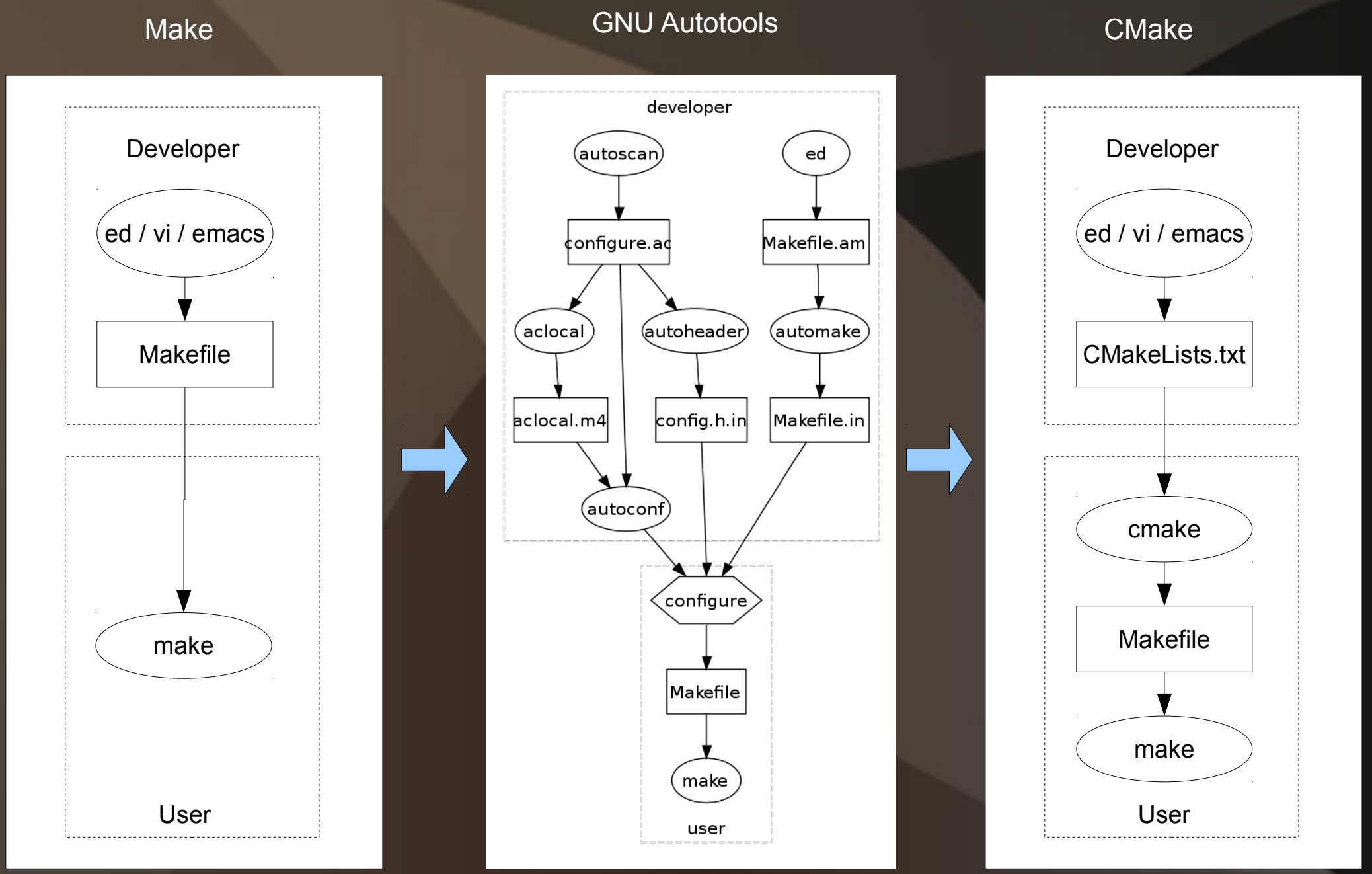
CMake
Behind the Scenes of Code Development

Rodolfo Lima
rodolfo@digitok.com.br

Outline

- Motivation
- CMake features
- Basic usage
- Medium-sized projects
- FLTK-based projects
- CMake scripting
- Multi-platform environments
- Troubleshooting
- Conclusion
- References

Motivation


Make
Developer
ed / vi / emacs
Makefile
make
User
GNU Autotools
developer
autoscan
ed
configure.ac
Makefile.am
aclocal
autoheader
automake
aclocal.m4
config.h.in
Makefile.in
autoconf
configure
Makefile
make
user
CMake
Developer
ed / vi / emacs
CMakeLists.txt
cmake
Makefile
make
User

Kitware's CMake Features

- Multi-environment / Multi-platform
  - Visual Studio projects, Make/GCC, XCode, you name it.
- Eases project maintenance
  - One “recipe” to rule them all :)
- High scalability
- C/C++ header dependency analysis
- Multiple languages (C, C++, Fortran)
- Human-parseable project definition
- Cross-compiling, canadian-cross style
- Nice compiler output formatter (when using Make)
- Can build project installers (works with Nullsoft's NSIS on Windows)
- Automated testsuites
- KDE and OpenCV use CMake, it must be good

Basic Usage

1. Create program source: main.c

```
#include <stdio.h>
int main()
{
    printf("Hello, nurse!\n");
    return 0;
}
```

2. Create project definition: CMakeLists.txt

```
project(hello)

add_executable(hello main.c)
```

3. Generate Makefile

```
rodolfo@sabbath ~ $ cmake .

-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/rodolfo
```

4. Compile!

```
rodolfo@sabbath ~ $ make
Scanning dependencies of target hello
[100%] Building C object CMakeFiles/hello.dir/main.c.o
Linking C executable hello
[100%] Built target hello
```

Filtered Make Output

```
rodolfo@sabbath ~/src/panostitch/rel/src $ make
[  0%] Generating ../../src/pch.h.gch/panostitch_RelWithDebInfo.h++
[  2%] Built target panostitch_pch.h
[  2%] Building C object lib/sba/CMakeFiles/sba.dir/sba_levmar.c.o
/home/rodolfo/src/panostitch/lib/sba/sba_levmar.c: In function 'emalloc_':
/home/rodolfo/src/panostitch/lib/sba/sba_levmar.c:69: warning: division by zero
[  4%] Building C object lib/sba/CMakeFiles/sba.dir/sba_levmar_wrap.c.o
[  4%] Building C object lib/sba/CMakeFiles/sba.dir/sba_lapack.c.o
[  7%] Building C object lib/sba/CMakeFiles/sba.dir/sba_crsm.c.o
[  7%] Building C object lib/sba/CMakeFiles/sba.dir/sba_chkjac.c.o
Linking C static library libsba.a
[  7%] Built target sba
[  7%] Building C object lib/levmar/CMakeFiles/levmar.dir/lm.c.o
[  9%] Building C object lib/levmar/CMakeFiles/levmar.dir/Axb.c.o
[  9%] Building C object lib/levmar/CMakeFiles/levmar.dir/misc.c.o
[ 12%] Building C object lib/levmar/CMakeFiles/levmar.dir/lmlec.c.o
[ 12%] Building C object lib/levmar/CMakeFiles/levmar.dir/lmbc.c.o
[ 14%] Building C object lib/levmar/CMakeFiles/levmar.dir/lmblec.c.o
Linking C static library liblevmar.a
[ 14%] Built target levmar
```

...and so on until 100% or a compiler/linker error stops the process

Created Targets

- Main target: hello
  - Builds the application
- clean:
  - Cleans up (some) generated files
- depends:
  - Rebuilds file dependencies in case something “feels” wrong
- Object file: main.o (per source file)
- Preprocessed source: main.i (per source file)
- Assembly output: main.s (per source file)
- If some dependent file changes, its targets will be rebuild automatically during “make”.
- If CMakeLists.txt changes, cmake will be run automatically during “make” to recreate the build tree
  - This works even in Visual Studio (the project gets reloaded)

Build Types

- Predefined compiler parameters according to build type
  - Debug – used during development
  - Release – used in production code
  - RelWithDebInfo – helpful when debugging production code
  - MinSizeRel – generates space optimized code
- Specified during build tree creation
  - `cmake . -DCMAKE_BUILD_TYPE=(Debug|RelWithDebInfo|...)`
- Good strategy: out-of-source builds
- Compiler parameters can be customized by setting variables
  - `set(CMAKE_C_FLAGS_DEBUG "${CMAKE_C_FLAGS_DEBUG} -ggdb")`
  - `set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -O3")`

External Libraries

- Search installed libraries *multiplatform-ly*

```
project(parser)

find_package(LibXml2 REQUIRED)

include_directories(${LIBXML2_INCLUDE_DIR})

add_executable(parser main.cpp)
target_link_libraries(parser ${LIBXML2_LIBRARIES})
```

- A **lot** of libraries supported
  - OpenGL, FLTK, wxWidgets, Boost, SDL, BLAS, FreeType,...
  - You can write your own finder (not for the faint-hearted)

Library Projects

- Creation as easy as executables

```
add_library(<name> [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
           source1 source2 source2...)
```

- When library type isn't specified:
  - use globally defined variable
    `BUILD_SHARED_LIBS=<true,false>`
- Executables AND libraries can have built libraries as dependencies
  - GNU “convenience libraries” concept
- Platform idiosyncrasies taken care of

Medium-Sized Projects

- Multiple source directories
- Custom processing not involving source files
- Typical source tree:

  - root
    - src
    - lib
      - lib1
      - lib2
      - lib3
    - testsuite
    - res
    - doc

- Each directory gets its CMakeLists.txt
- Parent directory adds its children using:
  - add_subdirectory(<subdir>)
- Build tree must be configured in root directory!
  - CMake intelligently configures the whole directory tree
- Targets from other directories can be used anywhere in the source tree
  - target_link_libraries(hello lib1 lib2 lib3)

Source File Configuration

- Sometimes C/C++ macros in source files are needed to cope with different systems
- `configure_file` and `add_definitions` comes to rescue:

```
configure_file(<input> <output> [COPYONLY] [ESCAPE_QUOTES] [@ONLY])
add_definitions(-DFOO -DBAR ...)
```

- Example

CMakeLists.txt

```
project(image_suite)

set(VERSION 3.0)
set(PACKAGE_NAME ${CMAKE_PROJECT_NAME})

find_package(JPEG)
find_package(PNG)
find_package(LibXml2)

configure_file(config.h.in config.h
           @ONLY)
if(LIBXML2_FOUND)
   add_definitions(-DHAS_XML=1)
endif()

add_executable(convert convert.cpp)
```

config.h.in

```
#cmakedefine PNG_FOUND 1
#cmakedefine JPEG_FOUND 1
#define PACKAGE_NAME “@PACKAGE_NAME@”
#define VERSION “@VERSION@”
```

Generated config.h

```
#define PNG_FOUND 1
// #undef JPEG_FOUND
#define PACKAGE_NAME “image_suite”
#define VERSION “3.0”
```

convert.cpp excerpt

```
#include “config.h”

#if JPEG_FOUND
   // do jpeg stuff
#endif
```

Custom Targets

- When custom-build files / actions are needed
  - flex / bison / moc / …
- Command to be used:

```
add_custom_target(<name> [ALL] [command1 [args1...]]
                  [COMMAND command2 [args2...] ...]
                  [DEPENDS depend depend depend ... ]
                  [WORKING_DIRECTORY dir]
                  [COMMENT comment] [VERBATIM]
                  [SOURCES src1 [src2...]])
```

- Example 1

```
add_custom_target(parser.c bison -o parser.c parser.y
                  SOURCES parser.y)
add_executable(hello main.c parser.c)
```

- Example 2

```
add_custom_target(car.png.c bin2c car.png
                  SOURCES car.png)
add_executable(bin2c bin2c.c)

add_executable(hello main.c car.png.c)
```

FLTK-Based Projects

- Includes commands to deal with fluid-generated files
- Example:

```
project(image_suite)

find_package(FLTK)
find_package(JPEG)
find_package(PNG REQUIRED)

add_library(convert convert.cpp)
target_link_libraries(convert ${PNG_LIBRARIES})
include_directories(${PNG_INCLUDE_DIR})

if(JPEG_FOUND)
   target_link_libraries(convert ${JPEG_LIBRARIES})
   include_directories(${JPEG_INCLUDE_DIR})
endif()

if(FLTK_FOUND)
   fltk_wrap_ui(viewer viewer.fl)
   add_executable(viewer viewer.cpp ${viewer_FLTK_UI_SRCS})
   target_link_libraries(viewer convert ${FLTK_LIBRARIES})
   include_directories(${FLTK_INCLUDE_DIRS})
   link_directories(${FLTK_LIBRARY_DIRS})
endif()
```

CMake Scripting

- Needed for special processing
- Includes typical programming language statements
  - Loops
  - Condition
  - Variables
  - Lists
  - Macros
  - “Functions”
- I'd rather use Lua, but it's too late now
- Example:

```
macro(add_tool name)
    add_custom_target(${name}.c create_tool ${name})
    add_executable(${name} ${name}.c)
    target_link_libraries(${name} ${Boost_LIBRARIES})
endmacro()

add_tool(resize)
add_tool(invert)
add_tool(mirror)
add_tool(crop)
```

Some Useful Statements

- For loop

```
foreach(tool resize invert mirror crop)
    message("Preparing tool ${tool}")
    add_custom_target(${tool}.c create_tool ${tool})
    add_executable(${tool} ${tool}.c)
    target_link_libraries(${tool} ${Boost_LIBRARIES})
endforeach()
```

- If clause

```
if(WIN32)
    do_something_weird()
elseif(APPLE OR UNIX)
    do_something_neat()
endif()

if(EXISTS some_file.dat)
    process(some_file.dat)
endif()

if(additional_file MATCHES "^file_.*$")
    process(${additional_file})
endif()
```

Lists

- Useful to manage long list of elements
- Elements can be manipulated depending on running platform
  - Useful for source file lists
- Example:

```
set(sources viewer.cpp config.cpp)

if(WIN32)
   list(APPEND sources viewer_mfc.cpp)
elseif(UNIX)
   list(APPEND sources viewer_gtk.cpp)
else
   message(FATAL “Platform not supported”)
endif()

add_executable(viewer ${sources})

list(LENGTH sources srclen)
message(“${srclen} source files”)

foreach(src ${sources})
   message(“Source: ${src}”)
endforeach()
```

Multi-platform Environments

- CMake generates project files for several environments
- It detects during build tree creation which environment the user is on
- If detected environment isn't the one you want, use:
  - `cmake . -G "<generator type>"`
- Generator type can be:
  - Unix: "Unix Makefiles", "CodeBlocks – Unix Makefiles", "Eclipse CDT4 – Unix Makefiles", "KDevelop3", …
  - Windows: "MinGW Makefiles", "MSYS Makefiles", "NMake Makefiles", "Borland Makefiles", "Watcom WMake", "Unix Makefiles" (cygwin), "Visual Studio 8 2005", "Visual Studio 9 2008",...
  - MacOS X: "KDevelop3", "Unix Makefiles", "XCode"

Troubleshooting

- Sometimes an installed library cannot be found
  - Edit generated CMakeCache.txt manually to inform cmake where the library really is.
  - CMakeCache.txt excerpt:

```
//The Boost FILESYSTEM library
Boost_FILESYSTEM_LIBRARY:FILEPATH=/usr/lib/libboost_filesystem-mt-1_40.so

//Path to a library.
Boost_FILESYSTEM_LIBRARY_DEBUG:FILEPATH=Boost_FILESYSTEM_LIBRARY_DEBUG-NOTFOUND

//Path to a library.
Boost_FILESYSTEM_LIBRARY_RELEASE:FILEPATH=/usr/lib/libboost_filesystem-mt-1_40.so

//Path to a file.
Boost_INCLUDE_DIR:PATH=/usr/include/boost-1_40
```

- Delete CMakeCache.txt to make CMake's library search start from scratch
  - Useful when installed libraries or tools have changed

Troubleshooting

- Linker errors difficult to pinpoint with filtered linker output
- Headers not included as they should
- Sometimes it's useful to see how the linker/compiler is being called
- Solution: `make VERBOSE=1`

```
rodolfo@sabbath ~/tst2 $ make VERBOSE=1

                                        … skip …

[100%] Building C object CMakeFiles/hello.dir/main.c.o
/usr/bin/gcc    -o CMakeFiles/hello.dir/main.c.o   -c /home/rodolfo/main.c
Linking C executable hello
/usr/bin/cmake -E cmake_link_script CMakeFiles/hello.dir/link.txt --verbose=1
/usr/bin/gcc     CMakeFiles/hello.dir/main.c.o  -o hello -rdynamic -lxml2
make[2]: Leaving directory `/home/rodolfo'
/usr/bin/cmake -E cmake_progress_report /home/rodolfo/CMakeFiles  1
[100%] Built target hello
make[1]: Leaving directory `/home/rodolfo'
/usr/bin/cmake -E cmake_progress_start /home/rodolfo/CMakeFiles 0
```

Conclusion

- Build systems are underated in general (but shouldn't)
  - When not thought out well, development time shifts towards build system tweaking instead of source file coding.
- Projects tend to grow bigger over time
  - Scalable build system desired
- Multi-platform development already a reality
- Stubborn coders hate working on different development environments

CMake takes care of your build system (with a little help from you),
you take care of your code.

References

- CMake homepage: http://www.cmake.org
- K. Martin and B. Hoffman – *Mastering CMake: A Cross-Platform Build System*, Kitware Inc., 2003
- K. Martin and B. Hoffman – *An Open Source Approach to Developing Software in a Small Organization*, in IEEE Software, Vol. 24 Number 1 IEEE, January 2007
- cmake manpage (very compreensive)
- CMake wikipage: http://www.cmake.org/Wiki/CMake