Haifa::C++ Meetup
28-Oct-2019

**Alexander Nezhinsky**
**Yan Vugenfirer**

# CMake for big projects

## from best practices to a framework

# About us

**Alexander Nezhinsky** has two decades of experience in storage and communication systems, in-depth knowledge of Linux system and kernel programming, build and continuous integration processes.

**Yan Vugenfirer** is a virtualization and cloud technologies veteran. Yan has strong skills in Windows and Linux internals, project management methodologies and DevOps ecosystems.

**BladeRunner Labs** is a consulting company specializing in state-of-the-art technologies and modern development methodologies. We leverage our broad experience in software development (including Open Source) for various domains and project scales.

# Presentation Outline

## Main Ideas

- Simplify build environments using a **templated CMake framework**:
  - Intuitive
  - Uniform
  - Recursive
- Run builds with **multiple Git repos**:
  - Automated updates
  - Module cross-references
  - Parallel builds: one-by-one or together

# Presentation Outline

## CMake ‘Tenses’

- CMake Present Simple `# Runs auto-magically`
- CMake Progressive `# Evolving`
- CMake Conditional `# Only if you do it right`
- CMake {Future} Perfect `# Will it work out?`
- CMake Imperative `# Do it right today!`

# CMake Present Simple

## Status

- Started as a simple Makefile **generator**
  - **Cross**-platform
  - Most details **Automated**
  - **Readable**
- Became a **build tool** of choice for many
  - **Easy to start** : works “out of the box”
  - Evolved into rich & extensible **meta-language**
  - Became *“kind of”* **object**-oriented
  - Supported by **IDEs**

# CMake Present Simple

## Runs auto-magically

```
minimum_required(VERSION 3.10)
project(simple-proj)
set(CMAKE_VERBOSE_MAKEFILE ON)
set(CMAKE_BUILD_TYPE Release)
set(CMAKE_CXX_FLAGS "-std=c++14 -Wall")
set(SIMPLE_SRC s1.cpp s2.cpp s.hpp)
add_executable(simple-app ${SIMPLE_SRC})
add_subdirectory(tests)
```

# CMake Progressive

## Evolving: Everything is a target

```
add_library(slib ${SLIB_SRC})
target_include_directories(slib
  PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}
  PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/api)
target_compile_definitions(slib
  _GNU_SOURCE)
target_compile_opions(slib -Wcast-align)
target_link_libraries(slib xlib ylib)
```

# CMake Progressive

## Packages: Bundle it for me

- Products of a module can be bundled in a **package**
- Entire targets:
  - files: binaries and headers
  - properties : symbols, options, dependencies etc.
- When a package is found, everything is available
- Packages export their **namespace** prefixes

# CMake Progressive Packages: Find it for me

- **public :** CMake packages created by many SDKs

```
find_package(Boost REQUIRED
    COMPONENTS thread system timer)
# Watch the namespace!
target_link_libraries(slib Boost::thread)
```

- **private :** packages can be created as installed artifacts (on top of the executables, libs etc)
- They behave the same as the official ones:

```
find_package( MyPackage REQUIRED )
target_link_libraries(slib MyPackage::my_lib)
```

# CMake Conditional

## Only if you do it right

- Multiple concepts evolve:
  - **Enjoy many options** - **if** you choose right
- Old methods linger:
  - **Consistency** - **if** you don’t mix styles
- Rich documentation:
  - **Comprehensible** - **if** you are not a newbie
- Complex projects:
  - **Scales well** – **if** you impose a structure

# CMake Imperative

## Do it right today!

- Create a new language subset
- Use functions/macros, to be included by all
- Handle all bloody details there
- Restrict all usage to the new “dialect”
- Support project modularization
- Support multiple Git repositories
- Plan for multiple teams and CI

# CMake Frameworks
## No need to RE-INVENT

- **We DO NOT RE-INVENT the CMake “Wheel”:**
  - the technology is mature
  - the methods are quite straightforward
  - the details are described elsewhere
- **But**...
  - the knowledge is scattered
  - project structure remains open-ended
  - some details are tedious
  - some syntax is obscure

# CMake Frameworks
## RE-FACTOR

**We DO RE-FACTOR using the CMake “Wheels”:**

- Build on experience and “good” practices
- Automate repetitive tasks and blocks
  - Identify
  - Isolate
  - Template
- Help define the structure
- Fill the gaps, provide the missing parts

# CMake Frameworks

## Requirements

- **Customizable**: fit the project
- Every-day **usage** : **easy**
- **Maintenance** : reasonably **simple**
- **Modularization**:
  - break the source tree into multiple subprojects, repositories etc.
  - Auto-update source using manifests
  - build all together or piece-by-piece
- **Uniform workflow**:
  - for different teams and automation

# Framework Usage

## Example: Library

```
# this package
define_package ( alpha ) …
# external packages
use_package ( ui ) …
# library alpha::abc_lib
define_lib ( abc_lib
    SOURCES abc_main.cpp abc_util.cpp
    PRIV_HEADERS abc_priv.hpp
    API_HEADERS abc_api.hpp API_DIR api
    LIBS alpha::xyz_lib ui::message_lib)
```

# Framework Usage

## Example: Application

```
define_package ( alpha ) …
use_package ( ui ) …
define_lib ( abc_lib ...
    LIBS alpha::xyz_lib ui::message_lib) ...
# application, note transitive dependencies
define_app ( abc_daemon
    SOURCES abc_daemon.cpp
    PRIV_HEADERS abc_daemon.hpp
    LIBS alpha::abc_lib)
```

# Framework Internals

## Packages Interaction

- Every **target** pertains to a **package**
- The package **namespace prefix** (*e.g.* `alpha::`) is used uniformly:
  - **Inside** the package a namespaced target resolved as an **alias** (*e.g.* `alpha::abc = abc`)
  - **Outside** the package it is a real **namespace**
- Each package is **installed** (products) and **exported** (properties)
- Then it can be found and imported

# Framework Usage

## Example: API library

```
# API is a bundle of headers which provides a
common interface, without producing a binary

define_package ( alpha ) …

# alpha::unicorn_api is treated like a library

define_api ( unicorn_api

    API_HEADERS uni.hpp corn.hpp

    API_DIR .) ...
```

# Framework Usage

## Example: API and Unit Test

```
define_package ( beta ) …

# unit_test an extension of executable

define_unit_test ( unicorn_utest

    SOURCES unicorn_test.cpp

    LIBS alpha::abc_lib alpha::unicorn_api)


# beta::unicorn_utest uses alpha::unicorn_api
```

# Live Example Project: demo_app

- Project is divided into modules:
  - **demo_app**: main functionality
  - **demo_infra**: infrastructure libraries
- **Multi-repository** (Git)
- **Multi-package** (CMake)

## # Let’s have a look...

# Framework Internals

## Example: Library - arguments

```
function(define_lib lib_name)
    set(args_single API_DIR SRC_DIR)
    set(args_multi SOURCES API_HEADERS LIBS)

    cmake_parse_arguments("LIB"
        "${args_single}" "${args_multi}" ${ARGN})
    …
    add_library(${PROJECT_NAME} ${LIB_SOURCES} …)
```

# Framework Internals

## Example: Library - definition

```
add_library(${PROJECT_NAME} ${LIB_SOURCES} …)

add_library(${NAMESPACE_NAME}${PROJECT_NAME}
    ALIAS ${PROJECT_NAME})

target_compile_definitions(${PROJECT_NAME} …)
target_include_directories(${PROJECT_NAME} …)
target_link_libraries(${PROJECT_NAME} …)
```

# Framework Internals

## Example: Library - install

```
install(TARGETS ${PROJECT_NAME}
    EXPORT ${EXPORT_NAME}
    ARCHIVE DESTINATION lib …)
install(EXPORT ${EXPORT_NAME}
    NAMESPACE ${NAMESPACE_NAME}
    DESTINATION ${PACKAGE_NAME}
    FILE ${EXPORT_FILE})
export(EXPORT ${EXPORT_NAME}
    NAMESPACE ${NAMESPACE_NAME})
```

# Multiple Git Repositories

## Package = Module = Repo

- Every **module:**
  - contains a manifest file: **git_pull.cfg**
  - may act as a **seed**
- **Seed** module:
  - Git clones/**pulls** other modules
  - according to its git_pull.cfg **rules**
- **git_pull.cfg rules**
  - based on **local** branch in seed
  - support regular expressions

# Multiple Git Repositories

## git_pull.cfg

```
# local-ref | url | remote-ref | dir
dev* | ${git_url} | dev | ${root}/demo_infra
qa* | ${git_url} | stable | ${root}/demo_infra
* | ${git_url} | master | ${root}/demo_infra
```

- **Local branch based rules**:
  - require branches/tags
  - for every dependency
  - according to the version being built

**# Let’s have a look...**

# Frameworks Overview

## Requirements Revisited

- All internals are **templated:**
  - uniform structure, single policy
  - any change is done in one place
- **Customizable**:
  - Not a product, but a framework base
  - Customized to fit **specific project needs**

# Frameworks Overview

## Requirements Revisited

- Every-day **usage** : **easy**
  - Target definitions contain only names & flags
- **Maintenance:** reasonably **simple**
  - Project wide definitions in central location

# Frameworks Overview

## Requirements Revisited

- **Modularization**: supported thru **packages**
  - source tree can be split: **module := package**
  - Build: all together or package-by-package
- **Uniform workflow**:
  - for all teams and automation
  - sources auto-updated, manifests from git
  - Foreign repos pulled by the local version:
    - CI builds triggered by push to a single repo
    - Allows features spanning multiple git repos

# Discussion

# code available at: github.com/

- bladerunnerlabs/**build-runner**
  - # branch: **demo_app**
- bladerunnerlabs/**build-demo-app**

# Thank You!