KAIZOU TOOLS DEMOS ?

A typical Linux project using CMake

03 Nov 2014 by David Corvoysier

When it comes to choosing a make system on Linux, you basically only have two options: autotools or CMake. I have always found Autotools a bit counter-intuitive, but was reluctant to make the effort to switch to CMake because I was worried the learning curve would be too steep for a task you don't have to perform that much often (I mean, you usually spend more time writing code than writing build rules).

A recent project of mine required writing a lot of new Linux packages, and I decided it was a good time to give CMake a try. This article is about how I have used it to build plain old Linux packages almost effortlessly.

Although CMake is fairly well documented, I personnally found the documentation (and especially the tutorial) a bit too CMake-oriented, forcing me to use cmake dedicated tools for tasks I had already tools for (tests and delivery for instance).

This is therefore my own tutorial to CMake, based on my primary requirement: just generate the makefiles using CMake, and use my own tools for everything else.

Project structure

The project structure is partly driven by the project design, but it would ususally contain at least two common sub-directories, along with several "module" sub-directories:

```
main
test
moduleA
moduleB
...
```

The `main` subdirectory contains the main project target, typically an executable.

The `test` directory contains one or more test executables.

The `moduleX` directories contain libraries to be used by either the tests or main executables.

At the root of the project, the main `CMakeLists.txt` should contain the common CMake directives that apply to all subdirectories.

First, the `CMakeLists.txt` would specify a minimum Cmake version, name your project and define a few common behaviours.

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

PROJECT(MyProject)

SET(CMAKE_INCLUDE_CURRENT_DIR ON)
```

Here, I only set one option that is of uttermost importance if you want to build out-of-tree AND generate some of your source files automatically (you most certainly do actually if you are using ANY modern framework like Qt). What it does is that it adds the `${CMAKE_CURRENT_SOURCE_DIR}` (this one you don't care that much) and `${CMAKE_CURRENT_BINARY_DIR}` to the include path, allowing generated include files to be found by the compiler.

Finally, the `CMakeLists.txt` would list all subdirectories to be included in the project:

```
ADD_SUBDIRECTORY(main)
ADD_SUBDIRECTORY(test)
ADD_SUBDIRECTORY(moduleA)
ADD_SUBDIRECTORY(moduleB)
...
```

Configuring Modules

As explained in the previous paragraph, each subdirectory would contain at least either one executable or one library defined in a dedicated `CMakeLists.txt` file.

Executables are declared using the `ADD_EXECUTABLE` command:

```
ADD_EXECUTABLE(myapp
  ${MY_SRCS}
)
```

Libraries are declared using the `ADD_LIBRARY` command:

```
ADD_LIBRARY(mylib STATIC
   ${MY_SRCS}
)
```

Source files are specified either explicitly or using a wildcard:

```
SET(MY_SRC
   fileA.cpp
   fileB.cpp
   ...
)
```

or

```
file(GLOB MY_SRC
   "*.h"
   "*.cpp"
)
```

> Note that using a wildcard, you need to rerun CMake if you add more files to a module

Solving dependencies between modules

Link dependencies

Link dependencies between modules are solved using the `TARGET_LINK_LIBRARIES` command.

CMake maintains throughout the whole project a named object for each target created by a command such as `ADD_EXECUTABLE()` or `ADD_LIBRARY()`.

This target name can be passed to the `TARGET_LINK_LIBRARIES` command to tell CMake that an object A depends on on object B.

Example:

Given a library defined in a specific subdirectory

```
ADD_LIBRARY(mylib STATIC
  ${MY_LIBSRCS}
)
```

One can specify a dependency from an application to that library

```
ADD_EXECUTABLE(myapp
  ${MY_APPSRCS}
)

TARGET_LINK_LIBRARIES(myapp
  mylib
)
```

Include dependencies

Include dependencies are automatically solved for dependent libraries declared in the `TARGET_LINK_LIBRARIES` command if the corresponding libraries have properly declared their include directories using the `TARGET_INCLUDE_DIRECTORIES` command.

Example:

Given a library defined in a specific subdirectory

```
ADD_LIBRARY(mylib STATIC
  ${MY_LIBSRCS}
)
```

Specifying

```
TARGET_INCLUDE_DIRECTORIES(mylib
  /path/to/includes
)
```

Allows a dependent app to be aware of the mylib include path just when adding the lib to the `TARGET_LINK_LIBRARIES`

```
ADD_EXECUTABLE(myapp
  ${MY_APPSRCS}
)
```

```
TARGET_LINK_LIBRARIES(myapp
   mylib
)
```

Additional include dependencies can be solved explicitly using the `INCLUDE_DIRECTORIES` command, but most of the time, you won't need it unless you have nested sub-directories that don't have a `CMakeLists.txt` of their own (as a matter of fact, needing to add an explicit `INCLUDE_DIRECTORIES` may be a good hint that something is wrong with your other directives).

Resolving Dependencies towards external packages

Packages known by CMake

CMake provides a set of tools to register and retrieve information about packages stored in a CMake package registry.

CMake packages dependencies are solved easily by specifying them using the built-in CMake `FIND_PACKAGE` commands.

```
FIND_PACKAGE(Qt5Core)
```

This command will create a CMake target Qt5::Core that can be referenced in `TARGET_LINK_LIBRARIES` commands.

```
ADD_LIBRARY(mylib STATIC
   ${MY_LIBSRCS}
)

TARGET_LINK_LIBRARIES(mylib
   Qt5::Core
)
```

> Note: The `FIND_PACKAGE` command will also export several related variables.

Just like when referencing an internal module, the paths to the specific includes of libraries found using `FIND_PACKAGE` are automatically added to the include search path. There is therefore no need to add them explicitly using an `INCLUDE_DIRECTORIES` directive.

Other packages: pkg-config

For package whose definition is not maintained in CMake (ie there is no FIND_PACKAGE macro written for them), you may rely on the generic pkg-config tool instead.

pkg-config is a helper tool used when compiling applications and libraries. It helps you insert the correct compiler options on the command line so an application can use `gcc -o test test.c pkg-config --libs --cflags glib-2.0` for instance, rather than hard-coding values on where to find glib (or other libraries). It is language-agnostic, so it can be used for defining the location of documentation tools, for instance.

pkg-config compatible packages declare their include path, compiler options and linking flags in dedicated `.pc` files installed on the system.

Here is for instance the `glib-2.0` pkg-configfile:

```
prefix=/usr
exec_prefix=${prefix}
libdir=${prefix}/lib/x86_64-linux-gnu
includedir=${prefix}/include

glib_genmarshal=glib-genmarshal
gobject_query=gobject-query
glib_mkenums=glib-mkenums

Name: GLib
Description: C Utility Library
Version: 2.36.0
Requires.private: libpcre
Libs: -L${libdir} -lglib-2.0
Libs.private: -pthread  -lpcre
Cflags: -I${includedir}/glib-2.0 -I${libdir}/glib-2.0/include
```

Before using pkg-config, you need to make sure the tool is available by inserting the following line in your `CMakeLists.txt`:

```
FIND_PACKAGE(PkgConfig)
```

Then, insert the following PKG_CHECK_MODULES command in your `CMakeLists.txt` file to tell CMake to resolve pkg-config dependencies for a specific package:

```
PKG_CHECK_MODULES(GLIB2 REQUIRED glib-2.0>=2.36.0)
```

The command will export several variables, including the `XXX_LIBRARIES` command that can be used in `TARGET_LINK_LIBRARIES` commands.

```
ADD_LIBRARY(mylib STATIC
   ${MY_LIBSRCS}
)

TARGET_LINK_LIBRARIES(mylib
   GLIB2_LIBRARIES
)
```

Unfortunately, I was unable to get the include paths of libraries found through pkg-config to be added automatically to the include source paths just like it it when using the standard `FIND_PACKAGE` function, so I needed to add them explicitly:

```
INCLUDE_DIRECTORIES(
   GLIB2_INCLUDE_DIRS
)
```

Exporting dependencies towards external packages

Although CMake supports its own mechanism to export dependencies, it is recommended to take advantage of the more generic pkg-config files.

CMake doesn't provide any specific mechanism to generate `.pc` files.

However, one can take advantage of CMake variables substitution to generate a specific pkg-config file from a predefined template.

```
CONFIGURE_FILE(
  "${CMAKE_CURRENT_SOURCE_DIR}/pkg-config.pc.cmake"
  "${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}.pc"
)
```

A typical `.pc` template could be:

```
Name: ${PROJECT_NAME}
Description: ${PROJECT_DESCRIPTION}
Version: ${PROJECT_VERSION}
Requires: ${PKG_CONFIG_REQUIRES}
prefix=${CMAKE_INSTALL_PREFIX}
```

```
includedir=${PKG_CONFIG_INCLUDEDIR}
libdir=${PKG_CONFIG_LIBDIR}
Libs: ${PKG_CONFIG_LIBS}
Cflags: ${PKG_CONFIG_CFLAGS}
```

Where the following variables are provided by CMake:

- PROJECT_NAME
- PROJECT_DESCRIPTION
- PROJECT_VERSION
- CMAKE_INSTALL_PREFIX

And these ones need to be specified explicitly:

- PKG_CONFIG_REQUIRES
- PKG_CONFIG_INCLUDEDIR
- PKG_CONFIG_LIBDIR
- PKG_CONFIG_LIBS
- PKG_CONFIG_CFLAGS

Example:

```
SET(PKG_CONFIG_REQUIRES glib-2.0)
SET(PKG_CONFIG_LIBDIR
    "\${prefix}/lib"
)
SET(PKG_CONFIG_INCLUDEDIR
    "\${prefix}/include/mylib"
)
SET(PKG_CONFIG_LIBS
    "-L\${libdir} -lmylib"
)
SET(PKG_CONFIG_CFLAGS
    "-I\${includedir}"
)

CONFIGURE_FILE(
  "${CMAKE_CURRENT_SOURCE_DIR}/pkg-config.pc.cmake"
  "${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}.pc"
)
```

Installing files on target

Installing files on target is as simple as adding the corresponding `INSTALL` command to the target `CMakeLists.txt`.

To install the main targets of a project, use the `TARGETS` directive:

```
INSTALL(TARGETS myapp
    DESTINATION bin)
```

or

```
INSTALL(TARGETS mylib ARCHIVE
    DESTINATION lib)
```

> Note: The files will be installed relatively to the path specified in the `CMAKE_INSTALL_PREFIX` cmake variable, prepended by the `DESTDIR` variable passed on the command line (ie `make install DESTDIR=/home/toto`)

Other project files can also be installed using the `FILES` directive:

```
INSTALL(FILES header.h
    DESTINATION include/mylib)
```

or

```
INSTALL(FILES "${CMAKE_BINARY_DIR}/${PROJECT_NAME}.pc"
    DESTINATION lib/pkgconfig)
```

Building the project

I personnally always recommend to build a project out-of-tree, ie to put all build subproducts into a separate directory. Incidentally, building out-of-tree is also a good way to find out if your project is properly configured …

So, the first step is to create a build directory

```
mkdir build && cd build
```

Then you need to tell CMake to generate the project makefiles according to specific directives you may specify on the command line (typically by setting variables). Most of the time, you can let CMake

apply default values:

```
cmake ..
```

But you may need for instance to specify a custom installation prefix (by default CMake will use `usr/local`):

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=usr ..
```

Once the makefiles have been generated you can simply build the project using make commands.

```
make
```

Finally, you can install the targets, either using defaults …

```
make install
```

… or specifying the destination directory (CMake use `/` as the default destination directory)

```
DESTDIR=/custom-destdir make install
```

comments powered by Disqus

Hi

I am David Corvoysier, versatile developer and open Source enthusiast.

Follow me: 

Related Posts

Explore Tensorflow features with the CIFAR10 dataset

26 Jun 2017 by David Corvoysier

The reason I started using Tensorflow was because of the limitations of my experiments so far, where I had coded my models from scratch following the guidance of the [CNN for visual recognition] (http://cs231n.github.io/) course. I already knew how CNN worked, and had already a good experience of what it takes to train a good model. I had also read a lot of papers presenting multiple variations of CNN topologies, those aiming at increasing accuracy like those aiming at reducing model complexity and size. I work in the embedded world, so performance is obviously one of my

primary concern, but I soon realized that the CNN state of the art for computer vision had not reached a consensus yet on the best compromise between accuracy and performance.

(more…)

Categories: Machine Learning Tags: tensorflow CIFAR10 CNN

Build and boot a minimal Linux system with qemu

23 Sep 2016 by David Corvoysier

When you want to build a Linux system for an embedded target these days, it is very unlikely that you decide to do it from scratch. Embedded Linux build systems are really smart and efficients, and will fit almost all use cases: should you need only a simple system, [buildroot](https://buildroot.org/) should be your first choice, and if you want to include more advanced features, or even create a full distribution, [Yocto](https://www.yoctoproject.org/) is the way to go. That said, even if these tools will do all the heavy-lifting for you, they are not perfect, and if you are using less common configurations, you may stumble upon issues that were not expected. In that case, it may be important to understand what happens behind the scenes. In this post, I will describe step-by-step how you can build a minimal Linux system for an embedded target and boot it using [QEMU] (http://wiki.qemu.org/Main_Page).

(more…)

Categories: System Tags: build linux croostool-ng qemu busybox

Benchmarking build systems for a large C project

01 Sep 2016 by David Corvoysier

The performance of build systems has been discussed at large in the developer community, with a strong emphasis made on the limitations of the legacy Make tool when dealing with large/complex projects. I recently had to develop a build-system to create firmwares for embedded targets from more than 1000 source files. The requirements were to use build recipes that could be customized for each directory and file in the source tree, similar to what the Linux Kernel does with [kbuild] (https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt). I designed a custom recursive Make solution inspired by [kbuild](https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt).

(more…)

Categories: System Tags: build make ninja kbuild cmake

Decentralized modules declarations in C using ELF sections

17 Aug 2016 by David Corvoysier

In modular programming, a standard practice is to define common interfaces allowing the same type of operation to be performed on a set of otherwise independent modules. ~~~~ modules = [a,b,...] for each m in modules: m.foo m.bar ~~~~ To implement this pattern, two mechanisms are required: - instantiation, to allow each module to define an 'instance' of the common interface, - registration, to allow each module to 'provide' this instance to other modules. Instantiation is typically supported natively in high-level languages. Registration is more difficult and usually requires specific code to be written, or relying on external frameworks. Let's see how these two mechanisms can be implemented for C programs.

(more…)

Categories: System Tags: ELF sections

Tag cloud

SVG XHTML Cache HTTP HTML HTML5 Video CSS3 Javascript Animation Benchmark Carousel Canvas video cross-compilation DBus CMake build

Generated by Jekyll