The Beetroot Project Documentation

Release 0.9.0

Adam Ryczkowski

Jul 25, 2019

Contents:

CHAPTER 1

Overview

Beetroot project is a set of CMake macros that set a new, user friendly paradigm of writing CMake code. It targets medium to large CMake deployments and helps by faciliating modularization and parametrization of the project description.

The idea was started in 2017, and was initially motivated by a) the fact, that usually CMake scripts rely heavily on global variables and b) CMake has very poor function parameter handling. This led to an effort to define a set of macros that facilate checking of types of function arguments. After several iterations and several complete redesignings, the current shape of the Beetroot was formed.

Beetroot tries to nudge developers to put their targets definitions and dependency declarations inside CMake functions with a clear API interface, so it is clear on what information each target depends. In return

1. it does a great deal of semantic checks on the user code,
2. allows a lot of flexibility of where and how to put the user CMake code,
3. allows to build any part of the project from anywhere,
4. can automatically turn a project into a superbuild if any of the targets are external.

CHAPTER 2

Steps to start using the Beetroot:

1. Put the file `root.cmake` in the subfolder `cmake` of the root of your project. The file name and path is not hard-coded and can easily be changed. It is there so the Beetroot can be aware where is the root of your project.
2. Put a standard common header in the beginning of the `CMakeLists.txt`, add a call `finalizer()` at the end of it.
3. Write target description files (either `targets.cmake` anywhere in the project tree, or any file with the extension `.cmake` in the subdirectory `cmake/targets` anywhere in the project tree.
4. Refer to the target you want to build inside the `CMakeLists.txt` by function `build_target(<TEMPLATE_NAME> [<ARGUMENTS>])`

2.1 Why Beetroot?

There are many CMake projects that superficially look similar in scope to us. Beetroot is unique in that it aims to replace/deprecate as little of the native CMake commands and idioms as possible, while delivering modern (if not experimental) approach to the build.

Here is a list of known to me at the time of writing conversion projects with its own description.

- https://github.com/ecmwf/ecbuild - A CMake-based build system, consisting of a collection of CMake macros and functions that ease the managing of software build systems
- https://github.com/DevSolar/jaws - Just A Working Setup.

All these projects are "total conversion mods" for CMake - they replace/reimplement bulk of the CMake functions. In order to use them, you need to learn many new commands with their parameters. Although the new systems are arguably better, you still, as a user, will inevitably learn the standard CMake commands anaway. That will happen either from necessity, because your use case was not forseen and you had to implement it in "bare" CMake, or the standard commands are better documented & supported. This may lead to the situation, contrary to the projects' claims, that using the mentioned systems would lead to the situation that requires you to actually learn more CMake, not less[#fs1]_ .

When implementing you project in Beetroot you will use most of the CMake commands as you did before. In particular, you still use `add_library()`, `add_executable()` and `add_custom_target()` to define

your targets. You will access and set the targets' properites with usual `target_compile_definitions()`, `target_compile_options()`, `target_include_directories()` and even `target_sources()`. You may use `target_link_libraries()` or you may defer to the Beetroot calling it automatically for you. You may use `ExternalProject` and `find_package()` or you may use the Beetroot's built-in system of handling them. The only command that is deprecated is `add_subdirectory()` - it is replaced with a far more controllable and less error-prone mechanism that is used to resolve information-dependency on the various parts of the CMake project.

There list will not be complete without the mention of the Artichoke: https://github.com/commontk/Artichoke - "CMake module allowing to easily create a build system on top of ExternalProjects." The project adheres to the "do one thing, but do it well" Unix principle, and should be, at least partially, compatible with the Beetroot.

CHAPTER 3

Beetroot data model

Since you, the reader, is most probably a seasoned programmer, I believe the most straightforward way of introducing you to the Beetroot is through data model.

CMake (among other things) maintain internally an object for each defined target, and at the glance, the Beetroot's equivalent to the CMake target is a "*FEATUREBASE*" object class [1]. *FEATUREBASE* also gets constructed if the user's code does not result in CMake targets (because e.g. it only modifies existing target). Also note that one target definition file (*targets.cmake*) can define multiple FeatureBase objects, because user can put several names in the *ENUM_TEMPLATES* stanza.

Each invocation of a function the requires a build of the target (e.g. *build_target()* or *get_existing_target()*) is internally represented by the object of the type "*INSTANCE*". In Beetroot INSTANCE and FEATUREBASE have one-to-many relationship; each INSTANCE is matched with a single FEATUREBASE, but a single FEATUREBASE may by linked to several INSTANCES.

In plain CMake, this duality does not exist, the sequence of calls *add_library(foo . . .)* and *target_link_libraries(myexec PRIVATE foo)* mean in simple English "build a library *foo*, and then make *myexec* depend on it". The *mytarget* alwyas gets defined or an error produced. On the other hand, in Beetroot one may express dependency relationship as "I need a library foo with a param *FLAG*". This would result in a new copy of library *foo* only if it has not been defined alsewhere with the sole param *FLAG*. It would be more equivalent to the

if(NOT TARGET foo) add_library(foo . . .)

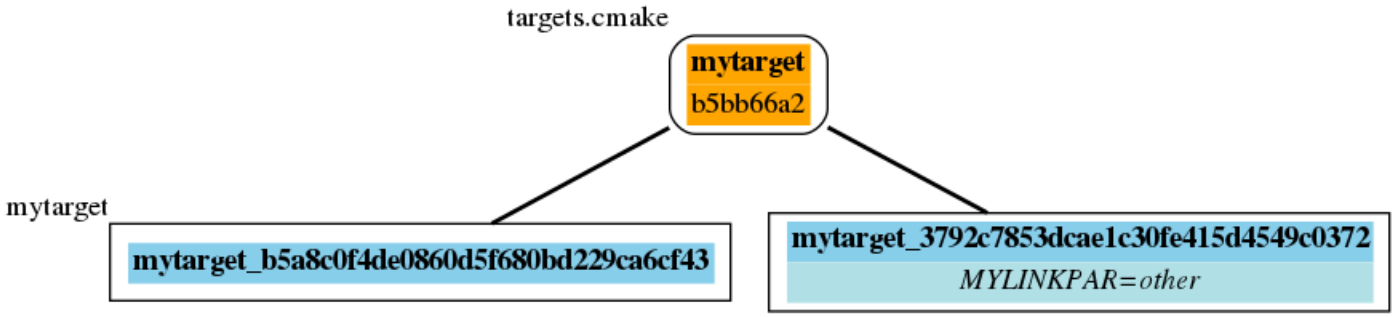
endif() target_link_libraries(myexec PRIVATE foo)

except that such code can be dangerous, if we may not be sure with what options the *foo* was built. Beetroot takes care of that, by matching all the parameters . . . between the pre-existing *foo* and the *foo* definition.

Beetroot also allows for more advanced dependency statement, expressed as "I need whatever library *foo* the project already comes with. Just make sure that among other parameters, it is compiled with a feature indicated by the param FLAG on". In that case Beetroot would gather all requested versions of the library *foo*, decide which are compatible with the specification, then decide if the feature *FLAG* can be added. If not - build a separate copy of *foo*.

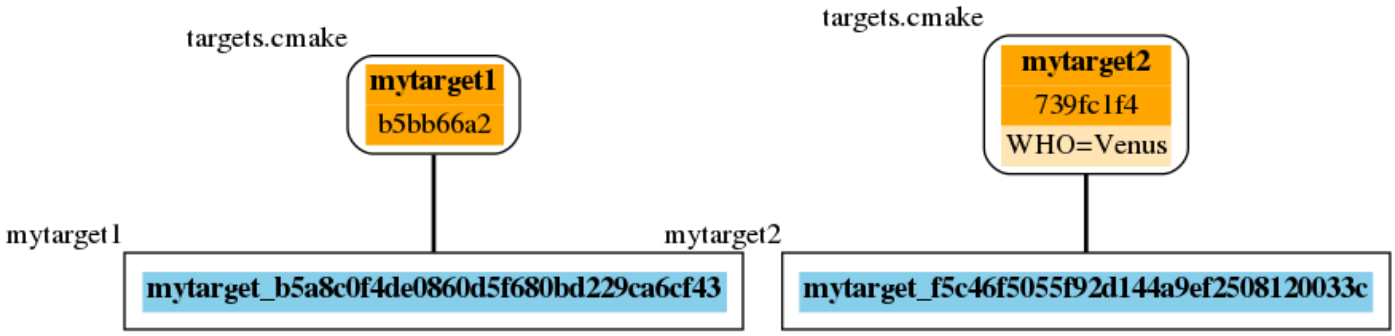
If user requires two targets but the only way they differ is through the link parameters, than it is not required to actually build two copies of them and in such case each *INSTANCE* link to the same *FEATUREBASE*:

build_target(MYTARGET) build_target(MYTARGET MYLINKPAR other)



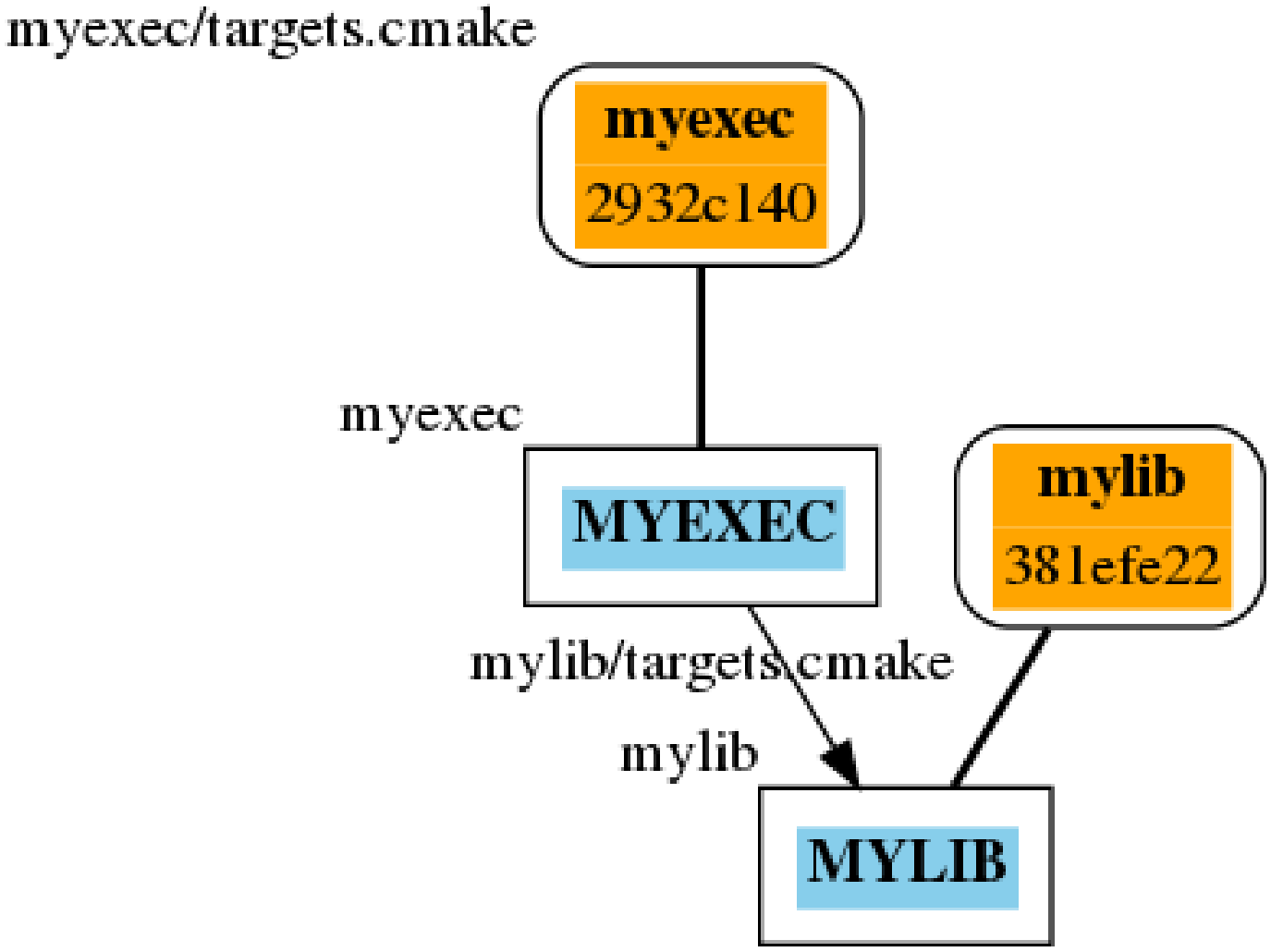
One of the base features of the Beetroot is the ability to build several copies of a target, by simply requiring it with different parameters. If such requirements are mutually incompatible (as is always the case if target parameters differ, but usually not if the features differ, and never with link parameters) than Beetroot will decide to instantiate two distinct FEATUREBASE (and CMake targets) and we will end up with

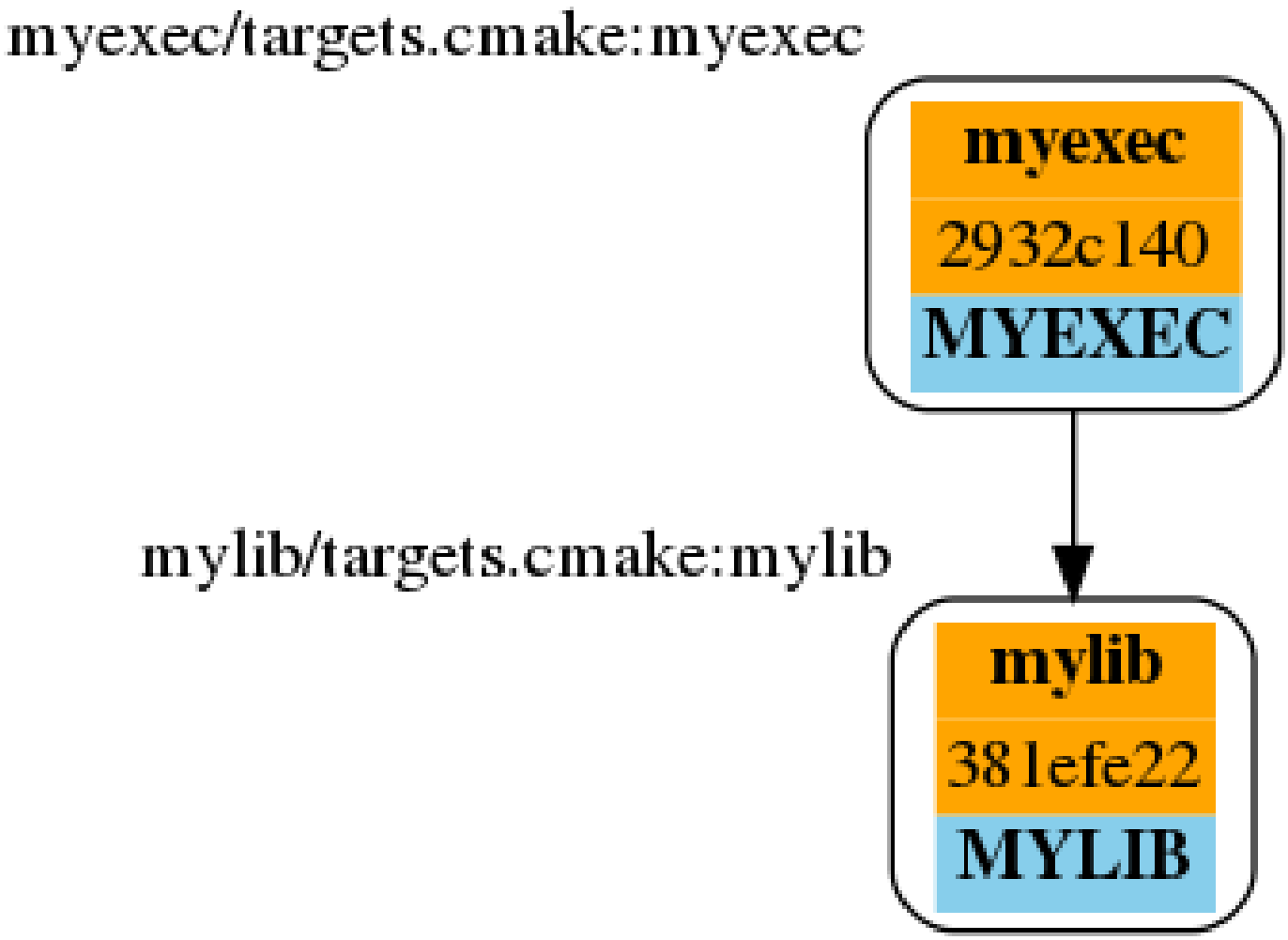
build_target(MYTARGET) build_target(MYTARGET PAR 42)



Because one-to-one relationship between an instance and a target is common, it will be later on depicted with a common box like this:

Dependencies between targets are realized as directed links between *INSTANCES*, like this:


myexec/targets.cmake
myexec
2932c140
myexec
MYEXEC
mylib
381efe22
mylib/targets.cmake
mylib
MYLIB


myexec/targets.cmake:myexec
myexec
2932c140
MYEXEC
mylib/targets.cmake:mylib
mylib
381efe22
MYLIB

CHAPTER 4

How does the Beetroot work?

4.1 Initialization

At the beginning, when Beetroot is loaded, it scans all the subfolders of the project to find target definition files and build a database that maps template/target names to the path of the target definition file.

It also initializes internal variables (held inside global CMake storage) and loads all internal functions.

4.2 Target declaration phase

When the initialization is complete, it reads through the rest of the `CMakeLists.txt` and expects to find calls to `build_target(<TEMPLATE_NAME> [<ARGUMENTS>])`. Each call ultimately triggers user defined function `declare_dependencies()`, where the Beetroot expects to find additional `build_target()` calls and marks the target to be defined later on, because no targets will be defined until the call to the `finalize()` at the end of the `CMakeList.txt`. It calls all encountered `build_target()` recursively.

4.3 Target definition phase

Target definition phase is handled by the call to `finalize()` and this is when targets get defined.

First of all, Beetroot tries to fully declare all targets that were declared with `build_existing_target()`.

Once all targets are declared then Beetroot can finally decide whether it is going to do a super build, or project build.

After that, if it is a project build, it enables all declared languages for all targets in the current build tree.

Finally it defines and links all the relevant targets, by calling `generate_targets()` user function and then `apply_dependency_to_target()` user function and/or `target_link_libraries()` CMake built in function. When on superbuild it will only attempt to define external targets.

`finalize()` returns and by default this should be the end of the `CMakeLists.txt`.

CHAPTER 5

Getting started

5.1 Set up the beetroot

Setting up simply means making the code of the beetroot somehow available to the CMake building process and making sure the Beetroot knows where is our superbuild root. The first part is as simple, as including the `beetroot.cmake` in the beginning of the `CMakeLists.txt`. To specify the root directory of you project you can either set it yourself using `set(SUPERBUILD_ROOT <path>)` *before* including the Beetroot, or relying on the Beetroots' heuristics that assume that the root directory is the topmost directory relative to the `CMakeLists.txt` project that contains `CMakeLists.txt``[#f1]__ . .. [#f1] Beetroot tries to find superproject root using the following algorithm: The root directory is assumed to be the first directory containing ``CMakeLists.txt` that is followed by two immidiate parent directories that are either without this file, or are a root of the filesystem.

Perhaps the simplest way to incorporate the Beetroot into your project is to either clone it into a projects' subdirectory or make it a submodule if you already use git.

We will now go through some tutorial build cases, which can also be found in the `examples` repository.

5.2 The simplest Hello World (`01_simplest_hello`)

We will to start small, with the very simple C++ CMake build.

Suppose this is our `source.cpp`:

```
// 'Hello World!' program
#include <iostream>
#define STRINGIFY2(X) #X
#define STRINGIFY(X) STRINGIFY2(X)
#ifndef WHO
#define WHO World
#endif
```

(continues on next page)

(continued from previous page)

```
int main() {
   std::cout << "Hello " STRINGIFY(WHO) "!" << std::endl;
   return 0;
}
```

For reference, to build it plain CMake, this would be our `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.5)
project(hello_simple)
add_executable(hello_simple source.cpp)
```

So far our file structure would like this:

```
| project_folder
| ├── build
| │   └── ...
| ├── source.cpp
| └── CMakeLists.txt
|
|
```

To be able to compile this code using Beetroot, we need first to make beetroot accessible to our build. Let's clone the beetroot into the subfolder cmake

$ cd project_folder
$ mkdir cmake
$ cd cmake
$ git clone –depth 1 https://github.com/beetroot-project/beetroot.git

Then we put the following content in the `targets.cmake`:

```
set(ENUM_TEMPLATES HELLO_SIMPLE)

function(generate_targets TEMPLATE_NAME)
   add_executable(${TARGET_NAME} "${CMAKE_CURRENT_SOURCE_DIR}/source.cpp")
endfunction()
```

We also need to adjust the `CMakeLists.txt` to have the Beetroot actually executed:

```
cmake_minimum_required(VERSION 3.13)
include(cmake/beetroot/cmake/beetroot)

project(hello_simple)

build_target(HELLO_SIMPLE)

finalize()
```

Our final folder structure should look like this:

```
| project_folder
| ├── build
| │   └── ...
```

(continues on next page)

(continued from previous page)

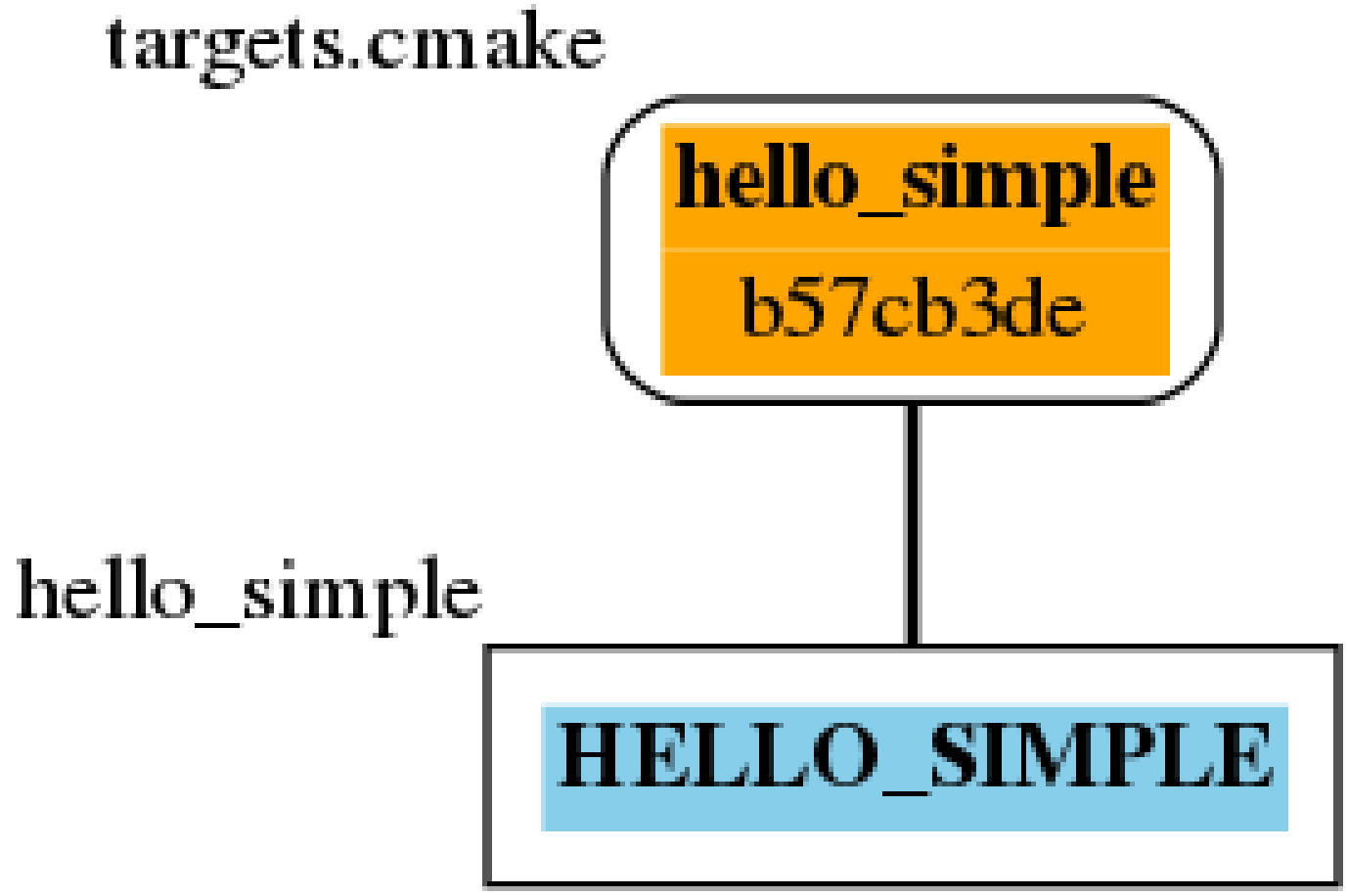
```
|  ├── cmake
|  │   └── beetroot (beetroot clone)
|  │       └── ...
|  ├── source.cpp
|  ├── targets.cmake
|  └── CMakeLists.txt
|
|
```

And finally we are set. Keep in mind, that Beetroot is designed to work best with middle and large size projects, so the amount of work to get the simplest C++ code compile is offset by the time we save when the project grows.

We compile it as usual:

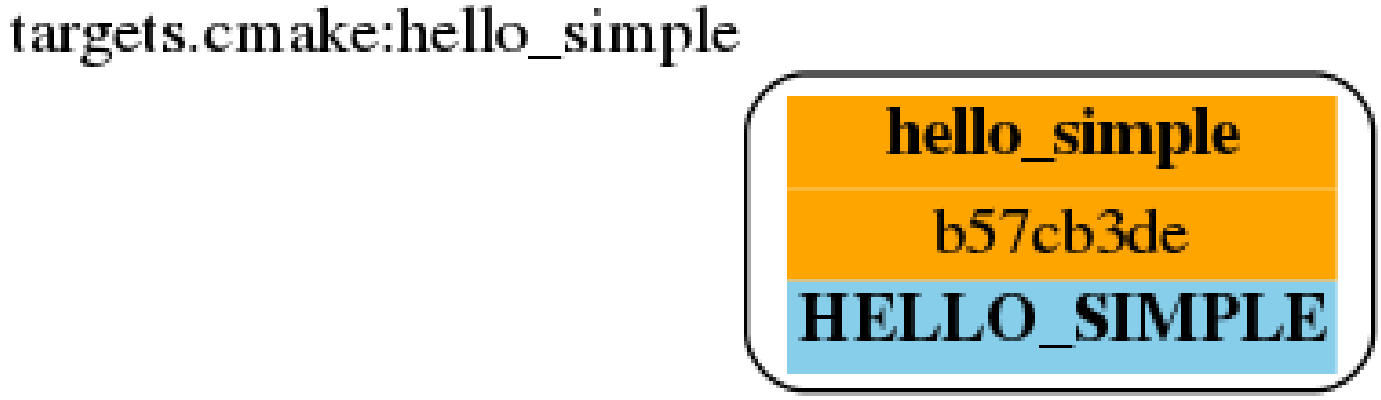
```
$ cd project_folder
$ mkdir build
$ cd build
$ cmake .. && make

    DECLARING  DEPENDENCIES  AND  DECIDING  WHETHER  TO  USE  SUPERBUILD

No languages in project bootstrapped_hello_simple
-- Discovering dependencies for HELLO_SIMPLE (HELLO_SIMPLE_
↪f9fc6118c955867490b6f80bce90dc5b)...



    DEFINING  TARGETS  IN  PROJECT BUILD
    TESTS  disabled

-- The CXX compiler identification is GNU 7.3.0
-- Check for working CXX compiler: /home/adam/spack/opt/spack/linux-ubuntu16.04-x86_
↪64/gcc-8.1.0/gcc-7.3.0-zclb4ttmy53mjkahiocmsqozhu6veriz/bin/g++
-- Check for working CXX compiler: /home/adam/spack/opt/spack/linux-ubuntu16.04-x86_
↪64/gcc-8.1.0/gcc-7.3.0-zclb4ttmy53mjkahiocmsqozhu6veriz/bin/g++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/adam/beetroot-examples/hello_simple/build

Scanning dependencies of target bootstrapped_hello_simple
[ 50%] Building CXX object CMakeFiles/bootstrapped_hello_simple.dir/source.cpp.o
[100%] Linking CXX executable bootstrapped_hello_simple
[100%] Built target bootstrapped_hello_simple
$ ls
hello_simple  CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile
$ ./hello_simple
Hello World!
```

Beetroot has built-in dependency graph generator in the dot language. This simples project can be visualized as



There are minimum two types of object directly involved when building even the simplest of projects: *INSTANCE* and *FEATUREBASE*. If the user code actually produce a CMake target, then *FEATUREBASE* is a proxy class to it (with one-to-one relationship if user code define a new target), whereas *INSTANCE* also encapsulates the linking information - how should the dependency influence the callee.

Because it is very common case that there is one-to-one relationship between the *FEATUREBASE* and the *INSTANCE* - i.e. between the target and the place it is defined (as always the case in plaine CMake), the former diagram is simplified to this:



5.3 The Hello World with parameter (02_parameter_hello)

Now let's start complicating things. You may have noticed, that we have a macro parameter WHO in our C++ file, that can be used to change the program's output. Let's do just that. After all, handling target parameters is one of the strongest sides of Beetroot. Let's modify our targets.cmake and insert definition of the parameter, which we will also call WHO:

```
set(ENUM_TEMPLATES HELLO_SIMPLE)

set(BUILD_PARAMETERS
   WHO SCALAR STRING "Beetroot"
)

function(generate_targets TEMPLATE_NAME)
   add_executable(${TARGET_NAME} "${CMAKE_CURRENT_SOURCE_DIR}/source.cpp")
   target_compile_definitions(${TARGET_NAME} PRIVATE "WHO=${WHO}")
endfunction()
```

The name of the parameter does not need to match the name of the preprocessor macro. The formal syntax is this: BUILD_PARAMETERS is an array organized into 4-element tuples.

1. First element of the tuple is the name of the parameter, then
2. container type. There are three container types: OPTIONAL, SCALAR and VECTOR, and they correspond to the CMake options, scalars and lists.
3. Element type. At the moment the are 5 possible types: BOOL, INTEGER, PATH, STRING and CHOICE(<colon-separated list of possible values>).
4. Default value.

In the function body we need to tie the parameter with the target, and we do that in the usual CMake way, by using target_compile_definitions(). All target parameters are always implicitely available in the function generate_targets, so we can simply use them.

If we compile the program and run we get:

```
$./hello_simple
Hello Beetroot!
```

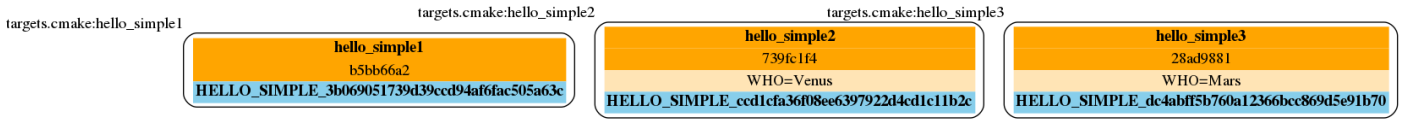
Let's say, that this file is our unit test and we need to compile three of them, one for the default string, and the other for a special string "Mars" and "Venus". It is easy with Beetroot, and by doing this we will demonstrate two ways of passing variables to targets. Let's re-write the CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.13)
include(../cmake/beetroot/cmake/beetroot_bootstrap)

project(hello_simple)

build_target(HELLO_SIMPLE)
set(WHO "Venus")
build_target(HELLO_SIMPLE)
build_target(HELLO_SIMPLE WHO Mars)

finalize()
```

After we build, we should get three executables: hello_simple1, hello_simple2 and hello_simple3.:

```
$./hello_simple1
Hello Beetroot!
$./hello_simple2
Hello Venus!
$./hello_simple3
Hello Mars!
```

The `targets.cmake` defines a target _template_, that can be used to define as many targets, as there are unique combinations of target parameters. That is why the `generate_targets()` function requires user to use `${TARGET_NAME}` instead of hard-coded name, that is usual in standard CMake practice. The function will be called exactly once for each distinct `${TARGET_NAME}` that Beetroot found is required to satisfy the parameters.



5.4 Targets composed from components (`03_subprojects_basics`)

Here you will learn how to combine targets together and use more realistic folder structure.

Suppose we have a program, that requires a function `get_string` from a library to run. The *hello_with_lib.cpp*:

```
#include <iostream>
#include "libhello.h"

#ifndef LIBPAR
#define LIBPAR 0
#endif

int main()
{
  int libpar = LIBPAR;

  std::cout << "Hello "<< get_string()<<"!"<< std::endl;
  return 0;
}
```

To compile it, we need a *libhello.h* that provides the `get_string()`:

```
#include<string>
std::string get_string();
```

The library's implementation is in the file `libhello.cpp`:

```
#include "libhello.h"
#define STRINGIFY2(X) #X
#define STRINGIFY(X) STRINGIFY2(X)

#ifndef WHO
#define WHO World
#endif

std::string get_string() {
        return(STRINGIFY(WHO));
}
```

The library depends on one macro: `WHO` that influences the text returned by the function.

We would like to have the `hello_with_lib.cpp` compiled and linked with the `libhello`. Although there is nothing wrong with putting the additional CMake commands in the old `targets.cmake` file, it is better to modularize our design and create two separate targets, so it will be easy to re-use the `libhello` by simply importing it.

Now is a time notice that the Beetroot by default does not care about the location of the target definitions. Instead it scans recursively all the superproject files in search for files `targets.cmake` and subfolder structure `cmake/targets/*.cmake`. Then it loads each fond file and learns the name of the targets/templates exposed there to build a mapping target/template name -> path of the target definition file, so user does not need to care about the paths anymore. On the other hand it requires that each each target/template name is unique across the whole superproject.

Let's create the following directory structure:

```
| superproject
| ├── cmake
| │   ├── beetroot (beetroot clone)
| │   │   └── ...
| │   └── root.cmake
| ├── hello_with_lib
| │   ├── hello_with_lib.cpp
| │   ├── CMakeLists.txt
| │   └── targets.cmake
| ├── libhello
| │   ├── include
| │   │   └── libhello.h
| │   ├── source
| │   │   └── libhello.cpp
| │   └── targets.cmake
| └── CMakeLists.txt
|
|
```

This is the definition of the `libhello/targets.cmake`:

```
set(ENUM_TEMPLATES LIBHELLO)

set(BUILD_PARAMETERS
   WHO       SCALAR  STRING  "Jupiter"
)

function(generate_targets TEMPLATE_NAME)
   add_library(${TARGET_NAME} "${CMAKE_CURRENT_SOURCE_DIR}/source/libhello.cpp")
   target_source(${TARGET_NAME} PRIVATE "${CMAKE_CURRENT_SOURCE_DIR}/include/libhello.
↪h") #For better IDE integration

   target_include_directories(${TARGET_NAME} PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/
↪include)
   target_compile_definitions(${TARGET_NAME} PRIVATE "WHO=${WHO}")
endfunction()
```

Nothing new, except we use `add_library` instead of `add_executable`. Adding `libhello.h` to sources is not strictly necessary, but is a good CMake practice, that helps various IDE generators generate better projects.

This is the definition of the `hello_with_lib/targets.cmake`:

```
set(ENUM_TEMPLATES HELLO_WITH_LIB)

```

(continues on next page)

(continued from previous page)

```
function(declare_dependencies TEMPLATE_NAME)
   build_target(LIBHELLO WHO "Saturn")
endfunction()

function(generate_targets TEMPLATE_NAME)
   add_executable(${TARGET_NAME} "${CMAKE_CURRENT_SOURCE_DIR}/hello_with_lib.cpp")
endfunction()
```

The new element, the `declare_dependencies()` function, is used to declare dependencies. It is a function, so user can build complex logic that turns certain dependencies on and off depending on the Target Parameters and Features. To declare a certain target/template a dependency we call a function `build_target(<TEMPLATE_OR_TARGET_NAME> [<PARAMETERS>...])`. The API and behaviour is exactly the same, as in `CMakeLists.txt`.

In `hello_with_lib/CMakeLists.txt` all we need is

cmake_minimum_required(VERSION 3.13) include(../cmake/beetroot/cmake/beetroot.cmake)

project(hello_simple)

build_target(HELLO_WITH_LIB)

finalize()

The location of the `CMakeLists.txt` is irrelevant in the Beetroot. You can as easily compile everything from within the root of the project if the root `CMakeLists.txt` is:

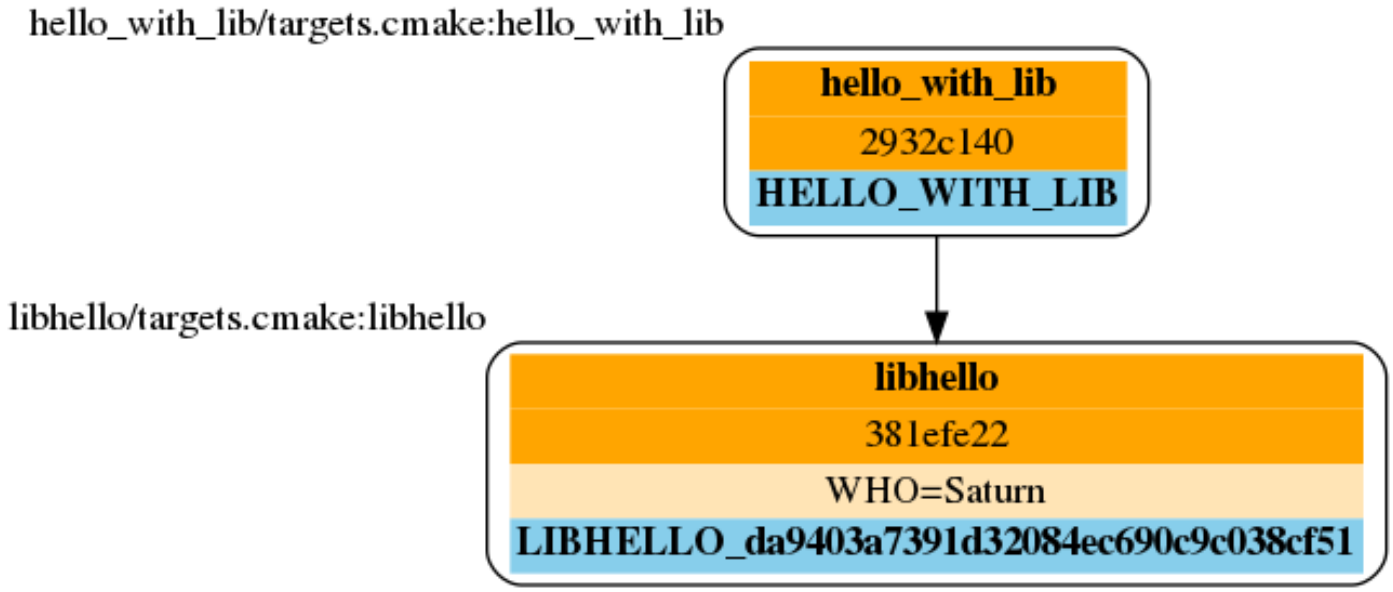
cmake_minimum_required(VERSION 3.13) include(cmake/beetroot/cmake/beetroot.cmake)

project(hello_simple)

build_target(HELLO_WITH_LIB)

finalize()

All we did aws a change to the directory of the beetroot library in the second line.



5.5 Forwarding parameters from dependencies (04_subproject_pars)

In the real life you will often find yourself putting many parametrized customizations to the components that play the role of the libraries in your project. Many of those parameters you would want to expose as customizations in the target executable - sort of forwarding those parameters from dependency to the dependee. Without an extra support for this common pattern, you would need to define again all the forwarded parameters in the body of dependee, and be carefull to match the type and container class to avoid configure errors.

To address this specific problem there are three functions: * `include_build_parameters_of()` to forward parameters, * `include_link_parameters_of()` to forward link parameters, and * `include_build_features_of()` to forward features (we will talk about them later).

Finally there is a universal function `include_parameters()` that incorporates functionality of all those three functions in one place.

The function call must be placed in the body of the `targets.cmake`, outside of the body of any function defined there, just along the place where you would normally define parameters.

The syntax is `include_parameters( <TEMPLATE_NAME> BUILD_PARAMETERS|LINK_PARAMETERS|BUILD_FEATURE`
`[NONRECURSIVE] [SOURCE BUILD_PARAMETERS|LINK_PARAMETERS|BUILD_FEATURES]`
`[ALL_EXCEPT <list of parameters>] [INCLUDE_ONLY <list of parameters>])`

The function imports the parameters from the specified template and acts as if you would copy-pasted them manually reducing code deduplication and ensuring consistency.

For better consistency user can choose whether to pick the names of imported parameter himself or to import all except the blacklisted names.

In the latter case, functions are capable of mass-importing all parameters (with exception of those in `ALL_EXCEPT`) from the single template. Since that template itself can use these functions to forward parameters from its dependencies, the amount of parameters can potentially get massive. In order to better control this situation, they offer `NONRECURSIVE` flag, that prevents it from importing the forwarded parameters.

The example `04_subproject_pars` is exactly the same with the exception of adding

include_build_parameters_of(LIBHELLO

INCLUDE_ONLY WHO

)

to the `hello_with_lib/targets.cmake`, so it reads like this:

set(ENUM_TEMPLATES HELLO_WITH_LIB)

include_build_parameters_of(LIBHELLO

INCLUDE_ONLY WHO

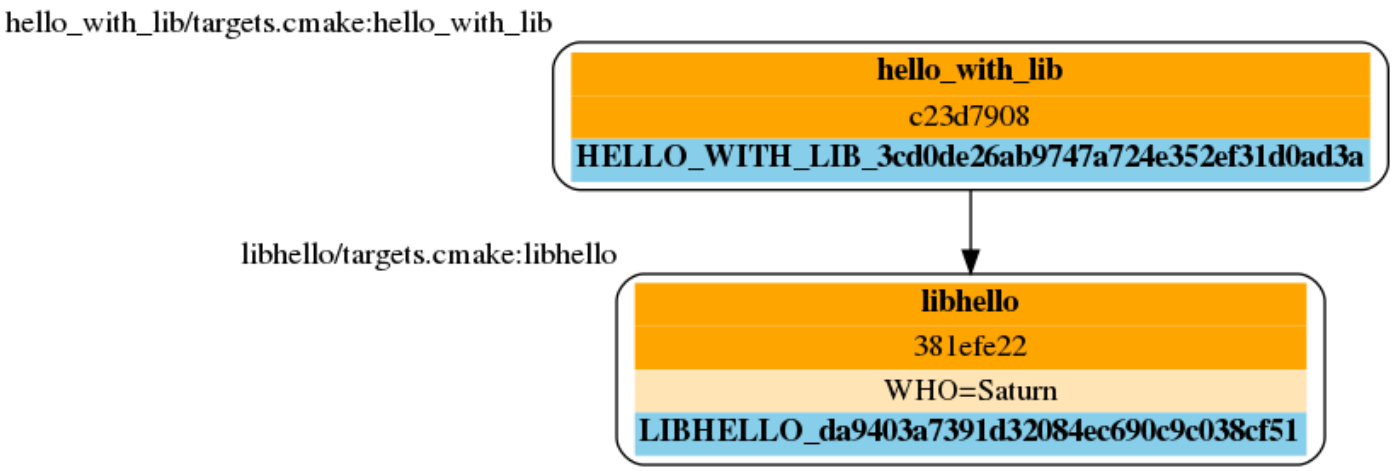
) #Implicitly imports (forwards) only WHO.

function(declare_dependencies TEMPLATE_NAME) build_target(LIBHELLO WHO "Saturn")

endfunction()

function(generate_targets TEMPLATE_NAME) add_executable(${TARGET_NAME} "${CMAKE_CURRENT_SOURCE_DIR}/hello_with_lib.cpp") target_compile_definitions(${TARGET_NAME} PRIVATE "WHO=${WHO}") # ${WHO} is now available and can be used as a compile option

endfunction()



5.6 Code generators, known generated files (05_codegen_known)

From the Beetroot point of view, code generators are targets that add a source file to the parent (dependee) taret. Actions that happen after the target (the source code) was build are called _link time **actions_**, and they usually apply special attributes to the dependee. In this particular case we need to add the generated file as a source to the dependee using `target_sources()` CMake function[1] .

Let us implement a simple code generator that uses `configure_file()`. If this example may look too simple to be realistic, remember that the Beetroot does not replace common CMake idioms regarding low-level file handling. The example can as well use `add_custom_command()` instead.

Imagine we have the following stub code generator, written in Python, in `lib_gen/generator.py`:

import argparse

parser = argparse.ArgumentParser() parser.add_argument("-o", "–output", help="output file") parser.add_argument("-s", "–string", help="debug string") args=parser.parse_args()

print(""" const char* getString() {

return "my generated string: "" + args.string + """;

}""", file=open(args.output+".cpp", "w"))

print("const char* getString();", file=open(args.output+".h", "w"))

All it does is generating two files: `<output>.cpp` and `<output>.h` in the current directory. Let's write the CMake code that drives the code generation:

Contents of the `lib_gen/targets.cmake`:

set(ENUM_TEMPLATES CODEGEN1_GEN)

set(BUILD_PARAMETERS SOURCE_OUT SCALAR PATH "generated_source"

)

set(FILE_OPTIONS NO_TARGETS)

[1] Note, that this approach assumes the names of the generated files are not known beforehand. In such case you would need to call the code generator in `apply_dependency_to_target()` during the configure phase via `execute_process()` and gather the resulted files by the means of file globbing or `OUTPUT_VARIABLE` option.

function(apply_dependency_to_target DEPENDEE_TARGET_NAME OUR_TARGET_NAME)
get_target_property(DEPENDEE_BINARY_DIR ${DEPENDEE_TARGET_NAME} BINARY_DIR) #Not used here, but kept as a code reference: get_target_property(DEPENDEE_SOURCE_DIR ${DEPENDEE_TARGET_NAME} SOURCE_DIR)

find_package(Python3 COMPONENTS Interpreter REQUIRED) set(Python_EXECUTABLE "${Python3_EXECUTABLE}")

add_custom_command(OUTPUT "${DEPENDEE_BINARY_DIR}/${SOURCE_OUT}.cpp" COMMAND ${Python_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/generator.py -o ${SOURCE_OUT} -s "Hello!" WORKING_DIRECTORY ${DEPENDEE_BINARY_DIR} COMMENT "Generating file ${SOURCE_OUT}..." VERBATIM

) target_include_directories(${DEPENDEE_TARGET_NAME} PRIVATE ${DEPENDEE_BINARY_DIR}) target_sources(${DEPENDEE_TARGET_NAME} ${KEYWORD} ${SOURCE_OUT}.cpp)

endfunction()

We call the template `CODEGEN1_GEN`, export the filename as a parameter. Then we instruct the Beetroot that this template will not generate targets with setting the flag ‘‘NO_TARGETS‘‘[#f3]_ .

The client C++ code that uses the generated library, nothing special here (`codegen_client/codegen1_client.cpp`):

#include <iostream> #include "my_generated_source.h"

int main() { std::cout << "Hello "<< getString()<<"!"<< std::endl; return 0; }

The best part is how we build the client (the executable): we just add a single dependency line! (`codegen_client/targets.cmake`):

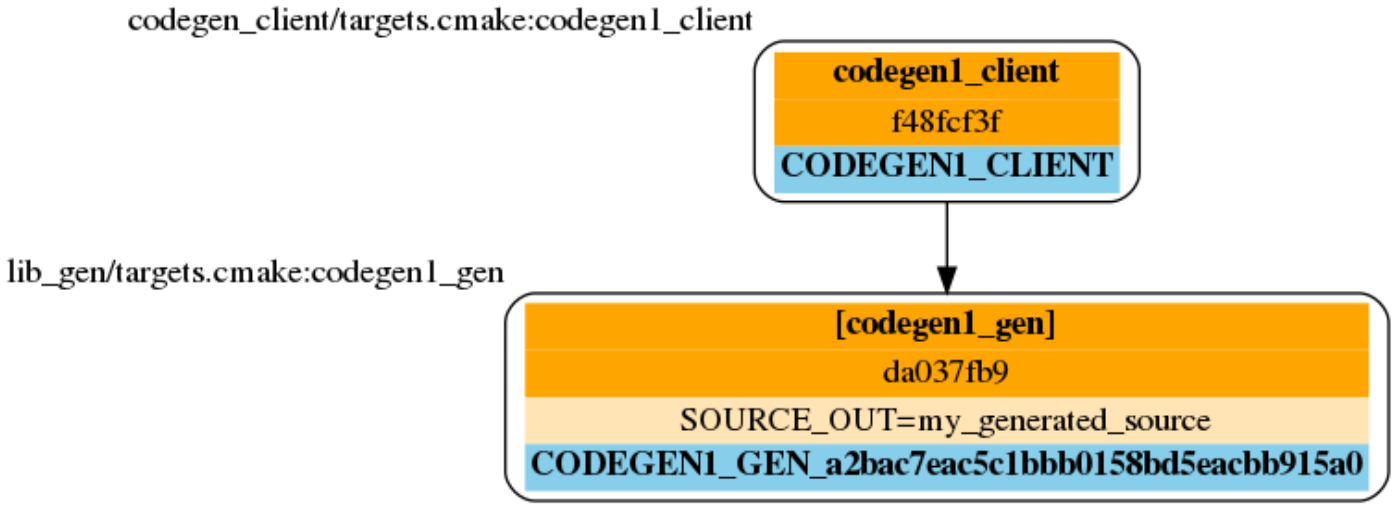
set(ENUM_TEMPLATES CODEGEN1_CLIENT)

function(declare_dependencies TEMPLATE_NAME) build_target(CODEGEN1_GEN SOURCE_OUT my_generated_source)

endfunction()

function(generate_targets TARGET_NAME TEMPLATE_NAME)
add_executable(${TARGET_NAME} "${CMAKE_CURRENT_SOURCE_DIR}/codegen1_client.cpp")

endfunction()



5.7 External projects and the superbuild idiom

External projects are CMake projects that need a separate CMake run to be built. If they are written properly, the act of installing them (after build) would result in the *<project name>Config.cmake* files describing the way the library shuld be linked to our project. Those files are then be used when we import the library using CMake's command *find_packages()* or one of its specialized forms, like *find_boost()*.

The library can either be already installed by the OS packaging system, or we need to provide the source code, build and install it ourselves. In the latter case, it is customary to include that library as our dependency in the form of a git submodule (if both the library and our project is using git) or download script executed during build.

The problem is that the *<project name>Config.cmake* files of the dependency appear only after it was build, and installed which is after the CMake finished running our script and there is no way for them to influence the configuration of our project, resuling in the build failure on the first build (The subsequent builds will be fine). The most robust way to solve this problem is to execute the *superbuild* idiom.

Superbuild idiom means putting our project as the last external dependency of the "super project", which depends on all the external dependencies and building that project instead of the original. When user calls *cmake <our_project>* CMake first makes sure all the external projects are built and installed, and then at the end calls the CMake again to process our own project - this time we can be sure that all the dependencies are built and update.

Beetroot automatically switches to the superbuild idiom automatically everytime we define any external targets.

The Beetroot treats the target as external if the template file sets non-empty contents to the *'DEFINE_EXTERNAL_PROJECT'* variable. There are

5.8 Non-compiled components (e.g. header libraries)

In the CMake, there are two ways of implementing the header-only libraries: the old, deprecated method that involves using `target_include_directories()` on the dependee target (target that is needs to use the library) or the current best-practise method that involves defining the target with only "interface" properties. Let's start with the modern way first:

Imagine, that the `hello_with_lib` is also responsible for setting a macro variable in the client's code. Let's predend that this variable modifies behavior of the header-only part of this library. Consequently will not change the library code. We only need to make sure, that clients linking to our library receive a new preprocessor macro:

```
set(ENUM_TEMPLATES LIBHELLO)

set(BUILD_PARAMETERS
   WHO       SCALAR  STRING  "Jupiter"
)

set(LINK_PARAMETERS
   LIBPAR    INTEGER
)

function(apply_dependency_to_target DEPENDEE_TARGET_NAME OUR_TARGET_NAME)
   target_compile_definitions(${DEPENDEE_TARGET_NAME} PRIVATE "LIBPAR=${LIBPAR}")
endfunction()
```

5.9 Subcomponents that influence the parent

When we require the subcomponent in function `declare_dependencies` we have a total control of all the information (i.e. parameters) the component receive. But what if we want the component to influence the build process of the parent project as well? Imagine this simple logging example - we want to include logging support to our application by

TODO: Find a good case (better than target_compile_definitions with log or target_include_directories for header libraries)

We have seen in **'The Hello World with parameter'_** that for each unique variation of the parameters of the compoment Beetroot defines a distinct target. That is a welcome feature if the parameter modifies the compilation process of the component, but what if we need to parametrize *linking*?

#include <iostream> #include <boost/log/trivial.hpp>

int main(int, char*[]) {

BOOST_LOG_TRIVIAL(info) << "This is an informational severity message"; std::cin.get();
return 0;

}

CHAPTER 6

List of most important and complex Beetroot functionalities

6.1 Advanced and convenient handling of targets' parameters

Parameters to each target are handled in three ways:

1. Default values are specified in the `targets.cmake`, when declaring the parameters.
2. Default values are overriden by existing variables that have the same name.
3. Existing variables are overriden by named arguments given to the `get_target` and `build_target` functions.

Furthermore, system allows for default values to include references to the any variables (including defined parameters themselves) and any other CMake string processing commands. For instance, this is a perfectly valid definition:

```
get_filename_component(TMP "${ARCH_PREFIX}" NAME)

set(DEFINE_MODIFIERS
        ARCH    SCALAR  CHOICE(GPU;CPU) CPU
        ARCH_PREFIX     SCALAR  STRING "${SUPERBUILD_ROOT}/external-${ARCH}"
        ARCH_RELDIR     SCALAR  STRING  "${TMP}"
)
```

The intention is to let the user specify only the `ARCH` parameter, and the `ARCH_PREFIX` and `ARCH_RELDIR` gets calculated from it.

6.2 Distinction between target parameters and link parameters

Generally all declared template parameters change the identity of the target. Occasionally though, when the target is used as a dependency, some settings should not change its identity target and trigger a separate build. Those settings include preprocessor macros that are applied for header-only libraries, or are consumed only by the library headers. In this case we can define those parameters as `LINK_PARAMETERS`. User cannot use them inside the `generate_target()` (unless the `GENERATE_TARGETS_INCLUDE_LINKPARS` flag is set,

and even then they should not be used in target definitions, but for non-target task, such as defining tests) and `declare_dependencies()` functions, but they are available in `apply_to_target()`.

6.3 Target definition can make multiple physical targets, each one with different set of parameters

When you require a target `FOO` and specify target parameters, the target with this specific set of parameters will be defined and build. If you require the same target `FOO` with different set of parameters, by default Beetroot will define *another* instance of the same target, built using the other set of parameters. The targets' CMake names (and file names) will have a numeral suffix to distinguish them apart. This implies that in general you cannot know the name of the target, until you query for it (or you declared that there can be only one instance of it).

6.4 Special handling for target parameters that describe features

Sometimes a parameter defines *additional feature* of the target, that is turned on/advanced independently of the rest of the target functionality. Parameter is a feature, when the dependee who does not require it can still use the target that has it. Features can be flags like `"FORTRAN_SUPPORT"` (if Fortran support does not affect the rest) or a `"VERSION"` (if target development makes sure the code backward compatible). Support for Features makes re-using the same target in more than one place much better, especially if the target is heavy to build (e.g. big external dependency) and have many different features that can be turned on/off that are consumed by different parts of our project.

6.5 Target definition file can describe more than one template/target

There is one-to-many relationship between target definition files and target definition. User can put multiple target/template names in the `targets.cmake`. First parameter of `generate_target()` and `declare_dependencies()` is the name of the target/template, so the user code that handles target definition and dependencies can handle each target/template in different way. The feature was introduced to faciliate defining multiple similar targets in one file. It is used typically in external targets, where e.g. Boost defines multiple targets.

6.6 Support for CMake code that do not produce targets

Sometimes we need CMake to do something that does not necessary ends up being a new target, but rather modify existing target. This situation include dedicated code for definition of unit tests (of course unit tests can also be defined inside `generate_target()` that defines the executable), code that applies additional preprocesor definitions or old external targets that not use imported targets CMake mechanism.

6.7 Comprehensive error checking

We spent a great deal of effort to catch as many usage errors as possible. In particular:

- We make sure that template modifiers and parameters share the same name space and report all violations.
- System makes sure, that the actual template(s) you want to build actually produce targets.
- System makes sure that dependencies that do not generate targets are not dependent on templates that also do not produce targets.

- System detects lack of both `generate_target()` and `apply_to_target()` for internal projects.
- System detects and forbids variables that start with double underscore, as this can interfere with the Beetroot's inner working.
- System checks the types and possible values of all arguments against a declaration in `targets.cmake`.

6.8 External dependencies are external to the Beetroot system

The system faciliates calling them, manages their external dependencies, build and install location, but ultimately it calls them as if they were a simple external CMake projects, builds them and installs them in the totally normal way.

6.9 Automatic superbuild by default

If there is at least one external target defined, all external dependencies are built in the superbuild phase, using the dependencies derrived from the template definitions. Only then the project build is called (implemented as a "external" dependency of all actual external dependencies), so all external dependencies are always already built and installed when you use them. If there are no external dependencies, CMake will build project directly.

CHAPTER 7

List of the most visible features of the Beetroot

7.1 Targets are defined inside the function `generate_target()` in target definition file

In CMake targets (in contrast to variables) are internally represented by the object which lives in a global namespace, even if defined inside the function. Putting target definitions inside a function prevents leaking of temporary variables and pollution of the variable namespace.

7.2 Dependencies of targets are set inside the function `declare_dependencies()` in target definition file

Target dependencies are handled by function rather than data structure, which allows for maximum flexibility (dependencies can depend in complicated way on the target parameters/features). Because Beetroot structure needs dependencies to be resolved *before* target definition (and possibly be called multiple times on the same target), the only place to put them is in a dedicated user-supplied function. Code inside this function should be omnipotent, because it can be executed multiple times in a single run. The code will be executed only during the target declaration phase.

7.3 By default the code you write (targets.cmake) does not depend on your target name

Unless instructed otherwise, the system dictates the name you give to each target. This way targets' names are not fixed, and it is possible to have multiple instances of them. This fact is used to let the target definition files (`targets.cmake`) define whole family of targets parametrized by the target parameters and features. The beetroot guarantees, that for each distinct set of target parameter there will be a separate target defined and built.

7.4 There is only one type of user-supplied input file that defines the targets

All code that define targets and their dependencies can be placed inside so-called target definition files. These files can be put anywhere in the project and must be named `targets.cmake`, or be placed in the special subfolder `cmake/targets` and have an extension `.cmake`. The latter files usually define external dependencies. The only thing that is influenced by the location of the file, is the value of the `${CMAKE_CURRENT_SOURCE_DIR}` CMake variable available in `generate_target()` user function.

The user file works by defining any of the following cmake variables: `ENUM_TEMPLATES`, `ENUM_TARGETS`, `BUILD_PARAMETERS`, `LINK_PARAMETERS`, `BUILD_FEATURES` `FILE_OPTIONS` and `DEFINE_EXTERNAL_PROJECT` and by defining any of the following functions: `generate_targets()`, `declare_dependencies()` and `apply_dependency_to_target()`. Of course, not all combinations of those definitions are legal and any violation of the legality of the definitions is cought and meaningfully reported to the user.

The other file a user needs to supply is a `CMakeLists.txt`. This file serves as a point of entry. This file should consist of a standard boilerplate code, calls to the `build_target()` and finally a call to `finalize()`. Standard CMake commands should not be used to define targets. The only purpose of this file is to specify what targets with what parameters must be build by calling a Beetroot function `build_target()` or `get_target()` and letting it do the work.

7.5 The role of the CMakeLists.txt is hugely downplayed

There is no need to use `add_subdirectory()`, because Beetroot knows where to look for every managed target. That's why the only `CMakeLists.txt` that is needed is the one you manually call with `cmake ...` (Besides `CMakeLists.txt` inside CMake external dependencies of your project)

7.6 The location of the CMakeLists.txt is no longer relevant

As long as the `CMakeLists.txt` is somewhere inside the root project and it adheres to the Beetroots' mandatory boilerplate code, its location is irrelevant. All components are searched for by name, not by folder, and the system requires them to be written in a way that all paths are absolute (it is achieved by simply prefixing filenames with the `${CMAKE_CURRENT_SOURCE_DIR}`).

As long Beetroot is responsible for all targets in your code, you can simply copy a `CMakeLists.txt` from one subfolder of your project to another and they will build just fine there, resulting in exactly the same executable.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search