CMake - Cross-Platform Make

R. Douglas Barbieri

Made to Order Software Corporation

Introduction

What is CMake?

- Control the software compilation process using simple platform-independent and compiler-independent configuration files
- Generate native makefiles and workspaces that can be used in the compiler environment of your choice
- Provides packaging (using CPack) and testing (using CTest)

Who is using it?

- Linden Lab (for their Second Life project)
- KDE4
- Boost
- MySQL
- The Half-Life 2 SDK
- Rosegarden
- Loads of others (see http://www.cmake.org/Wiki/CMake_Projects) for more details

Why CMake?

- Better than other build systems:
  - Custom GNU/Makefiles
  - autoconf/automake
  - CodeBlocks other FOSS IDEs (sadly, not DevC++)
  - Microsoft Visual Studio and other proprietary IDEs
  - Apple XCode

Why CMake (con't)

- Can target multiple compilers, build systems and IDEs using a single set of configuration files
- Other build systems are difficult to set up and debug (particulary autoconf/automake)
- Has a simple to use language to allow customization for multiple platforms with relative ease.
- Great even for a single platform!

How Does it Work?

- CMake uses ”Generators” to create your target build files
- Uses configuration files to target your particular system
- Uses your custom config or existing rules to locate and build against third party libraries
- Provides a simple language to help customize for platform-specific idioms

Generators

- What are CMake Generators?
  - They can produce make/project files for many different IDEs, GNU/Make and Microsoft's Nmake
  - Customized for your specific platform
  - Able to produce tailored project files specific to your favorite development IDE or system
  - Lots are available!

Generators

GNU/Linux:

```
$ cmake
 The following generators are available on this platform:
  Unix Makefiles                Generates standard UNIX makefiles.
  CodeBlocks - Unix Makefiles   Generates CodeBlocks project files.
  Eclipse CDT4 - Unix Makefiles Generates Eclipse CDT 4.0 project files.
  KDevelop3                     Generates KDevelop 3 project files.
  KDevelop3 - Unix Makefiles    Generates KDevelop 3 project files.
```

M$ Windows:

```
C:\ cmake
  Borland Makefiles     Generates Borland makefiles.
  MSYS Makefiles        Generates MSYS makefiles.
  MinGW Makefiles       Generates a make file for use with mingw32-make.
  NMake Makefiles       Generates NMake makefiles.
  Unix Makefiles        Generates standard UNIX makefiles.
```

Generators (cont'd)

Visual Studio 6	Generates Visual Studio 6 project files.
Visual Studio 7	Generates Visual Studio .NET 2002 project files.
Visual Studio 7 .NET 2003	Generates Visual Studio .NET 2003 project files.
Visual Studio 8 2005	Generates Visual Studio .NET 2005 project files.
Visual Studio 8 2005 Win64	Generates Visual Studio .NET 2005 Win64 project files.
Visual Studio 9 2008	Generates Visual Studio 9 2008 project files.
Visual Studio 9 2008 Win64	Generates Visual Studio 9 2008 Win64 project files.
Watcom WMake	Generates Watcom WMake makefiles.
CodeBlocks - MinGW Makefiles	Generates CodeBlocks project files.
CodeBlocks - Unix Makefiles	Generates CodeBlocks project files.
Eclipse CDT4 - MinGW Makefiles	Generates Eclipse CDT 4.0 project files.
Eclipse CDT4 - NMake Makefiles....	

...and etc.

Generators (cont'd)

MacOS/X:

$ cmake
The following generators are available on this platform:

Unix Makefiles	Generates standard UNIX makefiles.
Xcode	Generate XCode project files.
CodeBlocks - Unix Makefiles	Generates CodeBlocks project files.
Eclipse CDT4 - Unix Makefiles	Generates Eclipse CDT 4.0 project files.
KDevelop3	Generates KDevelop 3 project files.
KDevelop3 - Unix Makefiles	Generates KDevelop 3 project files.

Generators (cont'd)

As you can see....
loads of
GENERATORS!!!!

Configuration Script Syntax

Here is a simple example (CMakeLists.txt):

```
# The name of our project is "HELLO". CMakeLists files in this project can
# refer to the root source directory of the project as ${HELLO_SOURCE_DIR} and
# to the root binary directory of the project as ${HELLO_BINARY_DIR}.
cmake_minimum_required (VERSION 2.6)
project (HELLO)
# Add executable called "helloDemo" that is built from the source files
# "demo.cxx" and "demo_b.cxx". The extensions are automatically found.
add_executable (helloDemo demo.cxx demo_b.cxx)
```

Syntax (cont'd)

Add a library:

```
add_subdirectory (Hello)
# Make sure the compiler can find include files from our Hello library.
include_directories (${HELLO_SOURCE_DIR}/Hello)
# Make sure the linker can find the Hello library once it is built.
link_directories (${HELLO_BINARY_DIR}/Hello)

add_executable (helloDemo demo.cxx demo_b.cxx)

# Link the executable to the Hello library.
target_link_libraries (helloDemo Hello)
```

Library Syntax

Library CMakeLists.txt:

```
# Create a library called "Hello" which includes the
    source file "hello.cxx".
# Any number of sources could be listed here.
add_library (Hello hello.cxx)
```

Demo

hello_world

Demo Time!

Demo

- Tutorial
  - Step 1 – simple example with a configure file (.in)
  - Step 2 – with a user-configurable option
  - Step 3 – add install target and tests
  - Step 4 – using a macro
  - Step 6 – add installer commands
  - Step 7 – turn on dashboard scripting

Demo

- SLiteChat – an open-source text chat client for Second Life (http://www.slitechat.org/)
- Second Life – the official 3D viewer for Second Life (http://www.secondlife.com/)

Q and A

Ask me questions!
And
Thanks for coming to my talk!