HADOUKEN: NETTSI C++ Project Development Environment

> `(codename: HADOUKEN)`

Table of contents

Preface

`Hadouken` contains common boilerplate code between all C++ projects being developed/will be developed in NETTSI.

This guide will navigate you through the `hadouken`'s all features and how to use it.

Author

Mustafa Kemal GILOR, Chief Software Architect (mgilor@nettsi.com)

Getting started

Hadouken is a CMake- Bash shell script project which is created to resolve three most common problems in C++ projects, eg. tool integration, dependency resolution and environment consistency.

The name `hadouken` comes from the Street Fighter hero, Ryu's iconic fight move. It's simple, powerful and super effective against the common enemies of C++ projects. Just hadouken away all of your problems! ;)

Hadouken is very easy to install. You can integrate it with your project in matter of seconds. Pre-requirements for the hadouken are minimal. In order to be able to use `hadouken` in a project, the project has to be version controlled in `Git`, and the project must be a `CMake` project. There are no requirements other than that.

Installing `hadouken` to a project

You're lucky, I've prepared a video guide illustrating this. Check out Video tutorial: How to install hadouken on new projects? to watch it.

Hadouken is designed as to be a git submodule. First, you need to add it as a submodule to your existing git project. In order to do that, issue following command at your git project root:

Public domain

Add submodule (via SSH)

```
    git submodule add -b master git@github.com:nettsi/hadouken.git .hadouken
```

Add submodule (via HTTP)

```
    git submodule add -b master https://github.com/nettsi/hadouken.git .hadouken
```

Nettsi employees

Add submodule (via SSH)

```
    git submodule add -b master git@gitlab.nettsi.com:foss-projects/hadouken.git
.hadouken
```

Add submodule (via HTTP)

```
    git submodule add -b master https://gitlab.nettsi.com/foss-
projects/hadouken.git .hadouken
```

This will add `hadouken` project to your project, to `.hadouken` folder. `master` branch will be tracked by default. After this, run following command from terminal in your project root. The name for the submodule has to be `.hadouken`. After adding the submodule, invoke the following command at project root to setup `hadouken` to your project:

```
    bash .hadouken/script/setup.sh
```

If your project root does not contain any CMake projects, `Hadouken` will offer you to quick-start a new `CMake` project. If you choose 'yes', Hadouken will ask you for a project name and automatically generate CMake files and an example C++ application.

```
Hadouken parent directory does not contain a CMakeLists.txt
Do you want to quick-start a new cmake project to parent folder [Yy/Nn]? y
Project name:
```

The script will offer you to install tools required and recommended by Hadouken. These tools are:

- Docker
- Visual Studio Code
- Visual Studio Code Extensions

Automatic installation of these tools are tested in Debian 10 and Ubuntu 18.10. Fedora and CentOS are also supported, but not tested (yet). You can always install these tools manually if you choose to do so.

After running the script, your project root should have the following symbolic links, mapped to the boilerplate content:

```
    # Visual Studio Code - Remote Containers .devcontainer
    .devcontainer -------> .hadouken/.devcontainer
    # Visual Studio Code settings
    .vscode -------------> .hadouken/dotfiles/.vscode
    # Git vcs ignored settings
    .gitignore ----------> .hadouken/dotfiles/.gitignore
    # Git settings
    .gitconfig ----------> .hadouken/dotfiles/.gitconfig
    # Clang-format style file
    .clang-format -------> .hadouken/dotfiles/.clang-format
    # Clang-tidy style file
    .clang-tidy ---------> .hadouken/dotfiles/.clang-tidy
    # Lcov report generation settings file
    .lcovrc -------------> .hadouken/dotfiles/.lcovrc
    # Cppcheck settings
    .cppcheck-suppress --> .hadouken/dotfiles/.cppcheck-suppress
    # Hadouken shell script
    hadouken ------------> .hadouken/script/hadouken.sh
```

The following hidden file(s) will be added to your project root:

```
.hadouken.docker-compose.extend.yml # docker-compose extension file
.hadouken-bootstrap.sh              # container post-installation script
```

If any of the file(s) specified above already exist on project root, they will not be overridden.

Hadouken script commands

There are some commands defined to configure and build the project on command line. Note that these commands currently not being run in development environment container. This functionality will be implemented soon.

```
-b|--build      build project
        | build previously configured project. Any extra arguments will be
forwarded to CMake.
        | example: `hadouken --build`           # build via CMake.
-x|--clean      clean project
        | remove PROJECT_ROOT/build directory.
        | example: `hadouken --clean`           # removes PROJECT_ROOT/build
directory.
        | example: `hadouken --clean --deep`    # runs git clean -fxd on project
root.
-c|--configure
        | configure project located in PROJECT_ROOT to PROJECT_ROOT/build
directory with cmake
        | any following arguments will be forwarded to CMake.
        | example: `hadouken --configure --target=Debug`
-i|--install    install project
        | install previosuly built project. Any extra arguments will be forwarded
to CMake.
        | example: `hadouken --install`         # install via CMake.
-p|--pack       pack project (using cpack)
        | package previously build project. Any extra arguments will be forwarded
to CMake.
        | example: `hadouken --pack`            # pack via CMake.
-t|--test       run unit tests for project (using ctest)
        | run unit tests of previously build project. Any extra arguments will be
forwarded to CTest.
        | example: `hadouken --test`            # pack via CMake.
-a|--all        clean->configure->build->pack project
-u|--upgrade    upgrade hadouken to latest release
        | this basically runs submodule update, reduced to a command for ease of
use.
        | example: `hadouken --upgrade`         # upgrade hadouken to latest
master release.
```

You can print this information any time by invoking `./hadouken --help` command.

Updating hadouken

Hadouken project will be separately maintained, and there will be updates. In order to update the hadouken to latest stable release, issue the following command from your project's root directory:

```
    ./hadouken --upgrade
```

This will upgrade the hadouken submodule from remote `master` branch. If you have done changes to your local copy of hadouken, this command will respect to that. In order to force the upgrade, add `--force` parameter to upgrade command.

```
./hadouken --upgrade --force
```

Hadouken user's manual

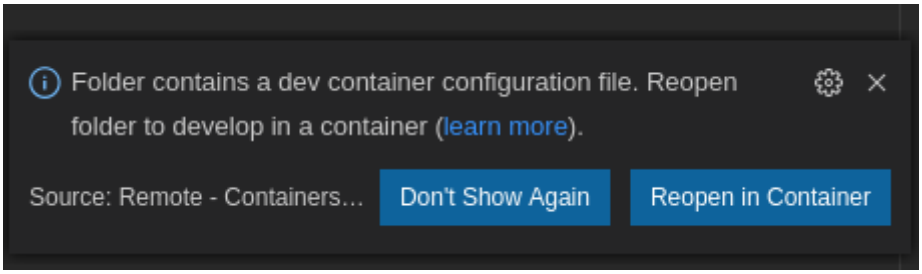
In this section we will cover what hadouken offers and how to use them.

Development environment container

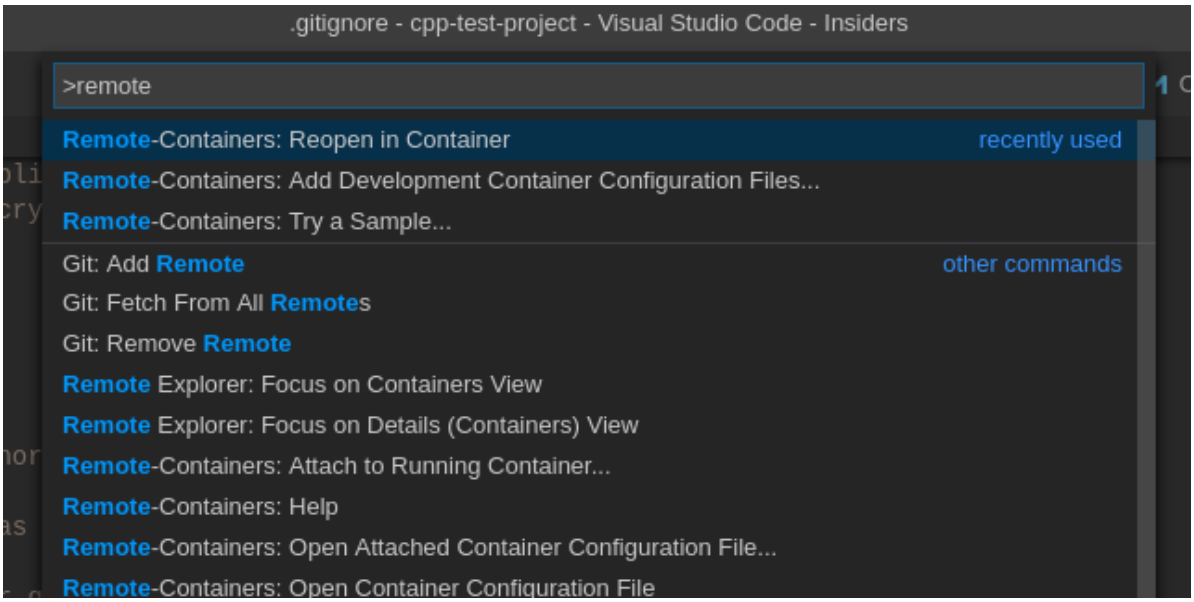
`Hadouken` offers a docker based development environment container, which features all neccessary utilities to satisfy the prerequisites of features offered by `Hadouken`.

The container is designed to be used with `Visual Studio Code - Remote - Containers` extension. If you are not an Visual Studio Code user, you can still use the container manually.

While running on VSCode with `Remote - Containers` extension, the extension will automatically recognize the `Dockerfile` and `devcontainer.json` files located in `.devcontainer` folder.



If that is not the case, you can still open your project in development container by pressing F1 (or Ctrl + Shift + P) and selecting `Remote-Containers: Reopen in Container` menu item.



This will open the project in development container. In first time, the extension will automatically build the docker image from the `Dockerfile` and then install a VSCode Server equipped with extensions specified in `devcontainer.json` file. This may take a while.

To go back to local environment, select `Remote-Containers: Reopen Locally` menu item.

Development environment is based on Debian Sid, and contains the following tools and utilities;

- Version Control System (git)
- GCC Toolchain (gcc-10, gdb)
- LLVM Toolchain (llvm-10, lldb-10, clang-10, clangd)
- Build System Utilities (make, ninja-build, autoconf, automake, libtool, m4, cmake)

- Static Analysis, Formatting, Linting, Diagnotics (clang-format-10 clang-tidy-10 iwyu cppcheck valgrind ccache)
- Unit Testing, Benchmarking (libgtest-dev, libgmock-dev)
- Code Coverage Analysis (lcov, gcovr)
- Documentation (doxygen, doxygen-doc, doxygen-latex, doxygen-doxyparse, graphviz)

Installing project-specific tools to development environment container

Method 1: Using .hadouken.bootstrap.sh

It is possible to run post-install commands in development environment container. To do that, edit the script file named `.hadouken.bootstrap.sh` at project root directory. This script file will be run when development environment container is built for first time.

> ![exclamation-mark] You can use `apt` or `pip` to install the packages you desire. As a rule of thumb, always check whether the third party dependency is available as conan package. If so, use conan to satisfy your project's dependency.

```
# Project requires boost, so install it.
apt install libboost-all-dev
```

If you already built the development environment container, the container is needed to be rebuilt in order to reflect the changes. To do that, Press F1 and type `Remote-Containers: Rebuild Container` and press Enter. This will re-create the container from scratch.

Method 2: Using .hadouken.docker-compose.extend.yml

You can extend base `docker-compose.yml` file to add more functionality, and even more images.

Hey, wait a second, what about git?

I asked the similar question myself and was going to implement a mechanism that exposes local git configuration and credentials to docker image, but realized that VSCode remote containers extension already does that. Your local git configuration is copied to the container as-is. If you are using http authentication, you can still proceed to authenticate using it. For authenticating with ssh keys, there is an extra step to take. Because of the security reasons, the extension does not copy the ssh keys to the container. Instead, it uses ssh-agent running on the host machine when ssh authentication is needed. So, if you want to be able to push to the remote repository from container shell, you have to spawn an instance of ssh-agent and add your relevant ssh private keys to it.

Before starting the container, run the following commands in a shell;

```
    eval "$(ssh-agent -s)" # Start ssh-agent, -s option prints the options used
by ssh agent to the stdout, eval sets them to the current shell.
    ssh-add ~/.ssh/id_rsa  # Add a ssh private key to the agent. The container
can only use the keys added to the agent.
```

After that, you can work with `git` as if you are on your local machine. While using ssh-agent, please bear in mind that all added ssh keys can be used by remote hosts which you `ssh`'ed into. I strongly encourage you to read ssh-agent's man page before using it: https://linux.die.net/man/1/ssh-agent

> ![exclamation-mark] If you want to spawn ssh-agent automatically on startup, you can add these lines to your `~/.bash_profile` script:

```
if [ -z "$SSH_AUTH_SOCK" ]
then
   # Check for a currently running instance of the agent
   RUNNING_AGENT="`ps -ax | grep 'ssh-agent -s' | grep -v grep | wc -l | tr -d
'[:space:]'`"
   if [ "$RUNNING_AGENT" = "0" ]
   then
        # Launch a new instance of the agent
        ssh-agent -s &> .ssh/ssh-agent
   fi
   eval `cat .ssh/ssh-agent`
   # You might add your ssh keys automatically here, but it is not recommended
   # for security reasons.
fi
```

Tool integration modules

Hadouken provides several modules to help with external tool integration and ease of use. All tools are included in `devenv` container, so skip any installation related stuff if you are using the `devenv` container.

`<project_name>` is capitalized and non-alphanumeric characters replaced (with an underscore) version of top level project name. The term `top level` here refers to the cmake project declaration prior to hadouken module inclusion.

ClangFormat

Controlled by `<project_name>_TOOLCONF_USE_CLANG_FORMAT` option.

Locate `clang-format*` in environment, if available. The status will be printed to the stdout.

This feature requires at least clang-format version 10 to be present in environment.

```
project(my-awesome-project)                              # Declare a new project
set(my-awesome-project_TOOLCONF_USE_CLANG_FORMAT TRUE)  # Locate clang-format and
use it if available.
include(.hadouken/hadouken.cmake)        # Use hadouken
# ClangFormat module will try to locate the clang-format executable.
# If clang-format executable is found, each target created via `make_target`
function call will have an additional
# target suffixed with `.format`, which will format the target's source code when
invoked.
# This `.format' target will use `.clang-format` file located in project root,
which will be a symbolic link to
# hadouken project's `.clang-format` file when not specified explicitly.
Hadouken's `.clang-format` file will
# format the source code according to NETTSI C++ Code Standards.
```

You can install `clang-format` by getting the desired revision of llvm project (clang-format is a part of it) https://github.com/llvm/llvm-project or you can get from you package manager.

Ubuntu/Debian

```
    sudo apt-get update
    sudo apt-get install clang-format
```

ClangTidy

Controlled by `<project_name>_TOOLCONF_USE_CLANG_TIDY` option.

Locate `clang-tidy*` in environment, if available. The status will be printed to the stdout.

This feature requires at least clang-format version 10 to be present in environment.

```
project(my-awesome-project)                              # Declare a new project
set(my-awesome-project_TOOLCONF_USE_CLANG_TIDY TRUE)     # Locate clang-format and
use it if available.
include(.hadouken/hadouken.cmake)        # Use hadouken
# ClangTidy module will try to locate the clang-format executable.
# If clang-tidy executable is found, each target created via `make_target`
function call will have an additional
# target suffixed with `.tidy`, which will tidy/lint the target's source code
when invoked.
# This `.tidy' target will use `.clang-tidy` file located in project root, which
will be a symbolic link to
# hadouken project's `.clang-tidy` file when not specified explicitly. Hadouken's
`.clang-tidy` file will
# tidy/lint the source code according to NETTSI C++ Code Standards.
```

You can install `clang-tidy` by getting the desired revision of llvm project (clang-tidy is a part of it) https://github.com/llvm/llvm-project or you can get from you package manager.

Ubuntu/Debian

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

> If you are using `vscode` for development, you can install the `vscode-clangd` extension in order to be able to apply quick fixes suggested by the `clang-tidy`. `vscode-clangd` requires clang language server `clangd` to be availabe in your environment. You will need to install `clangd` separately.
>
> For distros with apt based package managers:

```
    sudo apt-get install clangd
```

CppCheck

Controlled by `<project_name>_TOOLCONF_USE_CPPCHECK` option.

Locate `cppcheck` in environment, if available. The status will be printed to the stdout.

As CMake officially supports cppcheck, 'CMAKE_CXX_CPPCHECK' variable will be set accordingly and CMake will trigger cppcheck on every source file build.

```
project(my-awesome-project)                              # Declare a new project
set(my-awesome-project_TOOLCONF_USE_CPPCHECK TRUE)       # Locate cppcheck and use
it if available.
include(.hadouken/hadouken.cmake)        # Use hadouken
# No further action is required.
```

You can install `cppcheck` either by downloading it from the official site http://cppcheck.sourceforge.net/#download or you can get from you package manager.

Ubuntu/Debian

```
    sudo apt-get update
    sudo apt-get install cppcheck
```

GCov/LCov

Controlled by `<project_name>_TOOLCONF_USE_GCOV`, `<project_name>_TOOLCONF_USE_LCOV`, `<project_name>_TOOLCONF_USE_GCOVR` options.

Locate `gcov & lcov` in environment, if available. The status will be printed to stdout.

Code coverage is enabled per target basis. To enable code coverage target creation for a target, pass `WITH_COVERAGE` argument to `make_target` function. `WITH_COVERAGE` will cause `make_target` to create a new target named `<target_name>.lcov` (when lcov enabled and present) and `<target_name>.gcovr.xml`, `<target_name>.gcovr.html` (when gcovr enabled and present) which runs code coverage analysis and generates code coverage report when run.

```
project(my-awesome-project)                             # Declare a new project
set(my-awesome-project_TOOLCONF_USE_GCOV TRUE)          # Locate gcov and use it
if available.
set(my-awesome-project_TOOLCONF_USE_LCOV TRUE)          # Locate lcov and use it
if available.
set(my-awesome-project_TOOLCONF_USE_GCOVR TRUE)         # Locate gcovr and use
it if available.
include(.hadouken/hadouken.cmake)                       # Use hadouken

make_target(TYPE UNIT_TEST WITH_COVERAGE)
# Creates an unit test named my-awesome-project and my-awesome-project.cov
coverage target.
```

An example `lcov` unit test coverage scenario:

In this scenario, we have the following library and the corresponding unit test code:

lib.hpp

```
namespace hdktest{
    namespace lib{
        class library{
        public:
            library();
            ~library();

            bool should_return_true(int val);
        };
    }
}
```

lib.cpp

```
namespace hdktest::lib{
```

```
    library::library(){
        std::cout << "construct" << std::endl;
    }

    library::~library(){
        std::cout << "destruct" << std::endl;
    }

    bool library::should_return_true(int val){
        switch(val){
            case 1:
                return false;
            case 2:
                return true;
            case 3:
                return false;
        }
        return true;
    }
}
```

Library's CMakeLists.txt:

```
    project(hdktest.lib)

    make_target(TYPE STATIC)

    add_subdirectory(test)
```

ut_lib.cpp

```
TEST(test,test){
    hdktest::lib::library library;
    EXPECT_TRUE(true);
}

TEST(test, param1){
    hdktest::lib::library library;
    EXPECT_FALSE(library.should_return_true(1));
}

TEST(test, param2){
    hdktest::lib::library library;
    EXPECT_TRUE(library.should_return_true(2));
}

TEST(test, test_param3_Test){
    hdktest::lib::library library;
    EXPECT_FALSE(library.should_return_true(3));
}
```

Unit test's CMakeLists.txt:

```
    project(hdktest.lib.test)

    make_target(
        TYPE UNIT_TEST
        SOURCES ut_lib.cpp
        LINK hdktest.lib
        WITH_COVERAGE
        COVERAGE_TARGETS hdktest.lib
        NO_AUTO_COMPILATION_UNIT
    )
```

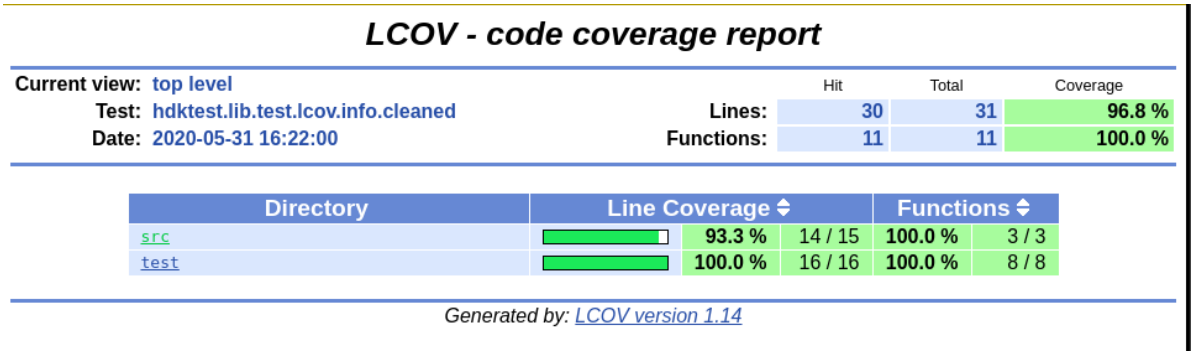
`gcov`, `lcov` and `gcovr` integrations are also enabled by specifying following options:

```
    SET(HDKTEST_TOOLCONF_USE_GCOV TRUE CACHE BOOL "Enable/disable gcov
integration" FORCE)
    SET(HDKTEST_TOOLCONF_USE_LCOV TRUE CACHE BOOL "Enable/disable lcov
integration" FORCE)
    SET(HDKTEST_TOOLCONF_USE_GCOVR TRUE CACHE BOOL "Enable/disable gcovr
integration" FORCE)
```

As we specified `WITH_COVERAGE` option and enabling coverage tools, following additional targets will be available for build (which will trigger code coverage report generation):

- hdktest.lib.test.gcovr.html
- hdktest.lib.test.gcovr.xml
- hdktest.lib.test.lcov

By building `hdktest.lib.test.lcov` target, a line coverage report will be generated in build folder.



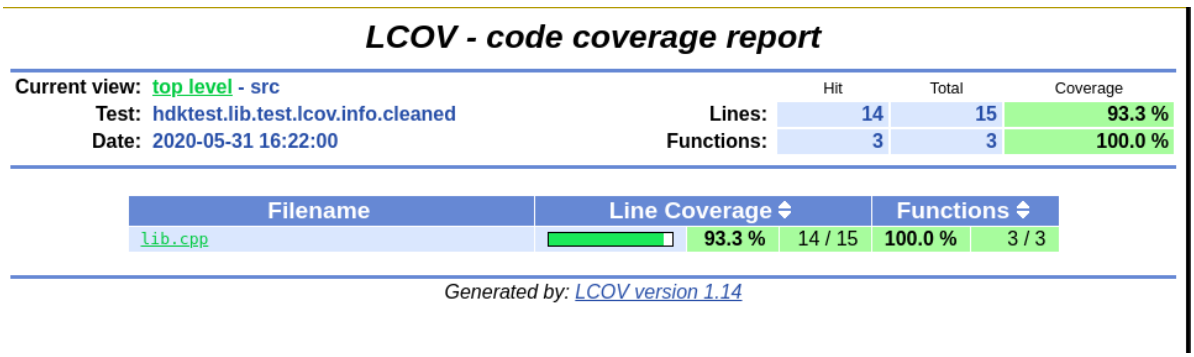
LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level				
Test:	hdktest.lib.test.lcov.info.cleaned	Lines:	30	31	96.8 %
Date:	2020-05-31 16:22:00	Functions:	11	11	100.0 %

Directory	Line Coverage		Functions	
src	93.3 %	14 / 15	100.0 %	3 / 3
test	100.0 %	16 / 16	100.0 %	8 / 8

Generated by: LCOV version 1.14

Directories shown in the report are `src` (source code of unit under test) and `test` (the unit test code). In the report, we can see the unit test is covered %93.3 of the unit under test. Let's dive into `src` to investigate situation more detailed.



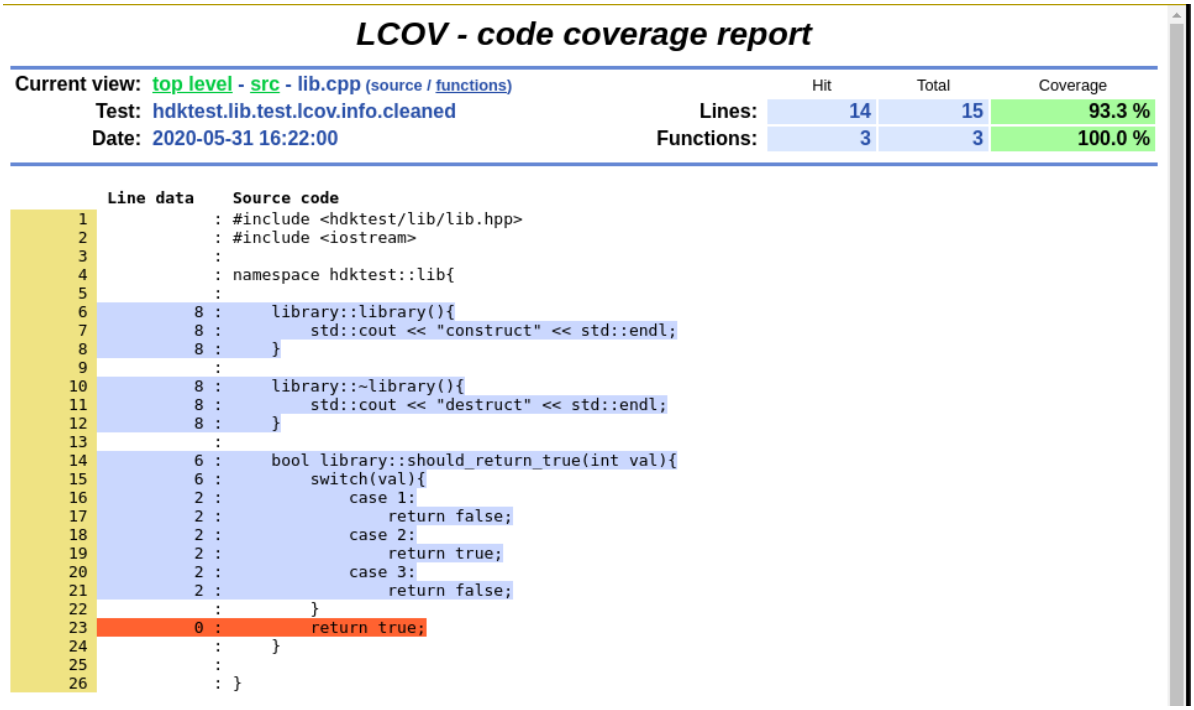
LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - src				
Test:	hdktest.lib.test.lcov.info.cleaned	Lines:	14	15	93.3 %
Date:	2020-05-31 16:22:00	Functions:	3	3	100.0 %

Filename	Line Coverage		Functions	
lib.cpp	93.3 %	14 / 15	100.0 %	3 / 3

Generated by: LCOV version 1.14

Here we can see unit under test consists of only `lib.cpp` file. By clicking `lib.cpp` we can check which lines of `lib.cpp` is not covered by the unit test.



LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level - src - lib.cpp (source / functions)			
Test:	hdktest.lib.test.lcov.info.cleaned	Lines:	14	15	93.3 %
Date:	2020-05-31 16:22:00	Functions:	3	3	100.0 %

	Line data	Source code
1		: #include <hdktest/lib/lib.hpp>
2		: #include <iostream>
3		:
4		: namespace hdktest::lib{
5		:
6	8	: library::library(){
7	8	: std::cout << "construct" << std::endl;
8	8	: }
9		:
10	8	: library::~library(){
11	8	: std::cout << "destruct" << std::endl;
12	8	: }
13		:
14	6	: bool library::should_return_true(int val){
15	6	: switch(val){
16	2	: case 1:
17	2	: return false;
18	2	: case 2:
19	2	: return true;
20	2	: case 3:
21	2	: return false;
22		: }
23	0	: return true;
24		: }
25		:
26		: }

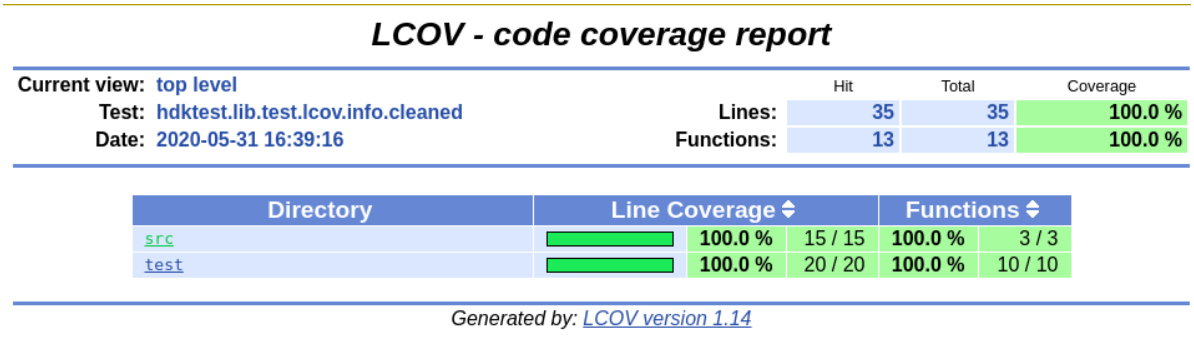
The blue lines are the lines which are executed by the unit test. The Line data column shows how many times the corresponding line is executed. The red lines are the lines which have not been executed by the unit test. By looking this line coverage result, we can clearly see that the unit test constructed & destructed an instance of library object 8 times, and `should_return_true` function has been called 6 times. In these calls, case 1 is taken 2 times, case 2 is taken 2 times and case 3 is taken 2 times respectively. We can conclude that our unit test did not supply an argument which is not handled by the switch case. To confirm that, let's take a look to the unit test code.

lib > test > ut_lib.cpp > TEST(test, param2)

```cpp
#include <gtest/gtest.h>
#include <hdktest/lib/lib.hpp>

TEST(test,test){
    hdktest::lib::library library;
    EXPECT_TRUE(true);
}

TEST(test, param1){
    hdktest::lib::library library;
    EXPECT_FALSE(library.should_return_true(1));
}

TEST(test, param2){
    hdktest::lib::library library;
    EXPECT_TRUE(library.should_return_true(2));
}

TEST(test, test_param3_Test){
    hdktest::lib::library library;
    EXPECT_FALSE(library.should_return_true(3));
}
```

Our unit test has 4 cases, but none of them calls `should_return_true` with a value which is outside of switch case's defined range of values. Let's fix that by adding a new unit test case, and check coverage report again.

```
TEST(test, test_param4_Test){
    hdktest::lib::library library;
    EXPECT_TRUE(library.should_return_true(4));
}
```



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level	**Lines:**	35	35	100.0 %
Test:	hdktest.lib.test.lcov.info.cleaned	**Functions:**	13	13	100.0 %
Date:	2020-05-31 16:39:16				

Directory	Line Coverage			Functions	
src		100.0 %	15 / 15	100.0 %	3 / 3
test		100.0 %	20 / 20	100.0 %	10 / 10

Generated by: LCOV version 1.14

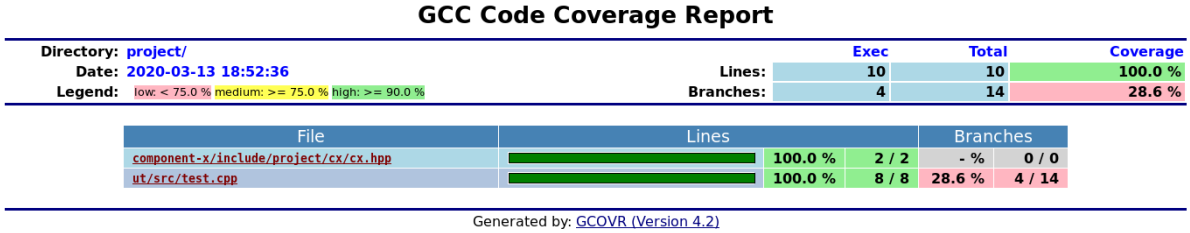
As you can see above, adding the missing case indeed fixed our coverage rate.

You can install `gcov, lcov, gcovr` either by downloading it from their respective official sites http://cppcheck.sourceforge.net/#download or you can get from you package manager.

Ubuntu/Debian

```
sudo apt-get update
sudo apt-get install gcov lcov gcovr
```

Example code coverage output of an unit test is shown below(gcovr/html):



GCC Code Coverage Report

Directory:	project/		**Exec**	**Total**	**Coverage**
Date:	2020-03-13 18:52:36	**Lines:**	10	10	100.0 %
Legend:	low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %	**Branches:**	4	14	28.6 %

File	Lines		Branches	
component-x/include/project/cx/cx.hpp	100.0 %	2 / 2	- %	0 / 0
ut/src/test.cpp	100.0 %	8 / 8	28.6 %	4 / 14

Generated by: GCOVR (Version 4.2)

GoogleTest/GoogleMock

Controlled by `<project_name>_TOOLCONF_USE_GOOGLE_TEST` variable.

Locate google test & google mock in environment, if available. The status will be printed to stdout.

Helper will create a target named `<project_name>.test` which includes google test & google mock header and libraries. You can link against this target in order to create an unit test.

If you are using `make_target`, the created target will be automatically linked against the test target.

```
project(my-awesome-project)
set(my-awesome-project_TOOLCONF_USE_GOOGLE_TEST TRUE)
include(.hadouken/hadouken.cmake)        # Use hadouken
# A target named `my-awesome-project.test` will be created if google test &
google mock
# are found in environment. Failure to do so results in CMake configure step
failure.
```

You can install `gtest/gmock` either by downloading it from the official site https://github.com/google/googletest/releases or you can get from you package manager.

Ubuntu/Debian

```
    sudo apt-get update
    sudo apt-get install libgtest-dev libgmock-dev
```

IncludeWhatYouUse (IWYU)

Controlled by `<project_name>_TOOLCONF_USE_IWYU` option.

Locate `include what you use` in environment, if available. The status will be printed to stdout.

As CMake officially supports `iwyu`, `CMAKE_CXX_INCLUDE_WHAT_YOU_USE` variable will be set accordingly and CMake will trigger iwyu on every source file build.

```
project(my-awesome-project)                          # Declare a new project
set(my-awesome-project_TOOLCONF_USE_IWYU TRUE)       # Locate iwyu and use it
if available.
include(.hadouken/hadouken.cmake)                    # Use hadouken
# No further action is required.
```

You can install `iwyu` either by downloading it from the official site https://include-what-you-use.org/downloads/ or you can get from you package manager.

Ubuntu/Debian

```
    sudo apt-get update
    sudo apt-get install iwyu
```

`iwyu` functions on specific clang version, so you also need to install that version. To determine which version is required, issue following command from the terminal;

```
    iwyu --version
    // Example output:
    // include-what-you-use 0.12 based on clang version 9.0.1-2
```

In the scenario above, we would need to install `clang-9`.

LinkWhatYouUse (LWYU)

Controlled by `<project_name>_TOOLCONF_USE_LWYU` option.

Link what you use requires no external tools. Standard toolchain linker will be used to perform this functionality.

As CMake officially supports `lwyu`, `CMAKE_LINK_WHAT_YOU_USE` variable will be set accordingly and CMake will trigger lwyu on every link action.

```
project(my-awesome-project)                          # Declare a new project
set(my-awesome-project_TOOLCONF_USE_LWYU TRUE)       # Locate iwyu and use it
if available.
include(.hadouken/hadouken.cmake)                    # Use hadouken
# No further action is required.
```

No extra tools required for this functionality besides the toolchain linker you are using.

CCache

Controlled by `<project_name>_TOOLCONF_USE_CCACHE` option.

Locate `ccache` in environment, if available. The status will be printed to the stdout.

CCache speeds up compilation times by caching the compilation artifacts, and reusing them on scenarios when re-compilation would produce the exact same artifact.

```
project(my-awesome-project)                          # Declare a new project
set(my-awesome-project_TOOLCONF_USE_CCACHE TRUE)     # Locate ccache and use
it if available.
include(.hadouken/hadouken.cmake)                    # Use hadouken
# No further action is required.
```

You can install ccache either by downloading it from the official site https://ccache.dev/download.html or you can get from you package manager.

Ubuntu/Debian

```
    sudo apt-get update
    sudo apt-get install ccache
```

Core modules

Core modules are wrappers which are used commonly in CMake based projects.

MakeCompilationUnit

CMake module which provides a function to gather the files which will be compiled for the project. Requires project folder to be structured in `include/` `src/` form. To gather compilation unit of a project, simply invoke `hdk_make_compilation_unit` function at project scope. This will define 3 CMake variables in project scope.

```
    ${HEADERS} # Content of the include/ folder. Paths are relative to include/
folder.
    ${SOURCES} # Content of the src/ folder. Paths are relative to src/ folder.
    ${COMPILATION_UNIT} # Content of the both include/ and src/ folders.
```

MakeTarget

CMake module which provides functions to create several CMake target types with less typing. To create a target using MakeTarget, simply invoke `make_target` function with desired arguments. The `make_target` function operates on projects, so it must be invoked after `project()` CMake command is invoked for the desired project.

All arguments are optional, except the `TYPE` argument.

The created target's compilation unit will be automatically gathered using AutoCompilationUnit module, so the project must be structured accordingly. Otherwise, you have to specify source and header files manually by populating `SOURCES` and `HEADERS` arguments.

```
    make_target(TYPE <EXECUTABLE|STATIC|SHARED|UNIT_TEST|INTERFACE|BENCHMARK>
                [LINK [<target_name|library_name> ...]]
                [COMPILE_OPTIONS [<options> ...]]
                [COMPILE_DEFINITIONS[<definitions> ...]]
                [DEPENDS [<target_name> ...]]
                [SUFFIX <name_suffix>]
                [NAME <desired_name>]
                [OUTPUT_NAME <desired_output_name>]
                [PREFIX <name_prefix>]
                [INCLUDES [<include_path> ...]]
                [PARTOF [<target_name> ...]]
                [SOURCES [<source_path> ...]]
                [HEADERS [<header_path> ...]]
                [WITH_COVERAGE]
                [WITH_INSTALL]
                [ARGUMENTS] [<argument-1> <argument-2> ...<argument-n>]
                [COVERAGE_TARGETS [<target_name> ...]]
                [COVERAGE_LCOV_FILTER_PATTERN <lcov-pattern>]
                [COVERAGE_GCOVR_FILTER_PATTERN <gcovr-pattern>]
                [EXPOSE_PROJECT_METADATA]
                [PROJECT_METADATA_PREFIX <prefix>]
                [NO_AUTO_COMPILATION_UNIT]
                [WORKING_DIRECTORY <working_directory>]
    )
```

TYPE (required)

Type of the target to be created. Possible values are;

EXECUTABLE

Executable target (add_executable)

Example:

```
project(proj.executable VERSION 0.1.0)

make_target(
    # Specify type as EXECUTABLE file
    TYPE EXECUTABLE
    # Link `proj.component-q.static` target to the executable
    LINK proj.component-q.static
)
```

STATIC

Static library target (add_library(STATIC))

Example:

```
project(proj.component-q VERSION 0.1.0.0 LANGUAGES CXX)

make_target(
    # Specify type as STATIC library
    TYPE STATIC
    # Append .static suffix to the target name
    SUFFIX .static
    # Link `proj.component-x` and `proj-component-y` to target
    # as PUBLIC (propogates to other targets using this target)
    LINK PUBLIC proj.component-x proj.component-y
)
```

SHARED

Shared library target (add_library(SHARED))

Example:

```
project(proj.component-q VERSION 0.1.0.0 LANGUAGES CXX)

make_target(
    # Specify type as SHARED library
    TYPE SHARED
    # Append .shared suffix to the target name
    SUFFIX .shared
    # Link `proj.component-x` and `proj-component-y` to target
    # as PUBLIC (propogates to other targets using this target)
    LINK PUBLIC proj.component-x proj.component-y
)
```

UNIT_TEST

Unit test target. Performs additional steps to make unit test discoverable by CTest. Also, target is automagically linked to `google test` and `google mock` libraries for convenience.

Example:

```
project(proj.component-x.test VERSION 0.1.0 LANGUAGES CXX)

# Create an unit test target with the name of
# `project.component-x.test.feature-1`
make_target(
    # Create an unit test target
    TYPE UNIT_TEST
    # Specify the source file(s) of the target
    SOURCES ${CMAKE_CURRENT_SOURCE_DIR}/ut_feature_1.cpp
    # Link the subject which will be tested
    LINK proj.component-x
    # Append `.feature-1` suffix to the created target
    SUFFIX .feature-1
    # Disable auto source/header file gathering for this target
    NO_AUTO_COMPILATION_UNIT
)

# Create an unit test target with the name of
# `project.component-x.test.feature-2`
make_target(
    # Create an unit test target
    TYPE UNIT_TEST
    # Specify the source file(s) of the target
    SOURCES ${CMAKE_CURRENT_SOURCE_DIR}/ut_feature_2.cpp
    # Link the subject which will be tested
    LINK proj.component-x
    # Append `.feature-2` suffix to the created target
    SUFFIX .feature-2
    # Disable auto source/header file gathering for this target
    NO_AUTO_COMPILATION_UNIT
)
```

INTERFACE

Header-only library target (add_library(INTERFACE))

BENCHMARK

Benchmark target. Similar to UNIT_TEST target, target is automagically linked to `google benchmark` library for convenience.

Example:

```
project(proj.component-x.bench VERSION 0.1.0 LANGUAGES CXX)

# Create an unit test target with the name of
# `project.component-x.bench`
make_target(
    # Create a benchmark target
    TYPE BENCHMARK
    # Specify the source file(s) of the target
```

```
        SOURCES ${CMAKE_CURRENT_SOURCE_DIR}/bm_component_x.cpp
        # Link the subject which will be tested
        LINK proj.component-x
        # Disable auto source/header file gathering for this target
        NO_AUTO_COMPILATION_UNIT
    )
```

LINK (optional)

The list of libraries/targets to link to the created target. The names which are target names are resolved by CMake and the actual .so/.a files will be linked according to the target type. Link target's public/interface include path also will be appended to the created target, so the public interface of the target can be used in created target's source files by using angle brackets (`<>`).

Example:

```
    make_target(TYPE EXECUTABLE LINK pthread my-awesome-target)
```

COMPILE_OPTIONS (optional)

The list of parameters to pass to the compiler when compiling the source code of the created target.

Example:

```
    make_target(TYPE STATIC COMPILE_OPTIONS -O3)
```

COMPILE_DEFINITIONS (optional)

The list of macro definitions to pass to the compiler when compiling the source code of the created target.

Example:

```
    make_target(TYPE STATIC COMPILE_DEFINITIONS USING_HADOUKEN=1)
```

DEPENDS (optional)

The list of dependencies of created target. In order to satisfy dependency, the specified targets will be processed before the created target is processed.

Example:

```
    project(proj.application VERSION 0.1.0 LANGUAGES CXX)

    # This command would create an executable target with the name of
    # `proj.application` without the explicit name specification.
    # When `NAME` paramater is available, the value of `NAME` parameter
    # will be used as target name instead.
    make_target(
        # Create a shared library target
        TYPE EXECUTABLE
        # This will ensure `proj.deps.boost` target is configured
        # before configuring the `proj.application` target.

        # This is useful for externally configured projects, which are
```

```
        # declared as custom targets via `add_custom_target`(*). Since custom
        # targets cannot be linked to a target, we somehow need to indicate
        # that `proj.deps.boost` should be configured before the target.
        # Because, in order to be able to link libboost_system.a and
libboost_thread.a
        # libraries, they must have been already compiled before the compilation
of proj.application begins.

        # Keep in mind that a `LINK` ed CMake target are also implicitly
        # dependency of the target, so there is no need to specify that
        # target in DEPENDS section again.
        DEPENDS proj.deps.boost proj.deps.spdlog
        # Explicitly specify the name
        NAME my_awesome_application
        # Specify include paths of linked libraries
        INCLUDES /usr/lib/boost/include /usr/lib/spdlog/include
        # Link libraries
        LINK libboost_system.a libboost_thread.a libspdlog.a
    )

    # The target will be named as `my_awesome_application`.
    # (*) Proper way of declaring an external library as a CMake target is,
    # to use IMPORTED targets. See:
    # <https://cmake.org/cmake/help/latest/command/add_library.html#imported-
libraries>
    # for details
```

NAME (optional)

The user-defined name for the created target. By default, name will be automatically determined from previous project call. If you want to give a different name, you can give by specifying this argument.

Example:

```
    project(proj.application VERSION 0.1.0 LANGUAGES CXX)

    # This command would create an executable target with the name of
    # `proj.application` without the explicit name specification.
    # When `NAME` paramater is available, the value of `NAME` parameter
    # will be used as target name instead.
    make_target(
        # Create a shared library target
        TYPE EXECUTABLE
        # Explicitly specify the name
        NAME my_awesome_application
        # Specify include paths of linked libraries
        INCLUDES /usr/lib/boost/include /usr/lib/spdlog/include
        # Link libraries
        LINK libboost_system.a libboost_thread.a libspdlog.a
    )

    # The target will be named as `my_awesome_application`.
```

OUTPUT_NAME (optional)

The user-defined output name for the created target. By default, name will be automatically determined from the target name. If you want to give a different name for the output (e.g. binaries), you can give by specifying this argument.

Example:

```
project(proj.application VERSION 0.1.0 LANGUAGES CXX)

# This command would create an executable target with the name of
# `proj.application` without the explicit name specification.
# When `NAME` paramater is available, the value of `NAME` parameter
# will be used as target name instead.
make_target(
    # Create a shared library target
    TYPE EXECUTABLE
    # Explicitly specify the name
    NAME my_awesome_application
    # Explicitly specify output name
    OUTPUT_NAME mapp
    # Specify include paths of linked libraries
    INCLUDES /usr/lib/boost/include /usr/lib/spdlog/include
    # Link libraries
    LINK libboost_system.a libboost_thread.a libspdlog.a
)

# The output binary name will be mapp.exe
```

PREFIX (optional)

The prefix to be prepended to the created target name.

```
project(my-awesome-component)

make_target(TYPE EXECUTABLE PREFIX test.)
# The target name will be `test.my-awesome-component`
```

SUFFIX (optional)

The suffix to be appended to the created target name. Usually used to distinguish between subtypes of the same target, or target with different configurations.

```
project(my-awesome-component)

make_target(TYPE SHARED SUFFIX .shared)
# The target name will be `my-awesome-component.shared`
make_target(TYPE SHARED SUFFIX .shared-pic COMPILE_OPTIONS -fPIC)
# The target name will be `my-awesome-component.shared-pic
make_target(TYPE STATIC SUFFIX .static)
# The target name will be `my-awesome-component.static`
```

INCLUDES (optional)

List of include paths to be added to the include directory of the created target. Useful when `LINK`ed library is not a CMake target.

Example:

```
project(proj.component-z VERSION 0.1.0 LANGUAGES CXX)

# Create a shared library target with name of
# `proj.component-z`
make_target(
    # Create a shared library target
    TYPE SHARED
    # Specify include paths of linked libraries
    INCLUDES /usr/lib/boost/include /usr/lib/spdlog/include
    # Link libraries
    LINK libboost_system.a libboost_thread.a libspdlog.a
)
```

PARTOF (optional)

Reverse of DEPENDS, makes specified argument (expected to be a target) to be dependent to the created target.

Example:

```
project(my-awesome-component)

make_target(TYPE SHARED PARTOF special-component)
# special-component now depends on my-awesome-component
```

SOURCES (optional)

Extra source files to be appended to created target's sources.

Example:

```
project(proj.component-y VERSION 0.1.0 LANGUAGES CXX)

# Create a static library target with name of
# `proj.component-y`
make_target(
    # Create a static library target
    TYPE STATIC
    # Specify additional source code files for the target
    SOURCES /home/extra/extra_file_1.cpp /home/extra/extra_file_2.cpp
    # Link dependencies
    LINK proj.component-x
)
# Without `NO_AUTO_COMPILATION_UNIT` parameter is specified,
# auto source gathering will still take place. Specified `SOURCES`
# will be appended to auto-gathered source file list.
```

HEADERS (optional)

Extra header files to be appended to created target's headers.

Example:

```
    project(proj.component-y VERSION 0.1.0 LANGUAGES CXX)

    # Create a static library target with name of
    # `proj.component-y`
    make_target(
        # Create a static library target
        TYPE STATIC
        # Specify additional source code files for the target
        HEADERS /home/extra/include/extras/extra_1.hpp
/home/extra/include/extras/extra_2.hpp
        # Link dependencies
        LINK proj.component-x
    )
    # Without `NO_AUTO_COMPILATION_UNIT` parameter is specified,
    # auto header gathering will still take place. Specified `HEADERS`
    # will be appended to auto-gathered header file list.
```

ARGUMENTS (optional)

Arguments to be forwarded to the created target. `ARGUMENTS` will only make sense for targets which invoke an executable or command. The `ARGUMENTS` parameter is only defined for the types specified below:

- UNIT_TEST (ARGUMENTS will be appended to the add_test COMMAND part)

Example:

```
    project(proj.component-x.test VERSION 0.1.0 LANGUAGES CXX)

    # Create an unit test target with the name of
    # `project.component-x.test.component.x`
    make_target(
        # Create an unit test target
        TYPE UNIT_TEST
        # Specify the source file(s) of the target
        SOURCES ${CMAKE_CURRENT_SOURCE_DIR}/ut_component_x.cpp
        # Link the subject which will be tested
        LINK proj.component-x
        # Append `.component-x` suffix to the created target
        SUFFIX .component-x
        # Enable code coverage instrumentation
        WITH_COVERAGE
        # Arguments which will be passed to the add_test COMMAND part.
        ARGUMENTS --gtest_shuffle --
gtest_output=xml:${PROJECT_BINARY_DIR}/ut_reports/${PROJECT_NAME}.xml
        # Disable auto source/header file gathering for this target
        NO_AUTO_COMPILATION_UNIT
    )
```

WITH_COVERAGE (optional)

Generate code coverage report for the project (useful for unit test targets)

Example:

```
    project(proj.component-x.test VERSION 0.1.0 LANGUAGES CXX)

    # Create an unit test target with the name of
    # `project.component-x.test.component.x`
    make_target(
        # Create an unit test target
        TYPE UNIT_TEST
        # Specify the source file(s) of the target
        SOURCES ${CMAKE_CURRENT_SOURCE_DIR}/ut_component_x.cpp
        # Link the subject which will be tested
        LINK proj.component-x
        # Append `.component-x` suffix to the created target
        SUFFIX .component-x
        # Enable code coverage instrumentation
        WITH_COVERAGE
        # Disable auto source/header file gathering for this target
        NO_AUTO_COMPILATION_UNIT
    )
```

WITH_INSTALL (optional)

Automatically generate install step for the target.

Example:

```
    project(proj VERSION 0.1.0 LANGUAGES CXX)

    # Create a static library target with the name of
    # `project.component-x`
    make_target(
        # Create an unit test target
        TYPE STATIC
        # Append `.component-x` suffix to the created target
        SUFFIX .component-x
        # Create installation step for project
        WITH_INSTALL
    )
```

COVERAGE_TARGETS (optional)

Requires `WITH_COVERAGE` to be set first. Injects required instrumentation parameters (-fprofile-arcs -ftest-coverage) and link libraries (-lgcov) to the specified targets. Without this, coverage reports may or may not include the desired target's coverage rate.

```
    project(proj.component-y.test VERSION 0.1.0 LANGUAGES CXX)

    make_target(
        TYPE UNIT_TEST
        SOURCES ${CMAKE_CURRENT_SOURCE_DIR}/ut_component_y.cpp
        LINK proj.component-y
        SUFFIX .component-y
```

```
    WITH_COVERAGE
    COVERAGE_TARGETS proj.component-y
    NO_AUTO_COMPILATION_UNIT
)
 # -fprofile-arcs and -ftest-coverage will be injected to
 # proj.component-y's compilation flags via
 # target_compile options and also -lgcov will be added to
 # target's link libraries via target_link_libraries.
```

COVERAGE_LCOV_FILTER_PATTERN (optional)

Requires `WITH_COVERAGE` to be set first. The argument will be forwarded to `lcov`'s `--extract` parameter, which allows to filter out coverage results.

Example:

```
project(hdktest.lib.test)

make_target(
    TYPE UNIT_TEST
    SOURCES ut_lib.cpp
    LINK hdktest.lib
    WITH_COVERAGE
    COVERAGE_TARGETS hdktest.lib
    COVERAGE_LCOV_FILTER_PATTERN "my_unit*.cpp"
    NO_AUTO_COMPILATION_UNIT
)
# Only files matching the pattern will be included on lcov report
```

COVERAGE_GCOVR_FILTER_PATTERN (optional)

Requires `WITH_COVERAGE` to be set first. The argument will be forwarded to `gcovr`'s `--f` parameter, which allows to filter out coverage results.

Example:

```
project(hdktest.lib.test)

make_target(
    TYPE UNIT_TEST
    SOURCES ut_lib.cpp
    LINK hdktest.lib
    WITH_COVERAGE
    COVERAGE_TARGETS hdktest.lib
    COVERAGE_GCOVR_FILTER_PATTERN "${PROJECT_SOURCE_DIR}/.*cpp"
    NO_AUTO_COMPILATION_UNIT
)
# Only files matching the pattern will be included on gcovr report
# Be aware that GCOVR's filter is a regex pattern. Use forward slash `/` as
# path separator.
```

EXPOSE_PROJECT_METADATA (optional)

Defines the following C/C++ preprocessor macros for the target created.

```
<prefix>MODULE_NAME
<prefix>MODULE_DESC
<prefix>MODULE_VERSION_MAJOR
<prefix>MODULE_VERSION_MINOR
<prefix>MODULE_VERSION_PATCH
<prefix>MODULE_VERSION_TWEAK
```

Defined macros will only be accessible (private) for the source/header files of the project.

`<prefix>` can be specified by supplying `PROJECT_METADATA_PREFIX` argument.

Example:

```
    project(hdktest.exe VERSION 1.0.3.3 DESCRIPTION "Very interesting project")

    make_target(
        TYPE EXECUTABLE
        SOURCES main.cpp
        NO_AUTO_COMPILATION_UNIT
        EXPOSE_PROJECT_METADATA
        PROJECT_METADATA_PREFIX HDKTEST_EXE_
    )
```

main.cpp:

```
#include <cstdio>

auto main(void) -> int{
    std::printf("%s\n", HDKTEST_EXE_MODULE_NAME);
    std::printf("%s\n", HDKTEST_EXE_MODULE_DESC);
    std::printf("%s.%s.%s\n", HDKTEST_EXE_MODULE_VERSION_MAJOR,
HDKTEST_EXE_MODULE_VERSION_MINOR, HDKTEST_EXE_MODULE_VERSION_PATCH,
HDKTEST_EXE_MODULE_VERSION_TWEAK);
}

// Expected output of the code above:
/*
    hdktest.exe
    Very interesting project
    1.0.3.3
*/
```

The information is gathered from the `project()` info.

PROJECT_METADATA_PREFIX (optional)

Requires `EXPOSE_PROJECT_METADATA` to be set.

Prefix to prepend created project metadata macros.

NO_AUTO_COMPILATION_UNIT (optional)

Disable automatic compilation unit gathering from `include/` and `src/` directories.

WORKING_DIRECTORY (optional)

Specify working directory for the created target. It is only valid for the following target types:

- UNIT_TEST (when executed by CTest)

```
    project(hdktest.lib.test)

    make_target(
        TYPE UNIT_TEST
        SOURCES ut_lib.cpp
        LINK hdktest.lib
        WITH_COVERAGE
        COVERAGE_TARGETS hdktest.lib
        COVERAGE_GCOVR_FILTER_PATTERN "${PROJECT_SOURCE_DIR}/.*cpp"
        NO_AUTO_COMPILATION_UNIT
        WORKING_DIRECTORY ${PROJECT_SOURCE_DIR} # Set working directory to
${PROJECT_SOURCE_DIR}
    )
```

TargetProperties

CMake module which provides information about a target's properties.

hdk_print_target_properties(<target_name>)

Prints all properties of the target specified. Useful for debug purposes.

Example:

```
    project(my_lib)
    make_target(TYPE STATIC)

    # Print properties of the target
    hdk_print_target_properties(my_lib)
```

Possible output:

```
    [cmake] -- ------------------------------------------------------------
    [cmake] -- Printing properties of target `my_lib`
    [cmake] -- ------------------------------------------------------------
    [cmake] --    my_lib AUTOGEN_ORIGIN_DEPENDS = ON
    [cmake] --    my_lib AUTOMOC_COMPILER_PREDEFINES = ON
    [cmake] --    my_lib AUTOMOC_MACRO_NAMES = Q_OBJECT
    [cmake] --    my_lib AUTOMOC_PATH_PREFIX = OFF
    [cmake] --    my_lib BINARY_DIR = /workspace/build
    [cmake] --    my_lib BUILD_WITH_INSTALL_RPATH = OFF
    [cmake] --    my_lib IMPORTED = FALSE
    [cmake] --    my_lib IMPORTED_GLOBAL = FALSE
    [cmake] --    my_lib INCLUDE_DIRECTORIES = /workspace/include/
    [cmake] --    my_lib INSTALL_RPATH =
    [cmake] --    my_lib INSTALL_RPATH_USE_LINK_PATH = OFF
    [cmake] --    my_lib INTERFACE_INCLUDE_DIRECTORIES = /workspace/include/
    [cmake] --    my_lib INTERFACE_LINK_LIBRARIES = gcov
```

```
[cmake] --    my_lib LINK_LIBRARIES = gcov
[cmake] --    my_lib NAME = my_lib
[cmake] --    my_lib PCH_WARN_INVALID = ON
[cmake] --    my_lib RULE_LAUNCH_COMPILE = /usr/bin/ccache
[cmake] --    my_lib SKIP_BUILD_RPATH = OFF
[cmake] --    my_lib SOURCES = /workspace/source.cpp
[cmake] --    my_lib SOURCE_DIR = /workspace
[cmake] --    my_lib TYPE = STATIC_LIBRARY
[cmake] --    my_lib UNITY_BUILD_BATCH_SIZE = 8
[cmake] --    my_lib UNITY_BUILD_MODE = BATCH
[cmake] --    my_lib VISIBILITY_INLINES_HIDDEN = OFF
```

EnvironmentUtilities

CMake module which contains utility functions for interacting with the environment.

hdk_read_environment_file(<env_filename> <prefix> [optional] <suffix> [optional])

Reads a dotenv (.env) file, and declares a CMake variable for each environment variable in it.

Example:

.env file content (test.env):

```
# this is a comment an will be skipped
HADOUKEN="is great"
VERY="nice"
```

```
hdk_read_environment_file("test.env", PRE_ _SUF)
# Now the environment variables in .env file are declared as
# `PRE_HADOUKEN_SUF` and `PRE_VERY_SUF`, containing the values specified
# in the file.
```

TargetUtilities

CMake module which contains utility functions for interacting with CMake targets.

hdk_copy_target_artifact_to(TARGET_NAME <target> DESTINATION <destination_path> STEP <step>)

Copy the artifact produced by `<target>` to `<destination_path>` on `<step>`. Step can be `PRE_BUILD, PRE_LINK or POST_BUILD`.

Example:

```
project(my_executable)

make_target(TYPE EXECUTABLE)

hdk_copy_target_artifact_to(
    STEP POST_BUILD
    TARGET_NAME my_executable
    DESTINATION /tmp
)

# Executable produced by my_executable target will be copied to /tmp
# after target `my_executable` gets built.
```

Git

This module has several functions to invoke git functionality from CMake.

hdk_git_get_branch_name(OUTPUT_VARIABLE DIRECTORY <dir>)

A function to retrieve active branch name for the project. Sets `OUTPUT_VARIABLE` variable to current branch name.

Example:

```
    hdk_git_get_branch_name(CURRENT_BRANCH DIRECTORY ${PROJECT_SOURCE_DIR})
    # ${CURRENT_BRANCH} now contains the branch name of the git repository in
${PROJECT_SOURCE_DIR}
```

hdk_git_get_head_commit_hash(OUTPUT_VARIABLE DIRECTORY <dir>)

A function to retrieve commit hash of the head for the git repository located in specified directory. Sets `OUTPUT_VARIABLE` variable to head commit hash of git repository located at `<dir>`.

Example:

```
    hdk_git_get_head_commit_hash(HEAD_COMMIT DIRECTORY ${PROJECT_SOURCE_DIR})
    # ${HEAD_COMMIT} now contains the head commit hash of the git repository in
${PROJECT_SOURCE_DIR}
```

hdk_git_is_worktree_dirty(OUTPUT_VARIABLE DIRECTORY <dir>)

A function to retrieve whether current work tree for the git repository located in specified directory is dirty. Sets `OUTPUT_VARIABLE` variable to dirtiness status of git repository located at `<dir>`.

Example:

```
    hdk_git_is_worktree_dirty(IS_DIRTY DIRECTORY ${PROJECT_SOURCE_DIR})
    # ${IS_DIRTY} now contains whether worktree of the git repository in
${PROJECT_SOURCE_DIR} is dirty or not.
```

hdk_git_get_config(OUTPUT_VARIABLE DIRECTORY <dir> CONFIG_KEY <key_to_be_retrieved>)

A function to retrieve a git configuration by its' key name. Sets `OUTPUT_VARIABLE` variable to value of the configuration value specified by `CONFIG_KEY` in git repository located at `<dir>`.

Example:

```
    hdk_git_get_config(USER_MAIL DIRECTORY ${PROJECT_SOURCE_DIR} CONFIG_KEY
user.email)
    # ${USER_MAIL} now contains the user.email configuration value of the git
repository located in ${PROJECT_SOURCE_DIR} folder.
```

hdk_git_get_tag(OUTPUT_VARIABLE DIRECTORY <dir> COMMIT <commit_hash> [optional])

A function to retrieve all tags pointing to `<commit_hash>` in git repository located at `<dir>`. Commit hash is defaulted to HEAD if not specified. Sets `OUTPUT_VARIABLE` to found tags.

```
    hdk_git_get_tag(TAGS DIRECTORY ${PROJECT_SOURCE_DIR})
    # ${TAGS} now contains the all tags pointing to head in git repository
located at ${PROJECT_SOURCE_DIR} folder.
```

hdk_target_needs_git_lfs_files(TARGET <target> LFS_ROOT <lfs_root> LFS_FILE_LIST <lfs_files...>)

A function to create a dependency to `<target>` target which fetches all git lfs files specified in `<lfs_files>` from the git lfs repository at `<lfs_root>` on CMake configuration.

Example:

```
    project(example.unit_test)

    # This unit test needs `${PROJECT_SOURCE_DIR}/data/json-files/geo-
location.json`
    # `${PROJECT_SOURCE_DIR}/data/bin-files/archive.bin` files to be present in
order
    # to run.
    make_target(
        TYPE UNIT_TEST
        SOURCES ${test_sources} ut_example.cpp
    )

    # This will cause `${PROJECT_SOURCE_DIR}/data/json-files/geo-location.json`
and
    # `${PROJECT_SOURCE_DIR}/data/bin-files/archive.bin` to be fetched from git
large
    # file storage on CMake configure step.
    # This ensures the files needed by the unit test are present when unit test
is
    # being build and being run.
    hdk_target_needs_git_lfs_files(
        # LFS file dependent target
        example.unit_test
        # LFS repository path
        ${PROJECT_SOURCE_DIR}/data
        # LFS file list (semicolon separated, one file per line)
        [=[
        json-files/geo-location.json;
        bin-files/archive.bin
        ]=]
    )
```

hdk_git_metadata_as_compile_defn(PREFIX <prefix> [optional] DIRECTORY <directory> [optional])

A function to export some important version control (git) variables as compile time definitions (preprocessor macros). Exported variables will be available to the all compiled code under the project. Variables are gathered from git repository located at `<directory>`. When no explicit directory is given as parameter, DIRECTORY will be defaulted to `${CMAKE_CURRENT_SOURCE_DIR}`.

Exported variables are as follows:

- `<prefix>`GIT_BRANCH_NAME
- `<prefix>`GIT_COMMIT_ID
- `<prefix>`GIT_WORKTREE_DIRTY

- `<prefix>`GIT_AUTHOR_NAME (current git author name)
- `<prefix>`GIT_AUTHOR_EMAIL (current git author e-mail)

Example:

```
    hdk_git_metadata_as_compile_defn(PREFIX MYPROJ_)
```

Later on, in code

```
    #include <cstdio>

    auto main(void) -> int {
        std::printf("%s\n", MYPROJ_GIT_BRANCH_NAME);
        std::printf("%s\n", MYPROJ_GIT_COMMIT_ID);
        std::printf("%s\n", MYPROJ_GIT_WORKTREE_DIRTY);
        std::printf("%s\n", MYPROJ_GIT_AUTHOR_NAME);
        std::printf("%s\n", MYPROJ_GIT_AUTHOR_EMAIL);
    }
```

hdk_git_print_status()

Print branch, commit hash and dirtiness status of the git repository specified in DIRECTORY argument.

```
    hdk_git_print_status()
    # Possible output to stdout might be:
    # VCS Status
    #   Branch: master
    #   tag: v.0.14.0
    #   Commit: 8f14b151ec1aba749f5ac579b6f4f8209738dee5
    #   Dirty: false
```

Conan

Official conan module provided by JFrog. Provides `conan_cmake_run` function, which allows conan to be run from CMake.

```
cmake_minimum_required(VERSION 3.16)

project(dummy)

include(.hadouken/hadouken.cmake)

# Fetch boost 1.70.0 from conan repositories.
# Build any dependencies which are missing on repository as pre-built
# Create cmake targets for resolved dependencies
# Created dependencies can be accessed from CONAN_PKG:: target. (eg.
CONAN_PKG::boost)
conan_cmake_run(REQUIRES boost/1.70.0
                BASIC_SETUP CMAKE_TARGETS
                BUILD missing)
```

HardwareConcurrency

A CMake module to retrieve logical core count of the machine. Sets `${HARDWARE_CONCURRENCY}` CMake variable.

Build variant determination and compiler diagnostic flags (warnings)

This modules are designed for determining the common and compiler specific compilation parameters according to the build variant. The base functionality is provided by `BuildVariant.cmake` module, which determines the compiler (using CMakeDetermine[C/CXX]Compiler CMake modules), sets the build variant if not set by the user (default is RelWithDebInfo), adjusts build variant specific compilation parameters, then includes the compiler specific module. Compiler-specific modules are suffixed with toolchain/compiler name and contains the distinct compilation flags for that compiler. These modules are used to enable some useful warnings, and also turn warnings into errors in release builds.

Warnings are important for keeping a bug-free and clean code base. Therefore, hadouken by default passes "-Wall" on debug and "-Wall -Werror" on release configuration to the compiler. On top of that, hadouken will pass additional useful warning flags depending on compiler version. These flags are contained in compiler specific diagnostic modules.

Hadouken currently supports `GCC` and `Clang` specific diagnostics flag modules. These modules enables several important compiler warning flags. Diagnostic flag modules are compiler version aware, therefore the compiler warning flags which requires compiler version higher than current compiler version will not be enabled on configuration.

See `DiagnosticFlags_GCC.cmake` and `DiagnosticFlags_Clang.cmake` for inspecting which version-specific, extra warning flags are being passed to the compiler.

Finders

Finders are located in `cmake/modules/find` folder. This folder is designated for CMake finders, which allows user to call `find_program` and locate executable/library/header files in environment. CMake has a quite exhaustive list of finder modules, but still lacks some. This folder will contain the finder modules which CMake does not contain.

The list of finder modules are provided below;

- FindGMock.cmake (google mock)

Feature check

Feature check modules are located in `cmake/modules/check` folder. They're driven by `FeatureCheck.cmake` file. The purpose of a feature check module is, to determine whether a feature, a library, a header (etc..) is available in compilation environment or not. This is done by compiling a small C++ code regarding the subject. If the file compiles successfully, the subject is said to be available in the environment. Compilation is done by invoking a special CMake function named `check_cxx_source_compiles`. This function takes a source code as an argument, tries to compile it, and sets a user-named variable to indicate compilation status. An example feature check module is shown below;

```
set(CMAKE_REQUIRED_QUIET true)              # Do not print anything to stdout
regarding feature check
```

```
set(CMAKE_REQUIRED_LIBRARIES "-lpthread")   # Feature check dependencies (linkage
against pthread is required for this scenario)

# A code which uses some of the `pthread` functions in order to determine
# whether pthread is available on the compilation environment
check_cxx_source_compiles("
    #include <pthread.h>
    extern \"C\" void* thread_proc(void* arg)
    {
    return arg;
    }
    int main(void)
    {
        pthread_mutex_t mut;
        int result = pthread_mutex_init(&mut, 0);
        if(0 == result)
        {
            result |= pthread_mutex_lock(&mut);
            result |= pthread_mutex_unlock(&mut);
            result |= pthread_mutex_trylock(&mut);
            result |= pthread_mutex_unlock(&mut);
            result |= pthread_mutex_destroy(&mut);

            pthread_t t;
            int r = pthread_create(&t, 0, &thread_proc, 0);
            // result |= r;
            if(r == 0)
            {
                //
                // If we can create a thread, then we must be able to join to it:
                //
                void* arg;
                r = pthread_join(t, &arg);
                result |= r;
            }
        }
        return result;
    }
    "
    ${HDK_ROOT_PROJECT_NAME_UPPER}_FEATURE_HAS_PTHREADS
)
# <project_name>_FEATURE_HAS_PTHREADS is set to true or false after this,
indicating the compilation status.
```

The result variable has a specific naming convention too. `FeatureCheck.cmake` expects the result variable to be defined in a specific way: (the actual check code is `if (NOT ${HDK_ROOT_PROJECT_NAME_UPPER}_HAS_${ARG_AS_UPPER})`)

`${HDK_ROOT_PROJECT_NAME_UPPER}` is a CMake variable defined by hadouken which contains the top level project name, converted to uppercase and all non-alphanumeric characters replaced with an underscore (_). The glue `_HAS_` have to come after this. The last word should match with the feature check module (capitalized).

Feature check modules have a specific naming convention. Every feature check module file name has to start with `Check`. The letters between `Check` and `.cmake` determines the feature check module name. The user will use this name to invoke feature check functions.

To perform a feature check, invoke feature_check_assert with desired feature names.

Example:

```
project(my-awesome-project)                              # Declare a new project
include(.hadouken/hadouken.cmake)                        # Use hadouken
# Check if pthreads is available on compilation environment.
# CMake configuration will result in failure if code in feature check macro fails
to compile.
feature_check_assert(Pthreads)
```

You can use Pthreads as a reference when creating new feature check modules.

Internal modules

Hadouken uses some CMake modules for internal usage only.

Banner

This is the CMake module which prints the hadouken's banner to the terminal when invoked.

Log

CMake module which contains some utility functions which are responsible for printing hadouken's log output.

Closing words

Hadouken will continue to evolve, and you may contribute to it. Don't hesitate to issue a PR.

References

VSCode Remote Containers Documentation: https://code.visualstudio.com/docs/remote/containers

Acknowledgements

- vscode developers: for creating such a flexible, versatile code editor and a huge ecosystem. https://github.com/microsoft/vscode
- vscode-remote-containers team: https://github.com/microsoft/vscode-dev-containers
- cmake-conan developers: https://github.com/conan-io/cmake-conan

License