для форматирования файлов проекта в едином стандарте, для её использования требуются установленные программы `dos2unix` и `uncrustify`. Утилита `dos2unix` приводит переводы строк в файлах к стандарту, принятому в Unix. Утилита `uncrustify` форматирует файлы с исходными кодами на языке C++ в соответствии с правилами, перечисленными в файле `cmake/etc/uncrustify/default.cfg`. Можно использовать собственный файл `default.cfg` или подключить подмодуль из репозитория:

```
git submodule add https://git.246060.ru/f1x1t/uncrustify-config.git
cmake/etc/uncrustify
git commit -m "Настройка uncrustify"
```

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!

Пример использования:

```
# Создание цели format-sources для автоматического форматирования кода
add_format_sources(${TRGT} ${TRGT_cpp} ${TRGT_headers})
```

Для автоматической проверки исходных текстов на соответствие стандарту форматирования можно к локальному репозиторию подключить скрипт, выполняемый перед фиксацией (`pre-commit`):

```
wget -O - https://git.246060.ru/f1x1t/githooks/archive/master.tar.gz | tar zx --strip
-components=1 -C .git/hooks
```

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

clazy

Функция `add_clazy_check` создаёт цели, которые используются для проверки исходных текстов анализатором `clang`. Пример использования:

```
# Создание цели clazy-check для проверки утилитой clazy
add_clazy_check(${TRGT} ${TRGT_cpp} ${TRGT_hpp} ${TRGT_moc_hpp})
```

Clang Tidy

Функция `add_clang_tidy_check` создаёт цели, которые используются для проверки исходных текстов анализатором `clang-tidy`. Правила проверок задаются в файле `.clang-tidy`, находящемся в корневом каталоге проекта. Пример использования:

```
# Создание цели clang-tidy-check для проверки утилитой clang-tidy
add_clang_tidy_check(${TRGT} ${TRGT_cpp} ${TRGT_hpp} ${TRGT_moc_hpp})
```

Clang Static Analyzer

Функция `add_clang_analyze_check` создаёт цели, которые используются для проверки исходных текстов анализатором `clang-analyze`. Пример использования:

```
# Создание цели clang-analyze-check для проверки утилитой clang-analyze
add_clang_analyze_check(${TRGT} ${TRGT_cpp} ${TRGT_hpp} ${TRGT_moc_hpp})
```

PVS-Studio

Функция `add_pvs_check` создаёт цели, которые используются для проверки исходных текстов анализатором `pvs-studio-analyzer`. Пример использования:

```
# Создание цели pvs-check для проверки утилитой pvs-studio-analyzer
add_pvs_check(${TRGT})
```

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 использования данной возможности необходимо установить пакеты:

```
sudo apt-get install clazy clang-tools
```

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

Рисунок 1. Выбор цели

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

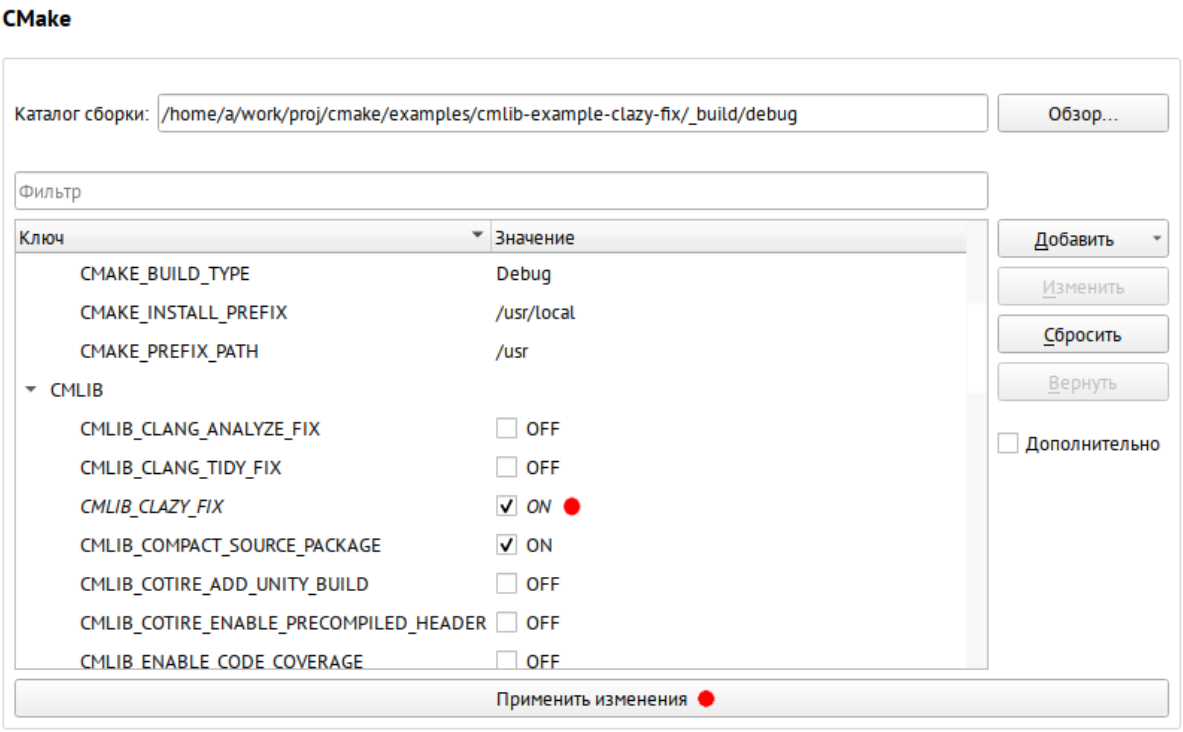

Рисунок 2. Разрешение автозамены

Пример проекта, в котором показаны возможности clazy, можно посмотреть здесь. Содержание изменений, произведённых автоматически, можно увидеть здесь.

Можно сделать копию репозитория и выполнить правки в автоматическом режиме самостоятельно:

```
git clone --recursive https://git.246060.ru/f1x1t/cmlib-example-clazy-fix
```

Clang-Tidy

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

```
sudo apt-get install clang-tools clang-tidy
```

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

Рисунок 3. Выбор цели

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

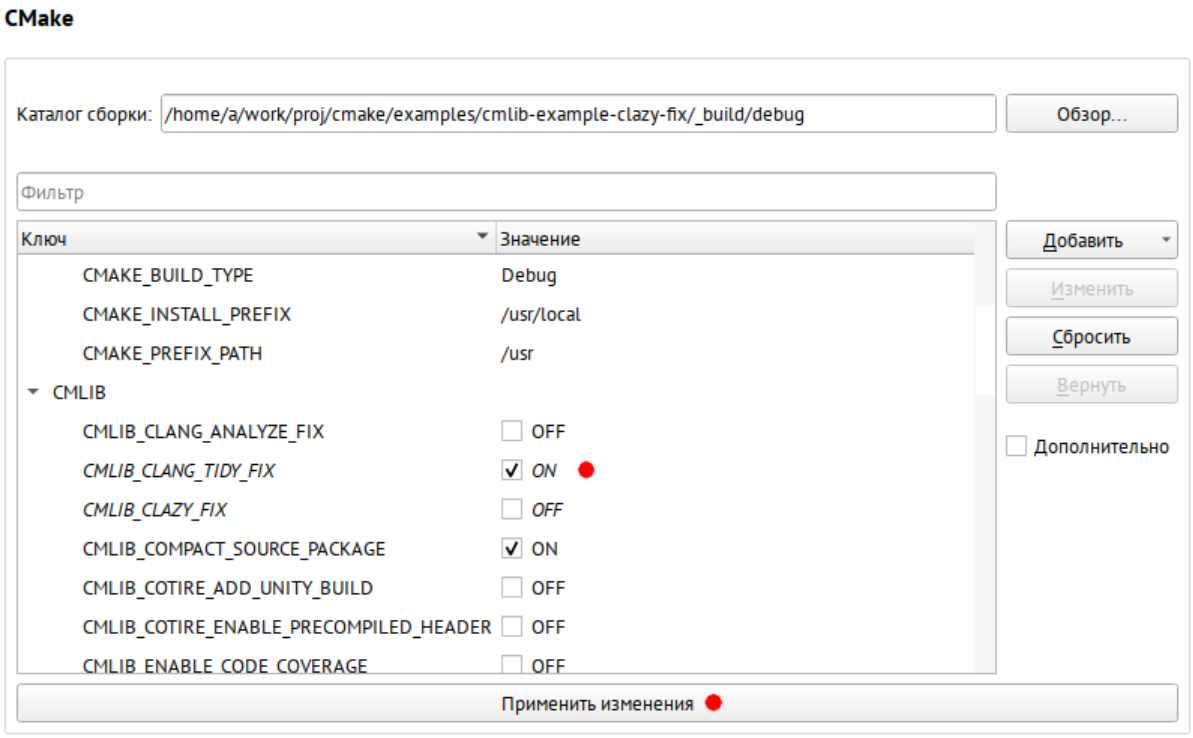

Рисунок 4. Разрешение автозамены

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 (обязательно после подключения всех библиотек):

```
# Подключение настроек для динамического анализа программы
add_sanitizers(${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

Анализ покрытия кода

Для сбора информации о точном количестве исполнений для каждого оператора в программе используется программа Gcov, входящая в состав компилятора GCC.

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

```
# Подключение возможности использования утилиты Gcov
# для исследования покрытия кода
add_code_coverage(${TRGT})
```

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

Пример анализа покрытия кода на примере проекта `cmlib-example-app-features`:

```
mkdir -p _build/debug
cd _build/debug
cmake ../.. -DENABLE_CODE_COVERAGE=ON -DCMAKE_BUILD_TYPE=Debug
make
bin/cmlib-example-app-features
make coverage-cmlib-example-app-features
make coverage-report-cmlib-example-app-features
```

После выполнения этих команд в каталоге `report-cmlib-example-app-features` будет сформирован отчёт в виде HTML-страниц.

Профилирование кода

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

```
mkdir -p _build/profile
cd _build/profile
cmake ../.. -DCMAKE_BUILD_TYPE=Profile
```

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

```
./cmlib-example-app-features
gprof -b cmlib-example-app-features gmon.out > analysis-tree.txt
gprof -b -p cmlib-example-app-features gmon.out > analysis-flat.txt
```

Ускорение компиляции

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

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

```
# Подключение возможности включения пакетного режима обработки
# исходных файлов в генераторах для ускорения сборки
cotire(${TRGT})
```

В результате будут созданы цели с суффиксом `_unity`, при сборки которых будут применяться приведённые выше методы ускорения.

Пример использования cotire для ускорения сборки на примере проекта `cmlib-example-app-features`:

```
mkdir -p _build/debug
cd _build/debug
cmake ../..
make all_unity
```

Документирование кода

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

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

Пример комментария:

```
/**
 * @brief Базовый класс
 */
class Base {
public:
    /**
     * @brief Конструктор
     */
    Base();
    /**
     * @brief Деструктор
     */
    ~Base();
    /**
     * @brief Вычисление квадратного корня
     * @param value Входное значение
     * @return Квадратный корень от value
     */
    double sqrt(double value);
};
```

Поддержка автоматической генерации документации реализована в функциях библиотеки CMLib `add_doxygen_target` и `add_breathe_target`, которые необходимо вызвать в основном файле `CMakeLists.txt` проекта.

```
# Документация
add_doxygen_target(doc-doxygen LATEX YES HTML YES)
add_breathe_target(doc-breathe)
```

В результате будут добавлены цели `doc-doxygen` и `doc-breathe`, которые можно использовать после конфигурирования проекта:

```
make doc-doxygen
make doc-breathe
```

Шаблоны для комментирования файлов, классов и функций можно автоматически расставить в файлах исходных кодов исполнением цели `doc-add-comments` при наличии установленной программы `uncrustify`:

```
make doc-add-comments
```