CMake && Friends

Open source tools to build, test and package software
CMake, CTest, CPack, CDash

CMake && Friends

Open source tools to build, test and package software
CMake, CTest, CPack, CDash

Preface

ANL has CMake on Theta and Cooley

CMake can be acquired from numerous locations for your local machine

https://www.cmake.org/download/

From your Linux distribution

Visual Studio 2017+

apt.kitware.com for Debian and Ubuntu

Snap universal linux package

pip install cmake

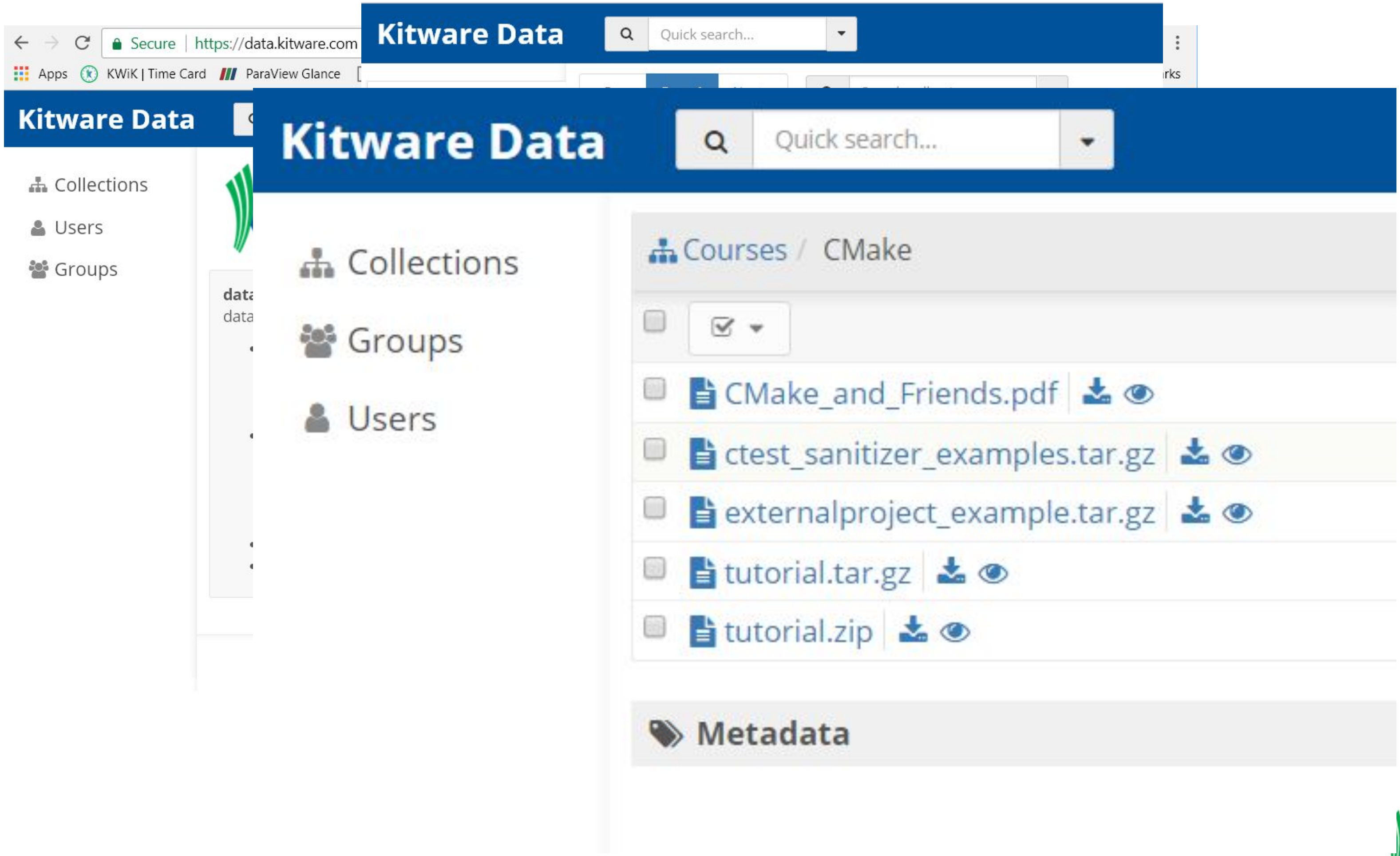
homebrew

Preface

You will need to install the exercises

Download at:

<ANL> resource


Kitware Data

Quick search...

Collections

Groups

Users

Courses / CMake

CMake_and_Friends.pdf

ctest_sanitizer_examples.tar.gz

externalproject_example.tar.gz

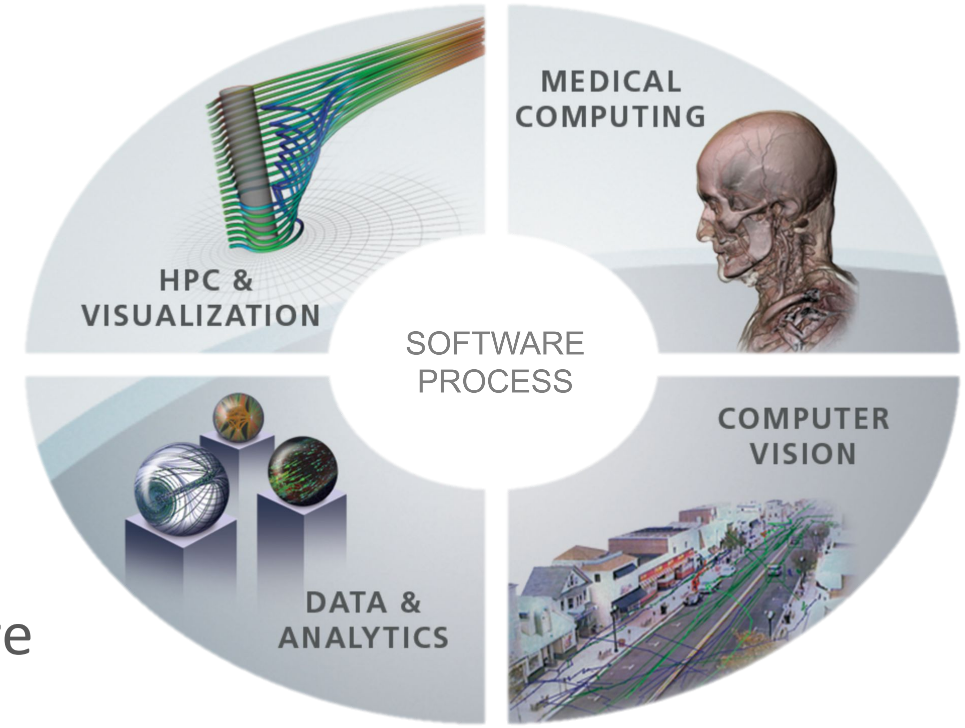
tutorial.tar.gz

tutorial.zip

Metadata

Collaborative software R&D

- Technical computing
- Algorithms & applications
- Software process & infrastructure
- Support & training
- Open source leadership



Successful small business

- Founded in 1998; privately owned; no debt
- 170 employees; 1/3 masters, 1/3 PhD
- $31M revenue in 2018

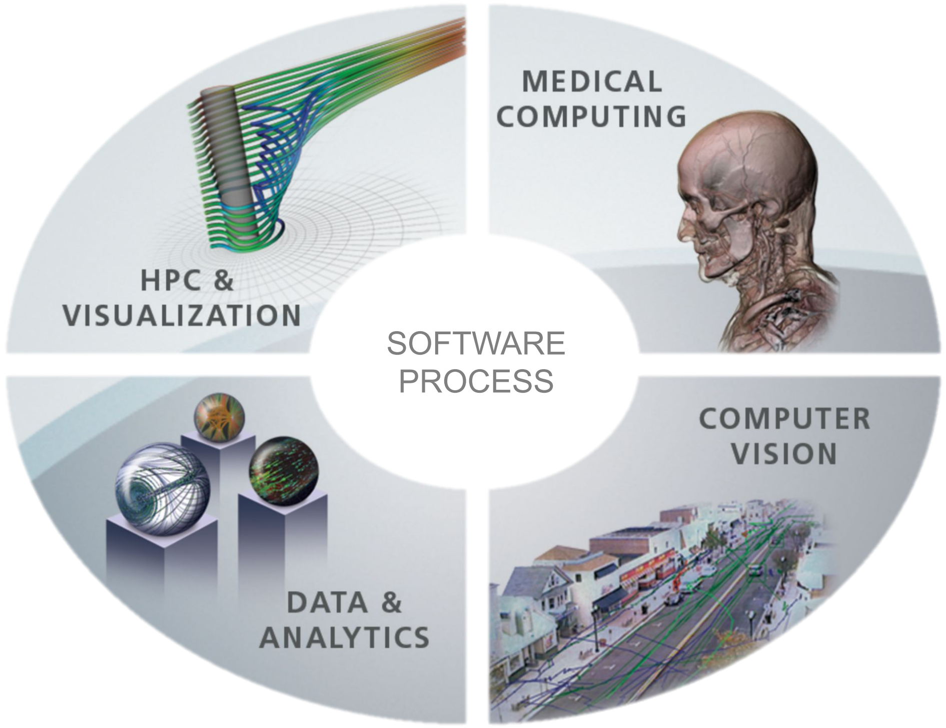
Awards

- HPC Best visualization product
- Tibbetts award for outstanding research
- 6-Time Inc. 5000 Honoree

2006 National Tibbetts Award
Presented to
Loki Incorporated
In Recognition of your Outstanding Contributions to the
SBIR Program
SBTC

Supporting all sectors

75+	Academic institutions
50+	Government agencies and labs
100+	Enterprises



Open source platforms

VTK & ParaView interactive visualization and analysis for scientific data

ITK & 3D Slicer medical image analysis and personalized medicine research

CMake cross-platform build system

- CDash, CTest, CPack, software process tools

Resonant informatics and infovis

KWIVER computer vision image and video analysis

Simulation, ultrasound, physiology, information security, materials science, …

The VTK User's Guide
Install, Use and Extend The Visual
The ITK Software Guide
The Insight Segmentation and Reg
The ParaView Guide
A Parallel Visualization Application
Mastering CMake
A Cross-Platform Build System
Luis Ibáñez
William Schroeder

Introduction to CMake

What is a Build System?

In the beginning there was the command line:

```
% gcc hello.cxx -o hello
```

What is a Build System?

Maybe use a shell script to avoid typing

buildhello.sh

```
#!/bin/sh
 gcc hello.cxx -o hello


% buildhello.sh
```

What is a Build System?

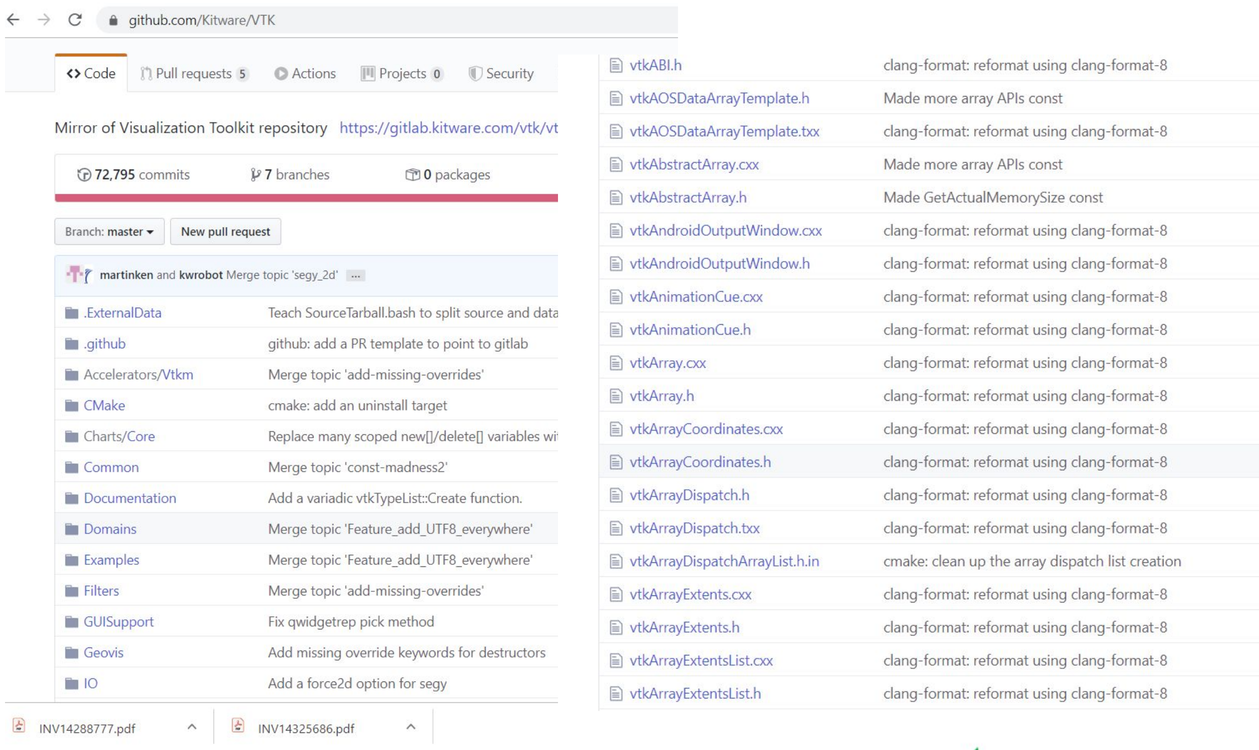
Even better use the make tool

Makefile

```
hello: hello.cxx
    gcc hello.cxx
% make hello
// what about #include files?
```

What is a Build System?

Shell scripts and command lines and simple makefiles work for a few files, but not for lots of files like VTK



github.com/Kitware/VTK

Mirror of Visualization Toolkit repository https://gitlab.kitware.com/vtk/vt

72,795 commits | 7 branches | 0 packages

martinken and kwrobot Merge topic 'segy_2d' …

.ExternalData	Teach SourceTarball.bash to split source and data
.github	github: add a PR template to point to gitlab
Accelerators/Vtkm	Merge topic 'add-missing-overrides'
CMake	cmake: add an uninstall target
Charts/Core	Replace many scoped new[]/delete[] variables wi
Common	Merge topic 'const-madness2'
Documentation	Add a variadic vtkTypeList::Create function.
Domains	Merge topic 'Feature_add_UTF8_everywhere'
Examples	Merge topic 'Feature_add_UTF8_everywhere'
Filters	Merge topic 'add-missing-overrides'
GUISupport	Fix qwidgetrep pick method
Geovis	Add missing override keywords for destructors
IO	Add a force2d option for segy

INV14288777.pdf   INV14325686.pdf

vtkABI.h	clang-format: reformat using clang-format-8
vtkAOSDataArrayTemplate.h	Made more array APIs const
vtkAOSDataArrayTemplate.txx	clang-format: reformat using clang-format-8
vtkAbstractArray.cxx	Made more array APIs const
vtkAbstractArray.h	Made GetActualMemorySize const
vtkAndroidOutputWindow.cxx	clang-format: reformat using clang-format-8
vtkAndroidOutputWindow.h	clang-format: reformat using clang-format-8
vtkAnimationCue.cxx	clang-format: reformat using clang-format-8
vtkAnimationCue.h	clang-format: reformat using clang-format-8
vtkArray.cxx	clang-format: reformat using clang-format-8
vtkArray.h	clang-format: reformat using clang-format-8
vtkArrayCoordinates.cxx	clang-format: reformat using clang-format-8
vtkArrayCoordinates.h	clang-format: reformat using clang-format-8
vtkArrayDispatch.h	clang-format: reformat using clang-format-8
vtkArrayDispatch.txx	clang-format: reformat using clang-format-8
vtkArrayDispatchArrayList.h.in	cmake: clean up the array dispatch list creation
vtkArrayExtents.cxx	clang-format: reformat using clang-format-8
vtkArrayExtents.h	clang-format: reformat using clang-format-8
vtkArrayExtentsList.cxx	clang-format: reformat using clang-format-8
vtkArrayExtentsList.h	clang-format: reformat using clang-format-8

What is a Build System?

A build system manages all of the source files and the complicated interdependencies for a project and turns them into hopefully useful programs and libraries.

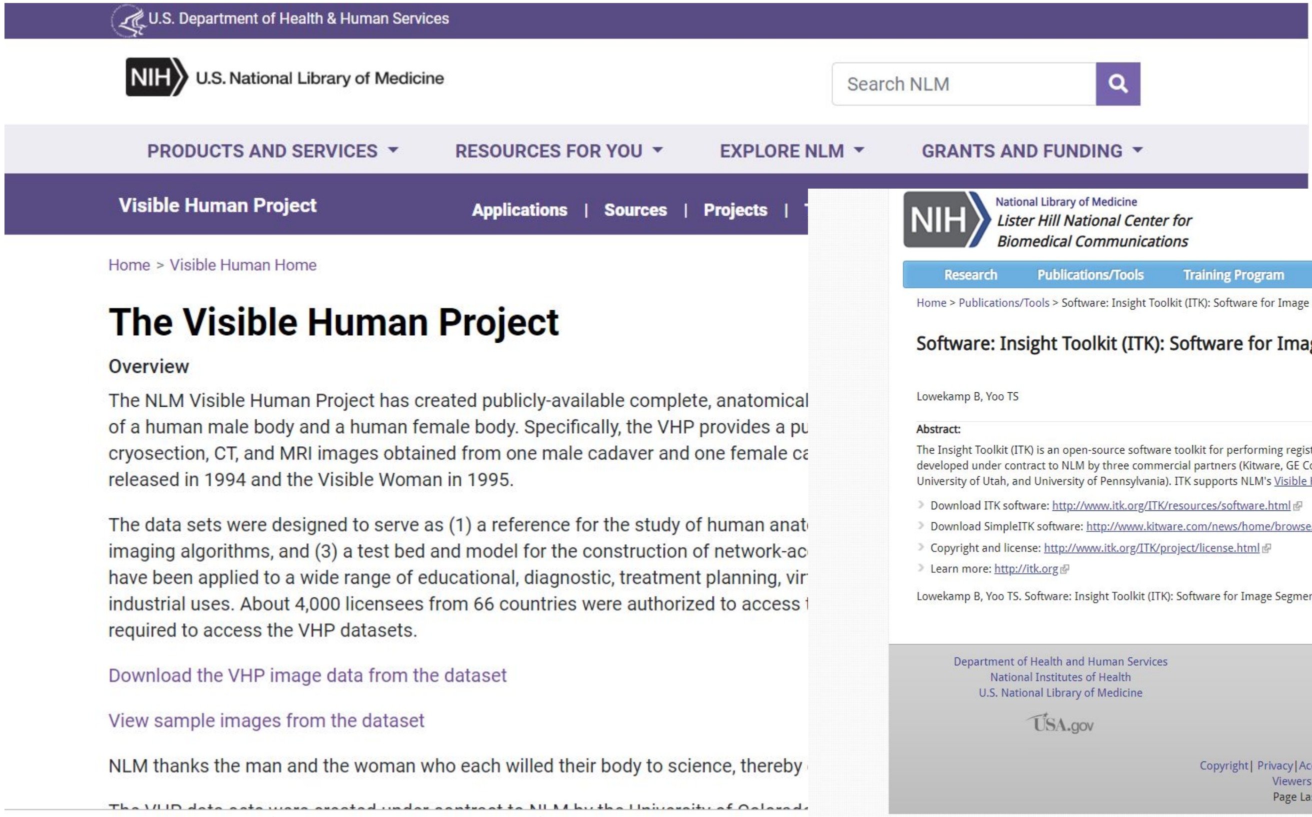
What is CMake?

- CMake is the **cross-platform**, **open-source build system** that lets you use the **native development tools** you love the most.
- It's a build system **generator**
- It takes **plain text files** as input that describe your project and **produces** project files or make files for use with a wide variety of **native development tools.**
- Family of Software Development Tools
  - Build = CMake Test = CTest/CDash Package = CPack

How CMake Changes The Way We Build C++

- Boost aims to give C++ a set of useful libraries like Java, Python, and C#
- CMake aims to give C++ compile portability like the compile once and run everywhere of Java, Python, and C#
  - Same build tool and files for all platforms
  - Easy to mix both large and small libraries

Where did CMake come from?



U.S. Department of Health & Human Services

NIH U.S. National Library of Medicine

Search NLM

PRODUCTS AND SERVICES | RESOURCES FOR YOU | EXPLORE NLM | GRANTS AND FUNDING

Visible Human Project — Applications | Sources | Projects |

Home > Visible Human Home

The Visible Human Project

Overview

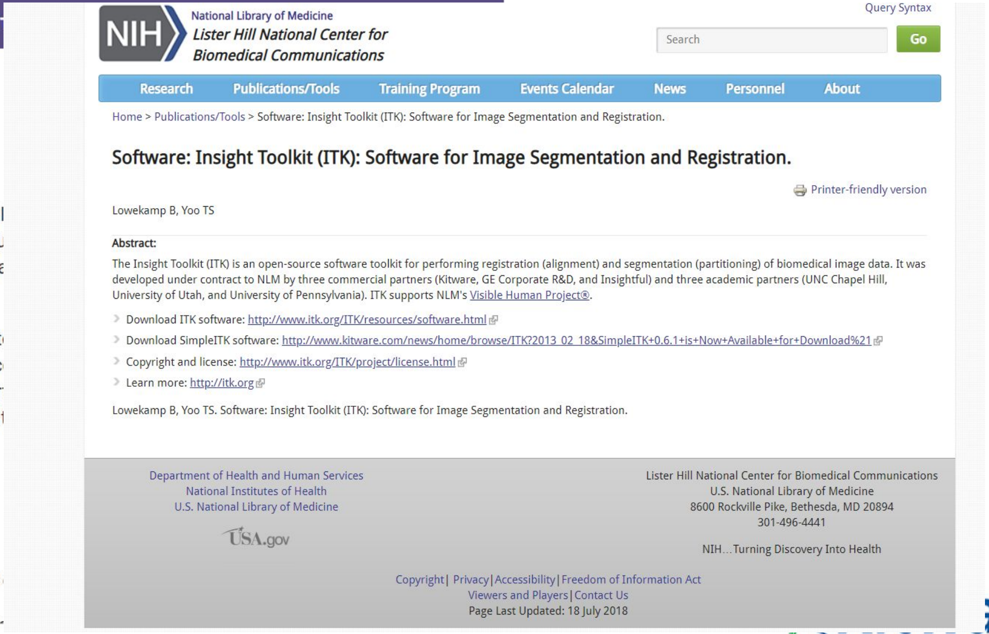
The NLM Visible Human Project has created publicly-available complete, anatomical
of a human male body and a human female body. Specifically, the VHP provides a pu
cryosection, CT, and MRI images obtained from one male cadaver and one female ca
released in 1994 and the Visible Woman in 1995.

The data sets were designed to serve as (1) a reference for the study of human anat
imaging algorithms, and (3) a test bed and model for the construction of network-ac
have been applied to a wide range of educational, diagnostic, treatment planning, vir
industrial uses. About 4,000 licensees from 66 countries were authorized to access t
required to access the VHP datasets.

Download the VHP image data from the dataset

View sample images from the dataset

NLM thanks the man and the woman who each willed their body to science, thereby



NIH National Library of Medicine *Lister Hill National Center for Biomedical Communications*

Query Syntax

Search Go

Research | Publications/Tools | Training Program | Events Calendar | News | Personnel | About

Home > Publications/Tools > Software: Insight Toolkit (ITK): Software for Image Segmentation and Registration.

Software: Insight Toolkit (ITK): Software for Image Segmentation and Registration.

Printer-friendly version

Lowekamp B, Yoo TS

Abstract:

The Insight Toolkit (ITK) is an open-source software toolkit for performing registration (alignment) and segmentation (partitioning) of biomedical image data. It was developed under contract to NLM by three commercial partners (Kitware, GE Corporate R&D, and Insightful) and three academic partners (UNC Chapel Hill, University of Utah, and University of Pennsylvania). ITK supports NLM's Visible Human Project®.

- Download ITK software: http://www.itk.org/ITK/resources/software.html
- Download SimpleITK software: http://www.kitware.com/news/home/browse/ITK?2013_02_18&SimpleITK+0.6.1+is+Now+Available+for+Download%21
- Copyright and license: http://www.itk.org/ITK/project/license.html
- Learn more: http://itk.org

Lowekamp B, Yoo TS. Software: Insight Toolkit (ITK): Software for Image Segmentation and Registration.

Department of Health and Human Services
National Institutes of Health
U.S. National Library of Medicine

USA.gov

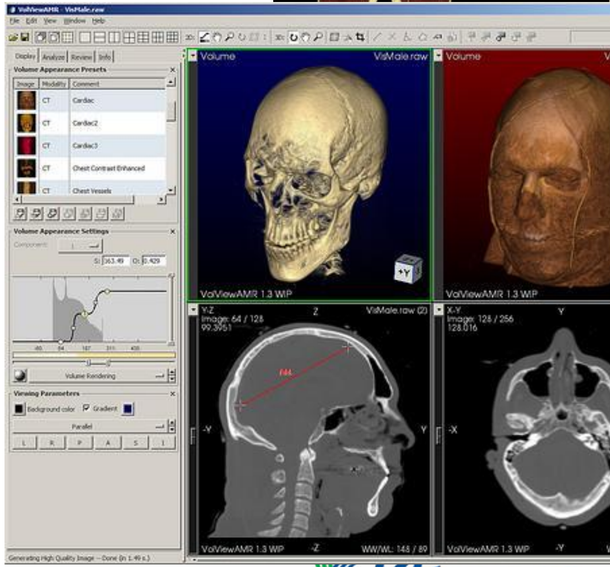
Lister Hill National Center for Biomedical Communications
U.S. National Library of Medicine
8600 Rockville Pike, Bethesda, MD 20894
301-496-4441

NIH...Turning Discovery Into Health

Copyright | Privacy | Accessibility | Freedom of Information Act
Viewers and Players | Contact Us
Page Last Updated: 18 July 2018

Where did CMake come from?

- Kitware was the lead engineering team for the Insight Segmentation and Registration Toolkit (ITK) http://www.itk.org
- Funded by National Library of Medicine (NLM): part of the Visible Human Project
  - Data CT/MR/Slice 1994/1995
  - Code (ITK) 1999

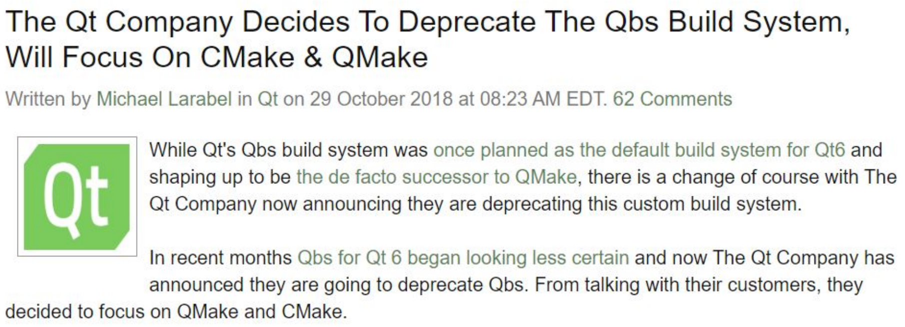


CMake: History

- Other available tools were insufficient: pcmaker (vtk3.2), autoconf, Apache ANT, qmake (Qt), JAM
- CMake provides the combination of native build tool use and platform introspection that none of the others provide.

- CMake Release-1-0 branch created in 2001
- CMake 2.8 released in November, 2009
- CMake 3.0 released June, 2014
- CMake 3.14 released March, 2019
- CMake 3.17 released March, 2020

CMake has high adoption



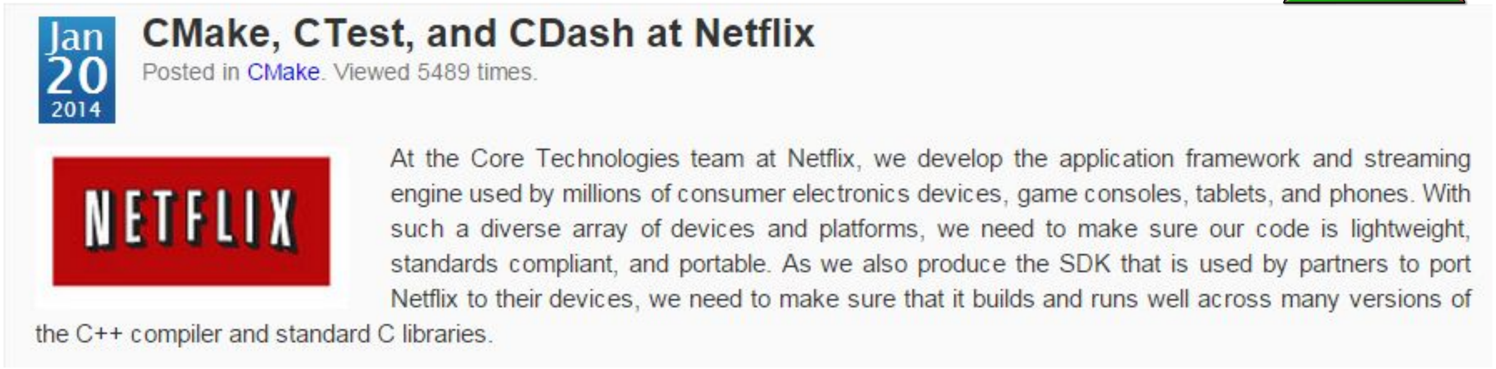
The Qt Company Decides To Deprecate The Qbs Build System, Will Focus On CMake & QMake

Written by Michael Larabel in Qt on 29 October 2018 at 08:23 AM EDT. 62 Comments

While Qt's Qbs build system was once planned as the default build system for Qt6 and shaping up to be the de facto successor to QMake, there is a change of course with The Qt Company now announcing they are deprecating this custom build system.

In recent months Qbs for Qt 6 began looking less certain and now The Qt Company has announced they are going to deprecate Qbs. From talking with their customers, they decided to focus on QMake and CMake.

5 million CMake downloads in 2019. 13809 downloads per day, or a daily rate of 307.3GiB, and yearly over 10TiB



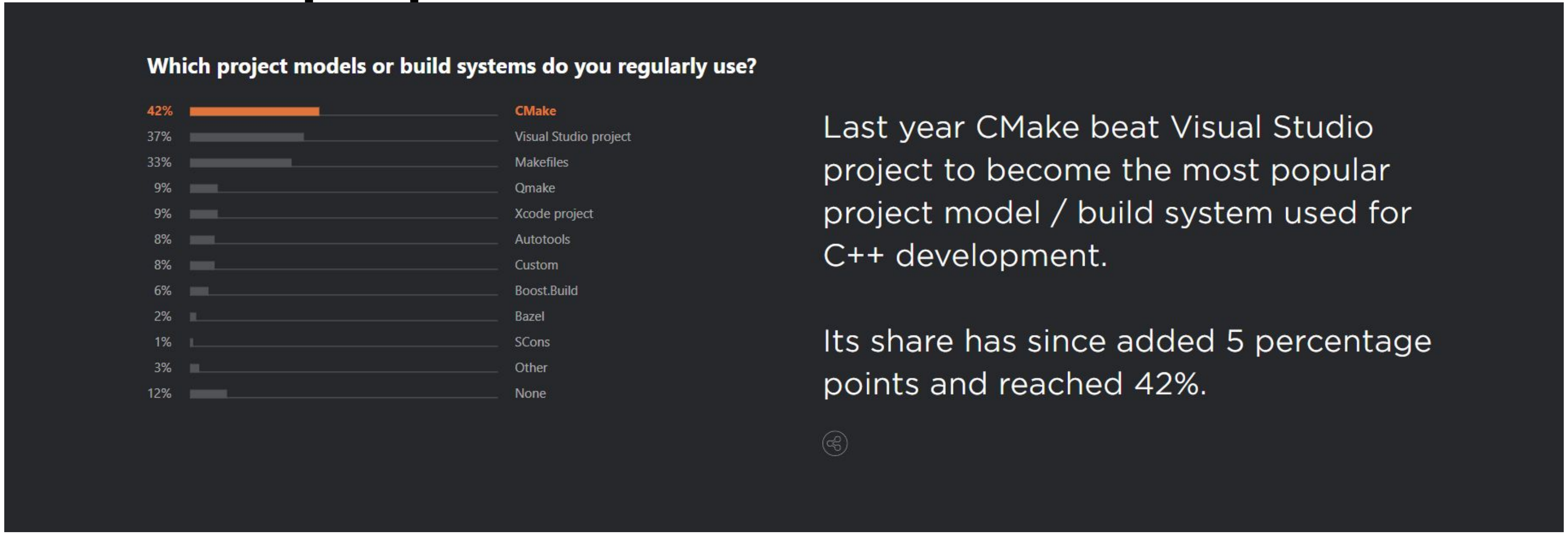
Jan 20 2014

CMake, CTest, and CDash at Netflix

Posted in CMake. Viewed 5489 times.

At the Core Technologies team at Netflix, we develop the application framework and streaming engine used by millions of consumer electronics devices, game consoles, tablets, and phones. With such a diverse array of devices and platforms, we need to make sure our code is lightweight, standards compliant, and portable. As we also produce the SDK that is used by partners to port Netflix to their devices, we need to make sure that it builds and runs well across many versions of the C++ compiler and standard C libraries.

Jetbrains IDE- CMake is the most popular build tool at 42%.


Which project models or build systems do you regularly use?

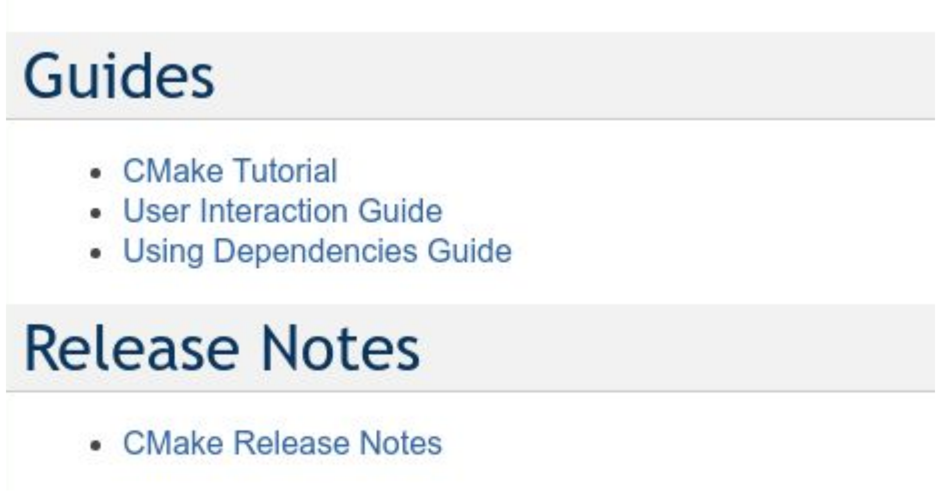
42%	CMake
37%	Visual Studio project
33%	Makefiles
9%	Qmake
9%	Xcode project
8%	Autotools
8%	Custom
6%	Boost.Build
2%	Bazel
1%	SCons
3%	Other
12%	None

Last year CMake beat Visual Studio project to become the most popular project model / build system used for C++ development.

Its share has since added 5 percentage points and reached 42%.

- Job openings requiring CMake experience, March, 2019 Indeed.com, 464 jobs, at Tesla Motors, DCS Corp, Mindsource, Quanergy, ...LinkedIn.com, 486 jobs, at Samsung, Johnson Controls, Apple, Uber, Toyota, Microsoft ...

Updated CMake Tutorial


Guides

- CMake Tutorial
- User Interaction Guide
- Using Dependencies Guide

Release Notes

- CMake Release Notes

A new series of *guides* provided with each CMake release to help with learning and using CMake.

Fully-tested source code embedded in HTML docs

Outsource your build to the CMake developers

- A build system that just works
- A build system that is easy to use cross platform

Typical Project without CMake (curl)

```
$ ls
CHANGES          RELEASE-NOTES  curl-config.in  missing
CMake            acinclude.m4   curl-style.el   mkinstalldirs
CMakeLists.txt   aclocal.m4     depcomp         notes
build       docs       notes~
COPYING          buildconf      include         packages
CVS              buildconf.bat  install-sh      reconf
ChangeLog        compile        lib             sample.emacs
Makefile         config.guess   libcurl.pc.in   src
Makefile.am      config.sub     ltmain.sh       tests
Makefile.in      configure      m4              vc6curl.dsw
README           configure.ac   maketgz

$ ls src/
CMakeLists.txt   Makefile.riscos   curlsrc.dsp   hugehelp.h      version.h
CVS              Makefile.vc6      curlsrc.dsw   macos           writeenv.c
Makefile.Watcom  Makefile.vc8      curlutil.c    main.c          writeenv.h
Makefile.am      config-amigaos.h  curlutil.h    makefile.amiga  writeout.c
Makefile.b32     config-mac.h      getpass.c     makefile.dj     writeout.h
Makefile.in      config-riscos.h   getpass.h     mkhelp.pl
Makefile.inc     config-win32.h    homedir.c     setup.h
Makefile.m32     config.h.in       homedir.h     urlglob.c
Makefile.netware curl.rc           hugehelp.c    urlglob.h
```

Kitware

CMake: Bridging C++ gaps

- Open-source cross-platform build manager using native tools
  - Visual Studio 6, 2003, 2005, 2008, 2010, 2012, 2013, 2015, 2017, 2019
  - Borland make, Nmake, Unix make, MSYS make, MinGW make
  - Ninja
  - Xcode
- IDE Support
  - Code::Blocks
  - CodeLite
  - CLion
  - Eclipse
  - KDevelop
  - Kate
  - Sublime Text
  - Visual Studio Code
  - Emacs and Vim syntax files can be found on the CMake Download page

CMake: Bridging C++ gaps

- Operating Systems:
  - HPUX, IRIX, Linux, MacOS, Windows, QNX, SunOS, and others
- Allows for platform inspection
  - Programs
  - Libraries and Header files
  - Packages
  - Determine hardware specifics like byte order
- Compiler Language Level Support
  - C, C++, ObjC, ObjC++, CSharp, CUDA, Fortran, Swift

CMake: Bridging C++ gaps

- Support for complex custom commands such as:
  - Qt's moc, VTK's wrapping
- Static, shared, object, and module library support
  - including versions .so support
- **Single input** format for all platforms
- Create configured .h files

CMake: Bridging C++ gaps

- Automatic **dependency** generation (C, C++, CUDA, Fortran)
  - build a target in some directory, and everything this target depends on will be up-to-date
- Automatic re-execution of cmake at build time if any cmake input file has changed
- **Parallel** builds
- **User defined** build directory

CMake: Bridging C++ gaps

- Color and progress output for make
- **Graphviz output** for visualizing dependency trees
- **Full cross platform** install() system
- Compute link depend information, and chaining of dependent libraries
- make help, make foo.o, make foo.i, make foo.s

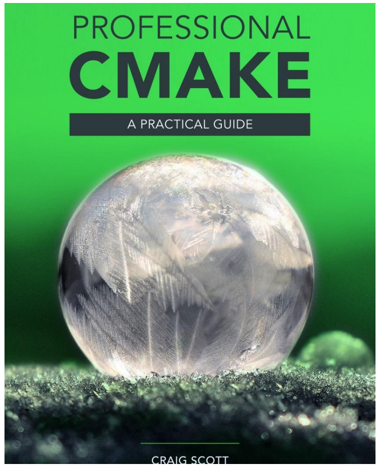
CMake: Bridging C++ gaps

- Advanced RPATH handling
  - Support for chrpath, i.e. changing the RPATH without need to actually link again
- Create OSX library frameworks and application bundles
- Extensive test suite and nightly builds/test on many platforms
- Supports cross compilation

Learning CMake

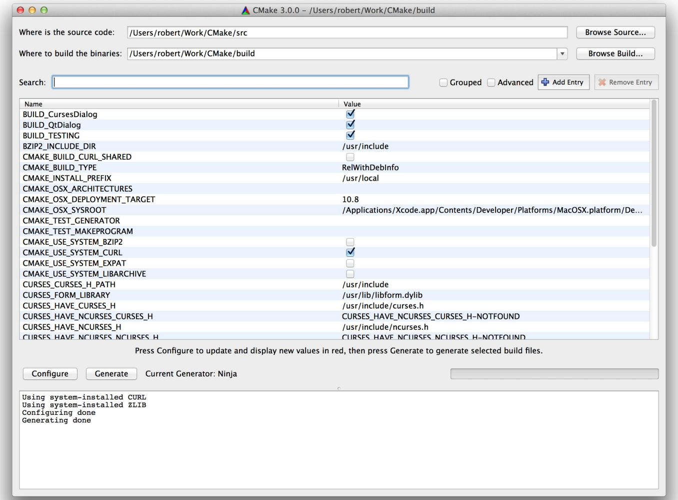
For help or more information see:

- Professional CMake by Craig Scott
- Discourse forum
  - https://discourse.cmake.org
- Documentation
  - https://www.cmake.org/cmake/help/latest/
- Tutorial
  - https://cmake.org/cmake/help/latest/guide/tutorial/index.html



Running CMake

- cmake-gui (the Qt gui)
- ccmake (the terminal cli)
- cmake (non-interactive command line)



Where is the source code: /Users/robert/Work/CMake/src

Where to build the binaries: /Users/robert/Work/CMake/build

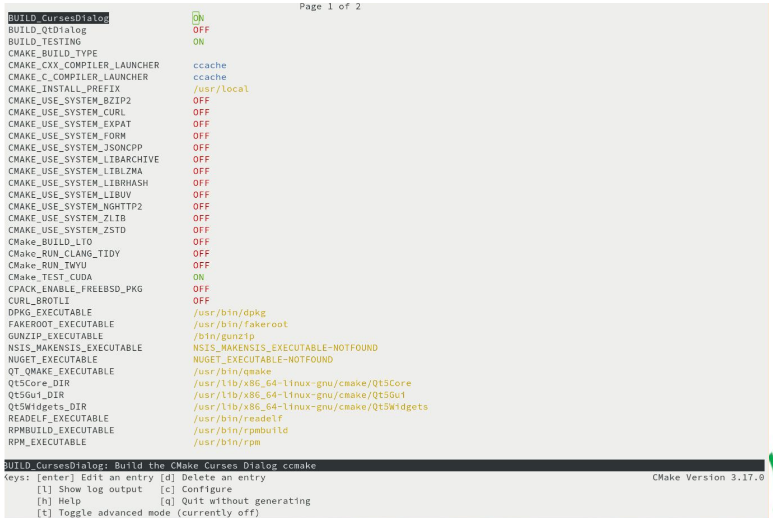
Search:

Name	Value
BUILD_CursesDialog	✓
BUILD_QtDialog	✓
BUILD_TESTING	✓
BZIP2_INCLUDE_DIR	/usr/include
CMAKE_BUILD_CURL_SHARED	☐
CMAKE_BUILD_TYPE	RelWithDebInfo
CMAKE_INSTALL_PREFIX	/usr/local
CMAKE_OSX_ARCHITECTURES	
CMAKE_OSX_DEPLOYMENT_TARGET	10.8
CMAKE_OSX_SYSROOT	/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/De...
CMAKE_TEST_GENERATOR	
CMAKE_TEST_MAKEPROGRAM	
CMAKE_USE_SYSTEM_BZIP2	☐
CMAKE_USE_SYSTEM_CURL	✓
CMAKE_USE_SYSTEM_EXPAT	☐
CMAKE_USE_SYSTEM_LIBARCHIVE	☐
CURSES_CURSES_H_PATH	/usr/include
CURSES_FORM_LIBRARY	/usr/lib/libform.dylib
CURSES_HAVE_CURSES_H	/usr/include/curses.h
CURSES_HAVE_NCURSES_CURSES_H	CURSES_HAVE_NCURSES_CURSES_H-NOTFOUND
CURSES_HAVE_NCURSES_H	/usr/include/ncurses.h
CURSES_HAVE_NCURSES_NCURSES_H	CURSES_HAVE_NCURSES_NCURSES_H-NOTFOUND

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Ninja

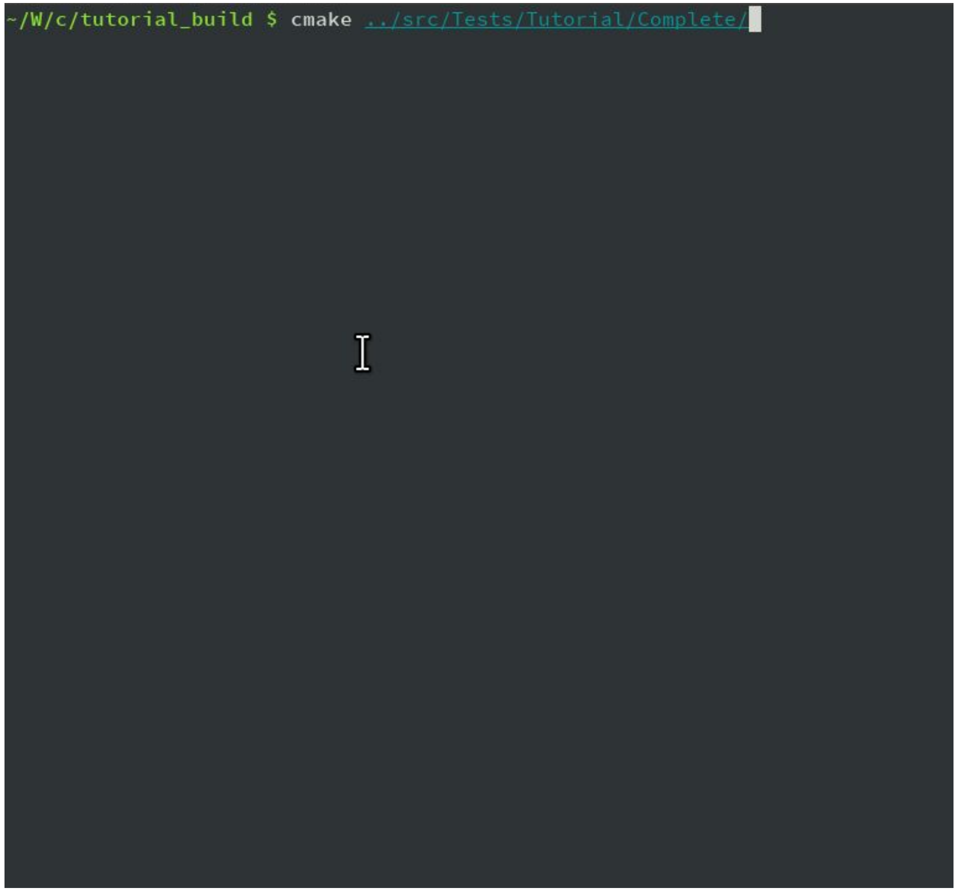
```
Using system-installed CURL
Using system-installed ZLIB
Configuring done
Generating done
```



```
                                                     Page 1 of 2
BUILD_CursesDialog               ON
BUILD_QtDialog                   OFF
BUILD_TESTING                    ON
CMAKE_BUILD_TYPE
CMAKE_CXX_COMPILER_LAUNCHER      ccache
CMAKE_C_COMPILER_LAUNCHER        ccache
CMAKE_INSTALL_PREFIX             /usr/local
CMAKE_USE_SYSTEM_BZIP2           OFF
CMAKE_USE_SYSTEM_CURL            OFF
CMAKE_USE_SYSTEM_EXPAT           OFF
CMAKE_USE_SYSTEM_FORM            OFF
CMAKE_USE_SYSTEM_JSONCPP         OFF
CMAKE_USE_SYSTEM_LIBARCHIVE      OFF
CMAKE_USE_SYSTEM_LIBLZMA         OFF
CMAKE_USE_SYSTEM_LIBRHASH        OFF
CMAKE_USE_SYSTEM_LIBUV           OFF
CMAKE_USE_SYSTEM_NGHTTP2         OFF
CMAKE_USE_SYSTEM_ZLIB            OFF
CMAKE_USE_SYSTEM_ZSTD            OFF
CMake_BUILD_LTO                  OFF
CMake_RUN_CLANG_TIDY             OFF
CMake_RUN_IWYU                   OFF
CMake_TEST_CUDA                  ON
CPACK_ENABLE_FREEBSD_PKG         OFF
CURL_BROTLI                      OFF
DPKG_EXECUTABLE                  /usr/bin/dpkg
FAKEROOT_EXECUTABLE              /usr/bin/fakeroot
GUNZIP_EXECUTABLE                /bin/gunzip
NSIS_MAKENSIS_EXECUTABLE         NSIS_MAKENSIS_EXECUTABLE-NOTFOUND
NUGET_EXECUTABLE                 NUGET_EXECUTABLE-NOTFOUND
QT_QMAKE_EXECUTABLE              /usr/bin/qmake
Qt5Core_DIR                      /usr/lib/x86_64-linux-gnu/cmake/Qt5Core
Qt5Gui_DIR                       /usr/lib/x86_64-linux-gnu/cmake/Qt5Gui
Qt5Widgets_DIR                   /usr/lib/x86_64-linux-gnu/cmake/Qt5Widgets
READELF_EXECUTABLE               /usr/bin/readelf
RPMBUILD_EXECUTABLE              /usr/bin/rpmbuild
RPM_EXECUTABLE                   /usr/bin/rpm

BUILD_CursesDialog: Build the CMake Curses Dialog ccmake
Keys: [enter] Edit an entry [d] Delete an entry                                       CMake Version 3.17.0
      [l] Show log output   [c] Configure
      [h] Help              [q] Quit without generating
      [t] Toggle advanced mode (currently off)
```

CMake Workflow



```
~/W/c/tutorial_build $ cmake ../src/Tests/Tutorial/Complete/
```

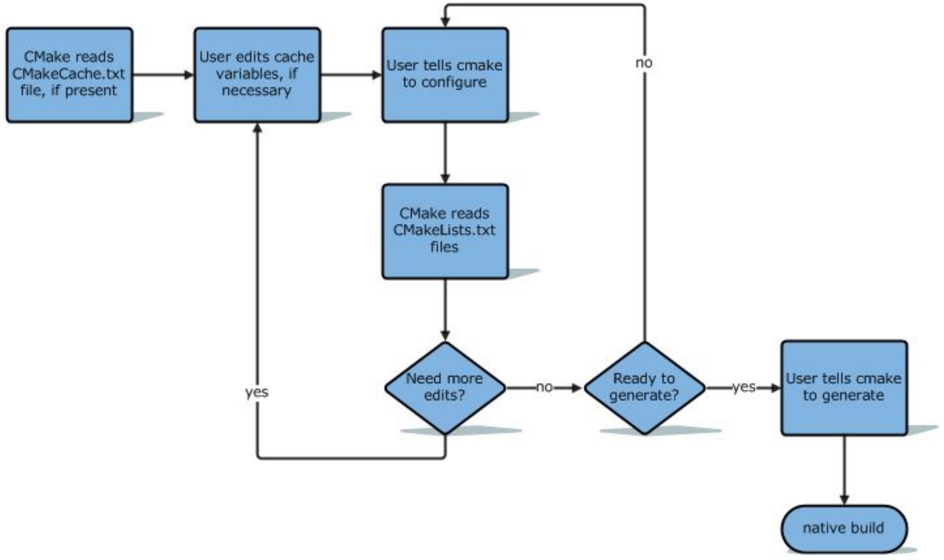
Step 0 - Run CMake

Cooley:
```
soft add +cmake-3.17.3
```
Theta:
```
module load cmake/3.16.2
module swap craype-mic-knl craype-broadwell
```

Exercise!

Run `cmake --version` from the command line

Run ccmake or cmake-gui



CMake reads CMakeCache.txt file, if present
User edits cache variables, if necessary
User tells cmake to configure
no
CMake reads CMakeLists.txt files
yes
Need more edits?
no
Ready to generate?
yes
User tells cmake to generate
native build

Basic CMake Syntax

The CMake language consists of:
Commands, Variables, and Comments

```
cmake_minimum_required(VERSION 3.8)
project(main)
# require C++11 and don’t decay down to 98
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

add_subdirectory(MathFunctions)
add_executable(main main.cxx)
target_link_libraries(main MathFunctions)
```

Basic CMake Syntax: examples

Commands may span multiple lines, but the command name and the opening parenthesis must be on the same line

```
set(sources
  CellSet.cxx
  CellSetExplicit.cxx
  CellSetStructured.cxx
  CoordinateSystem.cxx
  Field.cxx
  ImplicitFunction.cxx
)
```

CMake Commands

- Commands are documented online and within cmake itself:
  - https://cmake.org/cmake/help/latest/manual/cmake-commands.7.html
- Commands may be uppercase or lowercase.

```
add_executable(main main.cxx)
ADD_EXECUTABLE(main main.cxx)
```

Prefer Lowercase

Historical note: commands used to be all uppercase, earning CMake the affectionate nickname "SCREAMMake" Thankfully, those days are over!

CMake Commands (Arguments)

- Command arguments are space separated
- Arguments are case sensitive

```
add_executable(main WIN32 main.cxx)
```

- Quoted arguments
  - A quoted argument is always exactly one value.
  - An unquoted argument splits on semicolons and removes empty arguments.

CMake Variables

- Variable names are case sensitive

```
set(LEAF valueA)
set(leaf valueB)
```

  Creates two distinct variables named “LEAF” and “leaf”

- Names can only contain alpha-numerics and underscores
  - [A-Za-z0-9_]
- CMake uses many variables that begin with “CMAKE_” – avoid this naming convention (and establish your own) for CMake variables specific to your project

CMake Variables

CMake variables are strings

```
set(name myexe)

# These two statements are equivalent to each other.
set(srcs src1.cxx src2.cxx src3.cxx)
set(srcs "src1.cxx;src2.cxx;src3.cxx")
```

Kitware

CMake Variables

Special syntax for setting environment

use ENV and curly braces { }

```
set(ENV{PATH} <value>...)
```

CMake Variables

Variable dereferencing syntax: ${VAR}

```
message(STATUS "CMAKE_SOURCE_DIR='${CMAKE_SOURCE_DIR}'")
set(my_dir "${CMAKE_CURRENT_SOURCE_DIR}/my_dir")
message(STATUS "my_dir='${my_dir}'")
```

Environment Variable dereferencing syntax: $ENV{VAR}

```
set(my_path "$ENV{PATH}")
message(STATUS "my_path='${my_path}'")
```

CMake Variables

Escaping

- \ is the escape character used in CMake
- You can also use long brackets

```
set(classic_str "* here be \"dragons\" *")
set(long_brackets [=[* here be "dragons" *]=])
message(STATUS ${classic_str})
message(STATUS ${long_brackets})
```

Both of these print: * here be "dragons" *

Configuring Header Files

- Can put build parameters into a header file instead of passing them on the command line

```
configure_file(
  ${PROJECT_SOURCE_DIR}/projectConfig.h.in
  ${PROJECT_BINARY_DIR}/projectConfig.h
)
```

Configuring Header Files

- `#define VARIABLE @VARIABLE@`

```
// C++ source file
#define VARIABLE @VARIABLE@
#ifdef VARIABLE
  // will be hit when the CMake variable doesn't exist
#endif
```

Configuring Header Files

- **`#cmakedefine VARIABLE`**

```
// C++ source file
#cmakedefine @VARIABLE@
#ifdef VARIABLE
  // will not-be hit when the CMake
  // variable doesn't exist
#endif
```

Requiring a CMake Version

- First line of the top level CMakeLists.txt should always be **`cmake_minimum_required`**

```
cmake_minimum_required(VERSION 3.9)
project(Example LANGUAGES C CXX CUDA)
```

- Allows projects to require a given version of CMake
- Allows CMake to be backwards compatible

project() command

- Necessary for the top-level CMake. Should be set after the **`cmake_minimum_required`** command

```
cmake_minimum_required(VERSION 3.9)
project(Example LANGUAGES C CXX CUDA)
```

- VERSION: sets the PROJECT_VERSION_MAJOR/MINOR/TWEAK
- DESCRIPTION: sets the PROJECT_DESCRIPTION variable
- LANGUAGES:
  - C, CXX, FORTRAN, CSharp, CUDA, ASM
  - Default is C and CXX if not defined

Kitware

CMake Language Standards

- CMake offers a few different ways to specify which version of a language should be used.

```
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
```

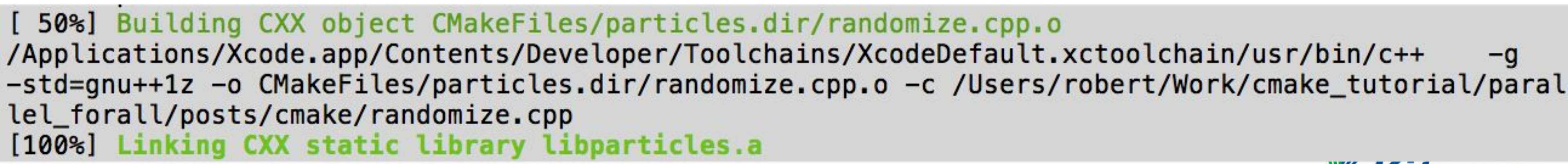
```
[ 50%] Building CXX object main.cxx.o
/usr/bin/c++  -std=gnu++11 -o main.cxx.o -c main.cxx
```

- CMAKE_CXX_EXTENSIONS controls if compiler specific extensions are enabled

CMake 3.8: meta-features

- Request compiler modes for specific language standard levels
  - cxx_std_11, cxx_std_14, cxx_std_17
  - Works with Clang, GCC, MSVC, Intel, Cray, PGI, XL
- These should be used instead of features like cxx_auto_type

```
# Request that particles be built with -std=c++17
# As this is a public compile feature anything that links to particles
# will also build with -std=c++17
target_compile_features(particles PUBLIC cxx_std_17)
```



```
[ 50%] Building CXX object CMakeFiles/particles.dir/randomize.cpp.o
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++    -g
-std=gnu++1z -o CMakeFiles/particles.dir/randomize.cpp.o -c /Users/robert/Work/cmake_tutorial/paral
lel_forall/posts/cmake/randomize.cpp
[100%] Linking CXX static library libparticles.a
```

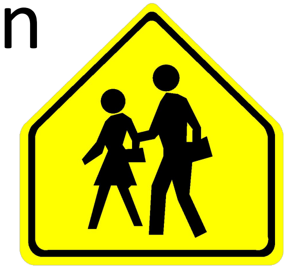
Step 0.5 - CMake and Compiler Selection

Cooley:

```
soft add +cmake-3.17.3
```

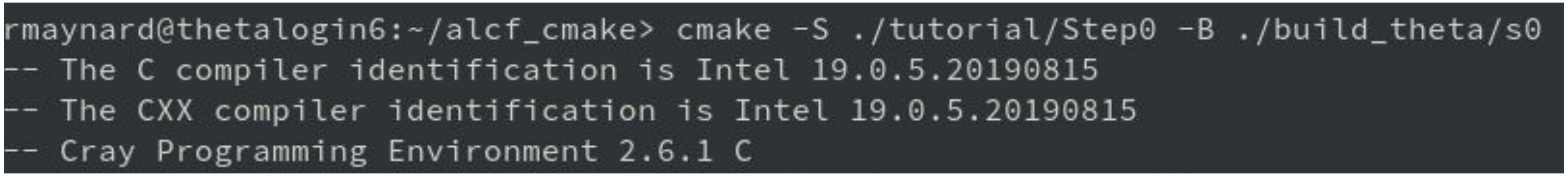
Theta:

```
module load cmake/3.16.2
module swap craype-mic-knl craype-broadwell
```

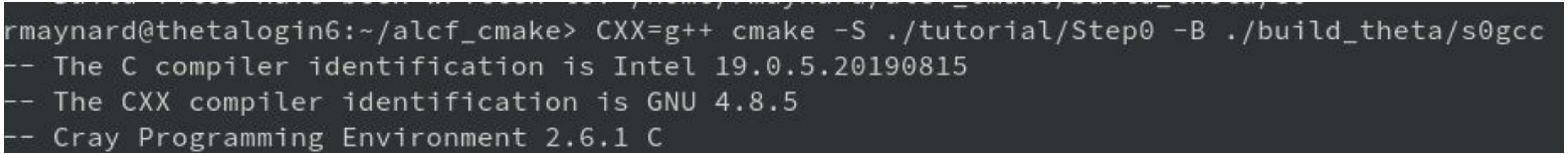


- CMake caches the compiler for a build directory on the first invocation.
- CMake compiler detection has the following preference
  - env variables (CC, CXX)
  - cc and cxx path entries
  - gcc and g++ path entries
- Try the following on cooley / theta
  - soft add +gcc-7.1.0 [cooley only]
  - cmake -S Step0 -B Step0CXXBuildDir
  - CXX=g++ cmake -S Step0 -B Step0GCCBuildDir

Step 0.5 - CMake and Compiler Selection

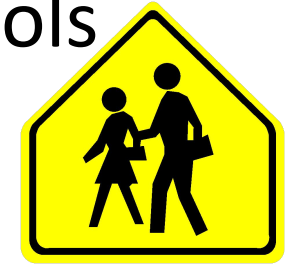


```
rmaynard@thetalogin6:~/alcf_cmake> cmake -S ./tutorial/Step0 -B ./build_theta/s0
-- The C compiler identification is Intel 19.0.5.20190815
-- The CXX compiler identification is Intel 19.0.5.20190815
-- Cray Programming Environment 2.6.1 C
```



```
rmaynard@thetalogin6:~/alcf_cmake> CXX=g++ cmake -S ./tutorial/Step0 -B ./build_theta/s0gcc
-- The C compiler identification is Intel 19.0.5.20190815
-- The CXX compiler identification is GNU 4.8.5
-- Cray Programming Environment 2.6.1 C
```

Step 1 - Configure a File and C++11 Controls



Cooley:

```
   soft add +cmake-3.17.3
```

Theta:

```
   module load cmake/3.16.2
   module swap craype-mic-knl craype-broadwell
```

- Follow the instructions in Step1
- Run cmake or cmake-gui to configure the project and then build it with your chosen build tool
- `cd` to the directory where Tutorial was built (likely the make directory or a Debug or Release build configuration subdirectory) and run these commands:
  - Tutorial 4294967296
  - Tutorial 10
  - Tutorial

Flow control (if)

CMake `if` command supports a wide range of expressions

```
if(my_var)
  set(result ${my_var})
endif()
if(NOT my_var)
if(my_var AND my_var2)
if(my_var OR my_var2)
if(my_var MATCHES regexp)
if(TARGET target)
if(EXISTS file)
if(my_var LESS my_var2)
if(my_ver VERSION_EQUAL "2.0.2")
```

Kitware

Flow control (if): FALSE values

- The following values are all equivalent to “FALSE” in a CMake if command:
  - “” (the empty string)
  - `OFF`
  - `0`
  - `NO`
  - `FALSE`
  - `N`
  - “`NOTFOUND`” exactly or ends in “`-NOTFOUND`”
  - `IGNORE`

Kitware

Flow control: loops

```
foreach(F IN ITEMS a b c)
  message(${F})
endforeach()

set(items a b c)
foreach(F IN LISTS items)
  message(${F})
endforeach()
```

```
while(MY_VAR)
  message(${MY_VAR})
  set(MY_VAR FALSE)
endwhile()
```

Kitware

add_subdirectory and variable scope

- The add_subdirectory command allows a project to be separated into directories
- Variable values are inherited by CMakeLists.txt files in sub directories
  - TopDir

```
set(MY_VAR 1)
add_subdirectory(Dir1)
```

  - TopDir/Dir1

```
Dir1/CMakeLists.txt  -> MY_VAR is 1
```

Function

```
function(showcase_args myarg)
  message("myarg: ${myarg}")
  message("ARGV0: ${ARGV0}")
  message("contents of myarg: ${${myarg}}")
  message("extra arguments: ${ARGN}")
  message("# of arguments: ${ARGC}")
endfunction()

set(items a b c)
showcase_args(items)
```

```
myarg: items
ARGV0: items
contents of myarg: a;b;c
extra arguments:
# of arguments: 1
```

- ARGC – number of arguments passed
- ARGV0, ARGV1, … - actual parameters passed in
- ARGV – list of all arguments
- ARGN – list of all arguments beyond the last formal parameter

Function

- Dynamically scoped, so any variables set are local to the function
- Use **set(...PARENT_SCOPE)** to set a variable in the calling

```
function(showcase_args myarg)
  set("${myarg}" ${${myarg}} d e f PARENT_SCOPE)
endfunction()
set(items a b c)
showcase_args(items)
message("items: ${items}")
```

```
items: a;b;c;d;e;f
```

Macro

- Not dynamically scoped, so all variables leak into the calling scope

```
macro(showcase_args arg1)
  set("${arg1}" ${${arg1}} d e f)
endmacro()
set(items a b c)
showcase_args(items)
message("items: ${items}")
```

```
items: a;b;c;d;e;f
```

CMake Common Command Review

```
cmake_minimum_required(VERSION 3.9)
project(Example LANGUAGES C CXX)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)

set(srcs
  Field.cxx
  CellSet.cxx
  CellSetExplicit.cxx
  ImplicitFunctions.cxx)
add_library(simplelib ${srcs})
add_executable(example main.cxx)
target_link_libraries(example
                      PRIVATE simplelib)
add_subdirectory(tests)
```

CMake Commands

- All commands
  - cmake --help-command-list
  - cmake --help-command command_name
  - https://cmake.org/cmake/help/latest/manual/cmake-commands.7.html

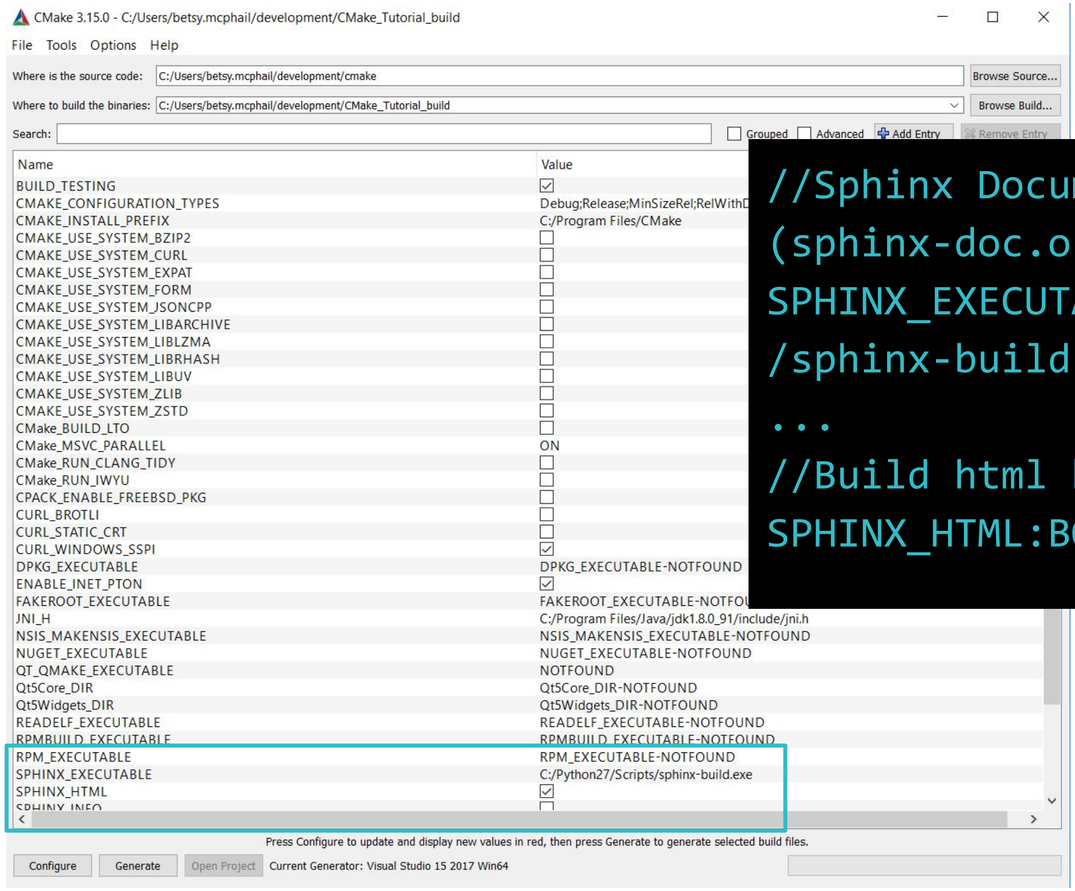
CMakeCache.txt

- Stores optional choices and provides a project global variable repository
- Variables are kept from run to run
- Located in the top directory of the build tree
- A set of entries like this:
  - KEY:TYPE=VALUE

- Valid types:
  - BOOL
  - STRING
  - PATH
  - FILEPATH
  - INTERNAL

(these are only used by cmake-gui and ccmake to display the appropriate type of edit widget)

CMake Cache


CMake 3.15.0 - C:/Users/betsy.mcphail/development/CMake_Tutorial_build

Where is the source code: C:/Users/betsy.mcphail/development/cmake

Where to build the binaries: C:/Users/betsy.mcphail/development/CMake_Tutorial_build

Name	Value
BUILD_TESTING	☑
CMAKE_CONFIGURATION_TYPES	Debug;Release;MinSizeRel;RelWithD
CMAKE_INSTALL_PREFIX	C:/Program Files/CMake
CMAKE_USE_SYSTEM_BZIP2	☐
CMAKE_USE_SYSTEM_CURL	☐
CMAKE_USE_SYSTEM_EXPAT	☐
CMAKE_USE_SYSTEM_FORM	☐
CMAKE_USE_SYSTEM_JSONCPP	☐
CMAKE_USE_SYSTEM_LIBARCHIVE	☐
CMAKE_USE_SYSTEM_LIBLZMA	☐
CMAKE_USE_SYSTEM_LIBRHASH	☐
CMAKE_USE_SYSTEM_LIBUV	☐
CMAKE_USE_SYSTEM_ZLIB	☐
CMAKE_USE_SYSTEM_ZSTD	☐
CMake_BUILD_LTO	☐
CMake_MSVC_PARALLEL	ON
CMake_RUN_CLANG_TIDY	☐
CMake_RUN_IWYU	☐
CPACK_ENABLE_FREEBSD_PKG	☐
CURL_BROTLI	☐
CURL_STATIC_CRT	☐
CURL_WINDOWS_SSPI	☑
DPKG_EXECUTABLE	DPKG_EXECUTABLE-NOTFOUND
ENABLE_INET_PTON	☑
FAKEROOT_EXECUTABLE	FAKEROOT_EXECUTABLE-NOTFOU
JNI_H	C:/Program Files/Java/jdk1.8.0_91/include/jni.h
NSIS_MAKENSIS_EXECUTABLE	NSIS_MAKENSIS_EXECUTABLE-NOTFOUND
NUGET_EXECUTABLE	NUGET_EXECUTABLE-NOTFOUND
QT_QMAKE_EXECUTABLE	NOTFOUND
Qt5Core_DIR	Qt5Core_DIR-NOTFOUND
Qt5Widgets_DIR	Qt5Widgets_DIR-NOTFOUND
READELF_EXECUTABLE	READELF_EXECUTABLE-NOTFOUND
RPMBUILD_EXECUTABLE	RPMBUILD_EXECUTABLE-NOTFOUND
RPM_EXECUTABLE	RPM_EXECUTABLE-NOTFOUND
SPHINX_EXECUTABLE	C:/Python27/Scripts/sphinx-build.exe
SPHINX_HTML	☑
SPHINX_INFO	☐

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 15 2017 Win64

```
//Sphinx Documentation Builder
(sphinx-doc.org)
SPHINX_EXECUTABLE:FILEPATH=C:/Python27/Scripts
/sphinx-build.exe
...
//Build html help with Sphinx
SPHINX_HTML:BOOL=ON
```

Variables and the Cache

- Use **option** command or **set** command with **CACHE** keyword

```
option(MY_VAR "only bool var" TRUE)
set(MY_VAR TRUE CACHE BOOL "bool var")
```

Variables and the Cache

Dereferences look first for a local variable, then in the cache if there is no local definition for a variable

Local variables hide cache variables

Variables and the Cache

Which one is the better option?

```
set(CMAKE_CXX_FLAGS "-Wall")
```

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
```

```
set(CMAKE_CXX_FLAGS "..." CACHE STRING "" FORCE)
```

Variables and the Cache

```
set(CMAKE_CXX_FLAGS "-Wall")
# Clears any users CXX FLAGS! :(
```

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
```

```
set(CMAKE_CXX_FLAGS "..." CACHE STRING "" FORCE)
# Will keep appending each time you reconfigure
# the project
```

`mark_as_advanced`

- Advanced variables are not displayed in the cache editors by default
- Allows for complicated, seldom changed options to be hidden from users

- Cache variables of the INTERNAL type are never shown in cache editors

CMake Special Variables

- cmake --help-variables or online docs
- User settable variables
  - BUILD_SHARED_LIBS
  - CMAKE_INSTALL_PREFIX
  - CMAKE_CXX_FLAGS / CMAKE_<LANG>_FLAGS
- CMake pre-defined variables (should not be set by user code)
  - WIN32, UNIX, APPLE, CMAKE_VERSION
  - CMAKE_SOURCE_DIR, CMAKE_BINARY_DIR
  - PROJECT_NAME
  - PROJECT_SOURCE_DIR, PROJECT_BINARY_DIR

Passing options to the compiler

```
add_compile_definitions(-Drevision=2902)
target_compile_definitions(Tutorial PRIVATE revision=2902)
target_compile_options(Tutorial PRIVATE -march=native)
```

- alternative to configuring header files
- targets, directories, and source files have the properties: `COMPILE_OPTIONS, COMPILE_DEFINITIONS`
  Which can be also be used instead of the target commands.

Build Configurations

- With Makefile generators(Makefile, Ninja):
  - CMAKE_BUILD_TYPE:STRING=Release
  - known values are: Debug, Release, MinSizeRel, RelWithDebInfo

- To build multiple configurations with a Makefile generator, use multiple build trees

Build Configurations

- With multi-config generators (Visual Studio / Xcode):
  - CMAKE_CONFIGURATION_TYPES
    - = list of valid values for config types
  - All binaries go into config subdirectory

```
${CMAKE_CURRENT_BINARY_DIR}/bin/Debug/
${CMAKE_CURRENT_BINARY_DIR}/bin/Release/
```

Build Configurations

- To set per configuration information:
  - per target use `$<CONFIG>`

```
target_compile_definitions(Tutorial PRIVATE
  $<$<CONFIG:DEBUG>:ENABLE_DEBUG_CHECKS>
)
```

  - globally use `CMAKE_CXX_FLAGS_<CONFIG>`

Build Configurations

- To get the current configuration type from multi-conf:
  - Generate Time:
    - `$<CONFIG>`
  - Build-time (deprecated):
    - `${CMAKE_CFG_INTDIR}`
  - In source file
    - `CMAKE_INTDIR` which is defined automatically

ADDING LIBRARIES

CMake Libraries

- Use the `add_library` command to build libraries.

```
option(BUILD_SHARED_LIBS
  "controls add_library default type" ON)
add_library(root root.cxx)
add_library(trunk STATIC trunk.cxx)
add_library(leaf SHARED leaf.cxx)
```

- STATIC => .a or .lib archive
- SHARED => .so, .dylib, or .dll dynamic library

CMake Libraries

```
option(BUILD_SHARED_LIBS
  "controls add_library default type" ON)
add_library(root root.cxx)
add_library(trunk STATIC trunk.cxx)
add_library(leaf SHARED leaf.cxx)
```

- SHARED will work on Unix where supported.
- SHARED on Windows requires code changes or *.def files to export symbols. CMake makes this easier.

Linking to Libraries

- `target_link_libraries` is how you specify what libraries a target requires.

```
add_library(root SHARED root.cxx)
add_library(trunk SHARED trunk.cxx)
add_library(leaf SHARED leaf.cxx)

target_link_libraries(trunk root)
target_link_libraries(leaf trunk)
```

Linking to Libraries

- By default `target_link_libraries` is transitive

```
[100%] Linking CXX shared library libleaf.so
/usr/bin/c++ -fPIC -shared -Wl,-soname,libleaf.so
             -o libleaf.so leaf.cxx.o libtrunk.so libroot.so
[100%] Built target leaf
```

MODULE Libraries

- Very similar to SHARED libraries but are not linked into other targets but can be loaded dynamically at runtime using dlopen-like functionality

```
add_library(parasite MODULE eat_leaf.cxx)
```

OBJECT Libraries

- Generate the object files but does not construct an archive or library
  - Can be installed [3.9]
  - Can be exported/imported [3.9]
  - Can be consumed with target_link_libraries [3.12]
  - Can have transitive information [3.12]

OBJECT Libraries

```
add_library(root  OBJECT root.cxx)
add_library(trunk OBJECT trunk.cxx)
add_library(leaf  SHARED leaf.cxx)
target_link_libraries(leaf root trunk)
```

```
[100%] Linking CXX shared library libleaf.so
/usr/bin/c++ -fPIC   -shared -Wl,-soname,libleaf.so
             -o libleaf.so leaf.cxx.o root.cxx.o trunk.cxx.o
```

OBJECT Libraries

```
add_library(root  OBJECT root.cxx)
add_library(trunk OBJECT trunk.cxx)
add_library(leaf  SHARED
            leaf.cxx
            $<TARGET_OBJECTS:root>
            $<TARGET_OBJECTS:trunk>)
```

```
[100%] Linking CXX shared library libleaf.so
/usr/bin/c++ -fPIC   -shared -Wl,-soname,libleaf.so
             -o libleaf.so leaf.cxx.o root.cxx.o trunk.cxx.o
```

OBJECT Libraries Caveats

- CMake 3.9 added ability for OBJECT libraries to be:
  - Installed / Exported / Imported
  - $<TARGET_OBJECTS> to be used in more generator expression locations

OBJECT Libraries Caveats

- CMake 3.12 added ability to link to OBJECT libraries:
  - Will behave like any other library for propagation
  - Anything that links to an OBJECT library will have the objects embedded into it.

Step 2- Adding a library

Cooley:

```
   soft add +cmake-3.17.3
```

Theta:

```
   module load cmake/3.16.2
   module swap craype-mic-knl craype-broadwell
```

- Follow the directions in Step2
- Run cmake or cmake-gui to configure the project and then build it with your chosen build tool
- Run the built Tutorial executable
- Which function gives better results,
  - Step1's sqrt or Step2's mysqrt?

Exercise!

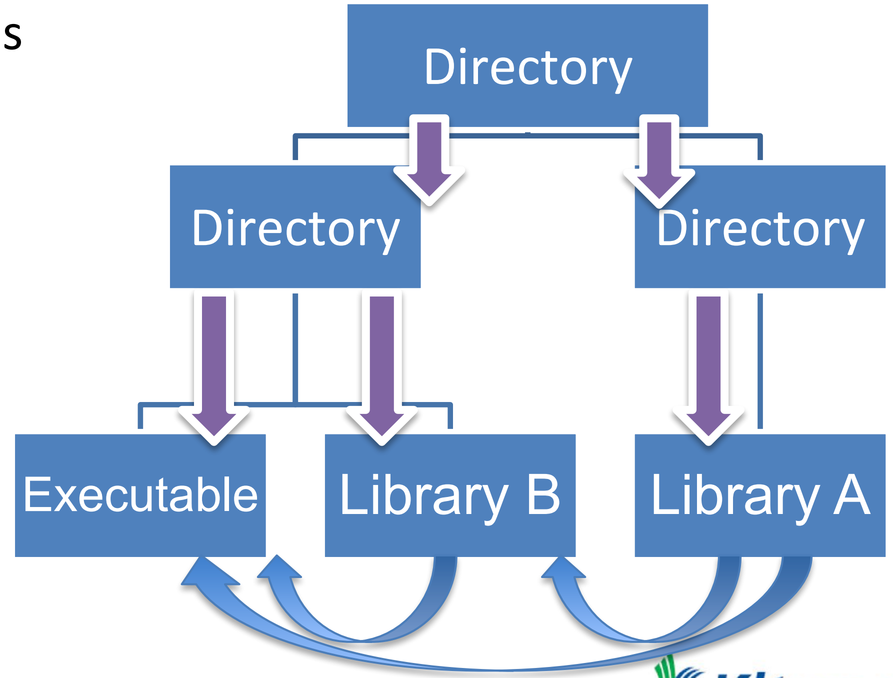
Kitware

Modern CMake

USAGE REQUIREMENTS

Before Usage Requirements

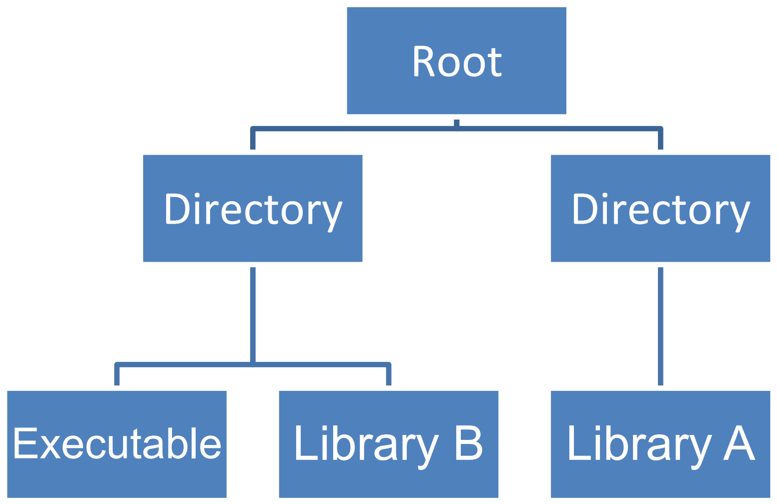
- Before Usage Requirements existed we used directory scoped commands such as:
  - `include_directories`
  - `compile_definitions`
  - `compile_options`
- Consumers have to know:
  - Does the dependency generate build tree files
  - Does the dependency use any new external package

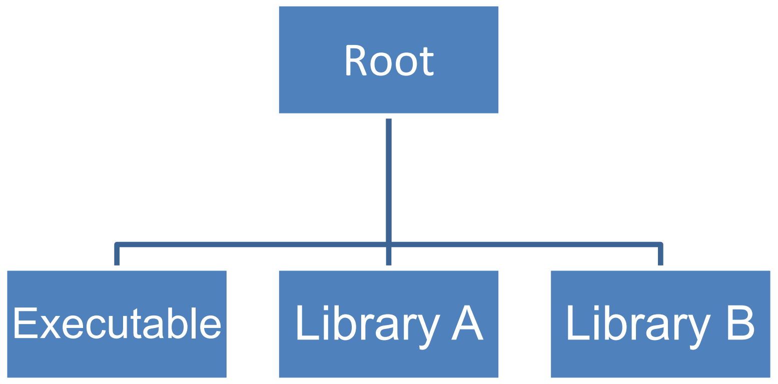


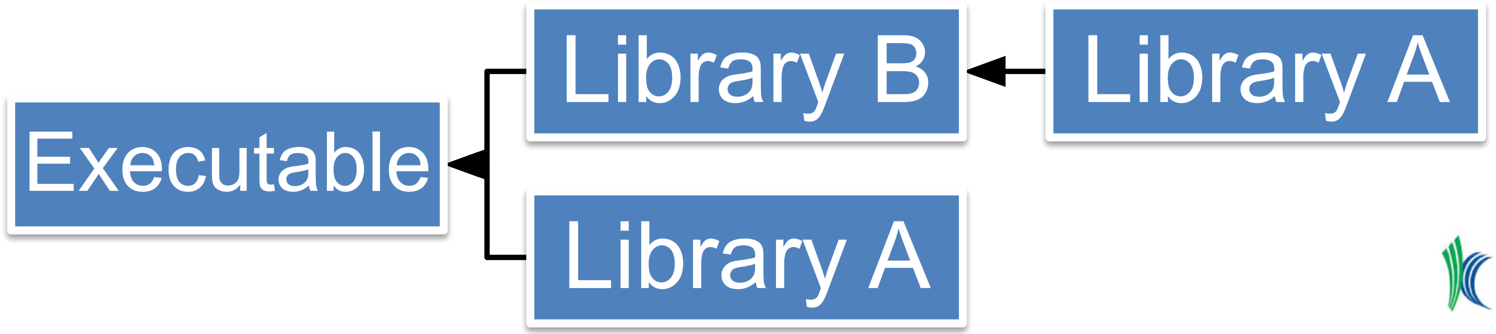
Modern CMake / Usage Requirements

- Modern CMake goal is to have each target fully describe how to properly use it
- No difference between using internal and external generated targets

Modern CMake







Usage Requirements

- `target_link_libraries` is the foundation for usage requirements
- This foundation is formed by
  - PUBLIC
  - PRIVATE
  - INTERFACE

```
target_link_libraries(trunk PRIVATE root)
target_link_libraries(leaf PUBLIC trunk)
```

Usage Requirements

```
target_link_libraries(trunk root)
target_link_libraries(leaf trunk)
```

```
/usr/bin/c++ -fPIC -shared -Wl,-soname,libleaf.so
             -o libleaf.so leaf.cxx.o libtrunk.so libroot.so
```

```
target_link_libraries(trunk PRIVATE root)
target_link_libraries(leaf PUBLIC trunk)
```

```
/usr/bin/c++ -fPIC -shared -Wl,-soname,libleaf.so
             -o libleaf.so leaf.cxx.o libtrunk.so
```

TLL (target link libraries)

- TLL can propagate dependencies when using:
  - `target_include_directories`
  - `target_compile_definitions`
  - `target_compile_options`
  - `target_sources`
  - `target_link_options`

target_include_directories

- Propagates include directories

```
target_include_directories(leaf INTERFACE ${zlib_dir})
```

- Anything that links to leaf will automatically have the zlib_dir on the include line

target_compile_options

- Propagates compiler options

```
target_compile_options(trunk PRIVATE -march=native)
```

- Only trunk will be built optimized for the current hardware. Anything that links to trunk will not get this flag

target_compile_definitions

- Propagates pre-processor definitions

```
target_compile_definitions(root PUBLIC "ROOT_VERSION=42")
```

- Root will have ROOT_VERSION defined and anything that links to it will also

INTERFACE Libraries

- An INTERFACE library target does not directly create build output, though it may have properties set on it and it may be installed, exported, and imported.

```
add_library(root INTERFACE)
target_compile_features(root INTERFACE cxx_std_11)
```

Step 3 - Usage Requirements for Library

Cooley:

```
soft add +cmake-3.17.3
```

Theta:

```
module load cmake/3.16.2
module swap craype-mic-knl craype-broadwell
```

- Follow the directions in Step3 of the Tutorial
- Run cmake or cmake-gui to configure the project and then build it with your chosen build tool

INSTALL RULES

Install Rules

- Specify rules to run at install time
- Can install targets, files, or directories
- Provides default install locations

```
add_library(leaf SHARED leaf.cxx)
install(TARGETS root trunk leaf parasite)
```

Install Targets

```
add_library(leaf SHARED leaf.cxx)
install(TARGETS root trunk leaf parasite)
```

```
add_library(leaf SHARED leaf.cxx)
install(TARGETS root trunk leaf parasite
   ARCHIVE DESTINATION lib
   LIBRARY DESTINATION lib
   RUNTIME DESTINATION bin)
```

Install Targets

```
add_library(leaf SHARED leaf.cxx)
install(TARGETS root trunk leaf parasite)
```

Target Type	GNUInstallDirs Variable	Built-in Default
RUNTIME	${CMAKE_INSTALL_BINDIR}	bin
LIBRARY	${CMAKE_INSTALL_LIBDIR}	lib
ARCHIVE	${CMAKE_INSTALL_LIBDIR}	lib
PRIVATE_HEADER	${CMAKE_INSTALL_INCLUDEDIR}	include
PUBLIC_HEADER	${CMAKE_INSTALL_INCLUDEDIR}	include

Install Files

```
install(FILES
  trunk.h
  leaf.h
  DESTINATION
      ${CMAKE_INSTALL_INCLUDEDIR}/tree
)
```

Install Files

```
add_library(leaf SHARED leaf.cxx)
set_target_properties(leaf PROPERTIES
                      PUBLIC_HEADER leaf.h)

install(TARGETS root trunk leaf parasite
  PUBLIC_HEADER DESTINATION
    ${CMAKE_INSTALL_INCLUDEDIR}/tree
  )
```

Exporting Targets

- Install rules can generate imported targets

```
add_library(parasite STATIC eat_leaf.cxx)
install(TARGETS parasite root trunk leaf
        EXPORT tree-targets)
install(EXPORT tree-targets
        DESTINATION lib/cmake/tree)
```

- Installs library and target import rules
  - `<prefix>/lib/libparasite.a`
  - `<prefix>/lib/cmake/tree/tree-targets.cmake`

CTEST

Testing with CMake

- Testing needs to be enabled by calling `include(CTest)` or `enable_testing()`

```
add_test(NAME testname
         COMMAND exename arg1 arg2 ...)
```

  – Executable should return 0 for a test that passes

- ctest – an executable that is distributed with cmake that can run tests in a project

Running CTest

- Run ctest at the top of a binary directory to run all tests

```
$ ctest
Test project /tmp/example/bin
    Start 1: case1
1/1 Test #1: case1 .............................   Passed    0.00 sec
    Start 2: case2
2/2 Test #2: case2 .............................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.01 sec
```

Running CTest

- **`-j`** option allows you to run tests in parallel
- **`-R`** option allows you to choose a test
- **`-VV`** for more verbose output
- **`--rerun-failed`** to repeat failed tests

- **`ctest --help`** for more information

Test fixtures

- Attach setup and cleanup tasks to a set of tests.
- Setup tests $\longrightarrow$ “main” tests $\longrightarrow$ cleanup tests
- If any setup tests fail the “main” tests will not be run.
- The cleanup tests are always executed, even if some of the setup tests failed.

Test Fixtures example

```
add_test(NAME myDBTest    COMMAND testDb)

add_test(NAME createDB    COMMAND initDB)
add_test(NAME setupUsers  COMMAND userCreation)

add_test(NAME cleanupDB   COMMAND deleteDB)
add_test(NAME testsDone   COMMAND emailResults)
```

Test Fixtures example

```
set_tests_properties(setupUsers PROPERTIES
   DEPENDS createDB)
set_tests_properties(createDB   PROPERTIES
   FIXTURES_SETUP    DB)
set_tests_properties(setupUsers PROPERTIES
   FIXTURES_SETUP    DB)
set_tests_properties(cleanupDB  PROPERTIES
   FIXTURES_CLEANUP  DB)
set_tests_properties(myDBTest   PROPERTIES
   FIXTURES_REQUIRED DB)
```

GoogleTest integration

```
include(GoogleTest)
add_executable(tests tests.cpp)
target_link_libraries(tests GTest::GTest)
```

- gtest_discover_tests: added in CMake 3.10.
  - CMake asks the test executable to list its tests. Finds new tests without rerunning CMake.

```
gtest_discover_tests(tests)
```

GoogleTest integration

```
include(GoogleTest)
add_executable(tests tests.cpp)
target_link_libraries(tests GTest::GTest)
```

- gtest_add_tests: use for CMake ≤ 3.9.
  - Scans source files to finds tests. New tests are only discovered when CMake re-runs.

```
gtest_add_tests(TARGET tests)
```

CTest and multi core tests

- When launching tests that use multi cores it is important to make sure you use the following:

```
set_tests_properties(myTest PROPERTIES
                            PROCESSORS 4)
```

CTest and multi core tests

- PROCESSOR_AFFINITY - when supported ties processes to specific processors

```
set_tests_properties(myTest PROPERTIES
                     PROCESSOR_AFFINITY ON
                     PROCESSORS 4)
```

Test Resource Allocation

- Tests specify the resources they need
- Users specify the resources available on the machine
- CTest keeps track of available resources when running tests in parallel
- New in CMake v3.16

Test Resource Allocation

- Tests using GPU
  - Run tests serially → too slow
  - Run tests in parallel → exhaust available memory → spurious failures
- Tests dictate how much GPU memory they use
- CTest can run tests in parallel without exhausting GPU memory

Test Resource Allocation

- CTest does not directly communicate with GPUs
- It keeps track of abstract resource types and the number of “slots” available

Test Resource Allocation

- For tests, set RESOURCE_GROUPS property

```
set_property(TEST MyTest PROPERTY
RESOURCE_GROUPS "gpus:2" "crypto_chips:1")
```

- For machines, define a Resource Specification JSON file.

```
ctest_test(RESOURCE_SPEC_FILE spec.json)
```

Step 4 - Installing and Testing

- Follow the directions in Step4 of the CMake Tutorial
  - Expect the `make install` to fail when on ANL hardware
- Run cmake or cmake-gui to configure the project and then build it with your chosen build tool
- Build the "install" target (make install, or right click install and choose "Build Project") – verify that the installed Tutorial runs
- cd to the binary directory and run

  "ctest -N" and "ctest -VV"

Exercise!

Kitware

SYSTEM INTROSPECTION

Using Find Modules

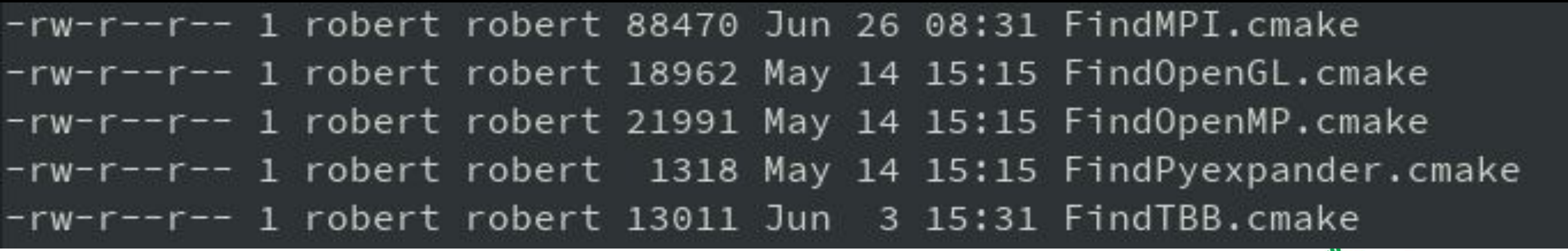
- One of CMake strengths is the `find_package` infrastructure
- CMake provides 150 find modules
  - cmake --help-module-list
  - https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html

```
find_package(PythonInterp)
find_package(TBB REQUIRED)
```

Using Find Modules

- CMake supports each project having custom find modules via CMAKE_MODULE_PATH
  - It is searched before using a module provided by CMake

```
set(CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/CMake)
```



```
-rw-r--r-- 1 robert robert 88470 Jun 26 08:31 FindMPI.cmake
-rw-r--r-- 1 robert robert 18962 May 14 15:15 FindOpenGL.cmake
-rw-r--r-- 1 robert robert 21991 May 14 15:15 FindOpenMP.cmake
-rw-r--r-- 1 robert robert  1318 May 14 15:15 FindPyexpander.cmake
-rw-r--r-- 1 robert robert 13011 Jun  3 15:31 FindTBB.cmake
```

Using Find Modules

- Modern approach: packages construct import targets which combine necessary information into a target.
- Classic CMake: when a package has been found it will define the following:
  - <NAME>_FOUND
  - <NAME>_INCLUDE_DIRS
  - <NAME>_LIBRARIES

Using Config Modules

- `find_package` also supports config modules
- Config modules are generated by the CMake **`export`** command
- Automatically generate import targets with all information, removing the need for consuming projects to write a find module

Using Find Modules

Our library “trunk” needs PNG

```
find_package(PNG REQUIRED)
add_library(trunk SHARED trunk.cxx)
```

Preferred Modern CMake approach:

```
target_link_libraries(trunk PRIVATE PNG::PNG)
```

Historical approach:

```
target_link_libraries(trunk ${PNG_LIBRARIES})
include_directories(trunk ${PNG_INCLUDE_DIRS})
```

Writing find modules

- Exporting your targets is preferred when possible.
- Only write a find module for a project you cannot change.
- Full example here:

  https://cmake.org/cmake/help/latest/manual/cmake-developer.7.html#a-sample-find-module

Writing find modules

Find required files

```
find_path(MyLib_INCLUDE_DIR NAMES mylib.h)
find_library(MyLib_LIBRARY  NAMES mylib)
```

Set version (if available)

```
set(MyLib_VERSION ${MyLib_VERSION})
```

Writing find modules

```
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(MyLib
  FOUND_VAR MyLib_FOUND
  REQUIRED_VARS
    MyLib_LIBRARY
    MyLib_INCLUDE_DIR
  VERSION_VAR MyLib_VERSION)
```

FindPackageHandleStandardArgs

- Makes sure **`REQUIRED_VARS`** are set
- Sets **`MyLib_FOUND`**
- Checks version if **`MyLib_VERSION`** is set and and a version was passed to **`find_package()`**
- Prints status messages indicating if the package was found.

Writing find modules

Provide a way to use the found results:

variables (classic approach)

```
if(MyLib_FOUND)
  set(MyLib_LIBRARIES ${MyLib_LIBRARY})
  set(MyLib_INCLUDE_DIRS ${MyLib_INCLUDE_DIR})
endif()
```

Writing find modules

Provide a way to use the found results:

imported targets (new approach)

```
if(MyLib_FOUND AND NOT TARGET MyLib::MyLib)
  add_library(MyLib::MyLib UNKNOWN IMPORTED)
  set_target_properties(MyLib::MyLib PROPERTIES
    IMPORTED_LOCATION "${MyLib_LIBRARY}"
    INTERFACE_INCLUDE_DIRECTORIES "${MyLib_INCLUDE_DIR}"
  )
endif()
```

Understanding Find Modules Searches

- CMake’s `find_package` uses the following pattern:
  - <PackageName>_ROOT from cmake, then env [3.12]
  - CMAKE_PREFIX_PATH from cmake
  - <PackageName>_DIR from env
  - CMAKE_PREFIX_PATH from env
  - Any path listed in

```
find_package(PNG HINTS /opt/png/)
```

Understanding Find Modules Searches

- PATH from env
- paths found in the CMake User Package Registry
- System paths as defined in the toolchain/platform
    - CMAKE_SYSTEM_PREFIX_PATH
- Any path listed in `find_package(PNG PATHS /opt/png/)`

Find Module Variables

- In general all the search steps can be selectively disabled. For example to disable environment paths:

```
find_package(<package> NO_SYSTEM_ENVIRONMENT_PATH)
```

CMake 3.15+

```
set(CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH FALSE)
find_package(<package>)
```

Directing Find Modules Searches

- The default search order is designed to be most-specific to least-specific for common use cases. When projects need to search an explicit set of places first, they can use the following pattern

```
find_package(<package> PATHS paths... NO_DEFAULT_PATH)
find_package(<package>)
```

Directing Find Modules Searches

- CMAKE_FIND_ROOT_PATH
  - N directories to "re-root" the entire search under.

```
cmake -DCMAKE_FIND_ROOT_PATH=/home/user/pi .
Checking prefix [/home/user/pi/usr/local/]
Checking prefix [/home/user/pi/usr/]
Checking prefix [/home/user/pi/]
```

Direct Find Modules Searches

- CMAKE_PREFIX_PATH
  - Prefix used by find_package as the second search path

```
<prefix>/                                                (W)
<prefix>/(cmake|CMake)/                                  (W)
<prefix>/<name>*/                                        (W)
<prefix>/<name>*/(cmake|CMake)/                          (W)
<prefix>/(lib/<arch>|lib|share)/cmake/<name>*/           (U)
<prefix>/(lib/<arch>|lib|share)/<name>*/                 (U)
<prefix>/(lib/<arch>|lib|share)/<name>*/(cmake|CMake)/   (U)
<prefix>/<name>*/(lib/<arch>|lib|share)/cmake/<name>*/   (W/U)
<prefix>/<name>*/(lib/<arch>|lib|share)/<name>*/         (W/U)
<prefix>/<name>*/(lib/<arch>|lib|share)/<name>*/(cmake|CMake)/  (W/U)
```

Direct Find Modules Searches

- <PackageName>_ROOT
  - Prefix used by find_package to start searching for the given package
  - The package root variables are maintained as a stack so if called from within a find module, root paths from the parent's find module will also be searched after paths for the current package.

Debugging Find Modules [3.17+]

```
find_package(XYZ REQUIRED)
```

```
cmake --find-debug .
…
find_package considered the following paths for XYZ.cmake
  /opt/cmake/.../Modules/FindXYZ.cmake
The file was not found.
find_package considered the following locations for the Config module:
  /home/robert/.local/XYZConfig.cmake
  /home/robert/.local/xyz-config.cmake
  …
  /opt/cmake/XYZConfig.cmake
  /opt/cmake/xyz-config.cmake
The file was not found.
```

Debugging Find Calls [3.17+]

```
set(CMAKE_FIND_DEBUG_MODE 1)
find_library(PNG_LIBRARY_RELEASE
             NAMES png png_static)
```

```
find_library called with the following settings:
    VAR: PNG_LIBRARY_RELEASE
    NAMES: "png"
           "png_static"
    Documentation: Path to a library.
    ...
  find_library considered the following locations:
    /home/robert/.local/bin/(lib)png(\.so|\.a)
    /usr/local/cuda/bin/(lib)png(\.so|\.a)
  …
  The item was not found.
```

Include command

- Allows for including of helper CMake routines that are located in different files
- Use the CMAKE_MODULE_PATH to search for files

```
# Setup default build types
include(VTKmBuildType)
```

Example include file

```
# Set a default build type if none was specified
set(default_build_type "Release")
if(EXISTS "${CMAKE_SOURCE_DIR}/.git")
  set(default_build_type "Debug")
endif()
if(NOT CMAKE_BUILD_TYPE AND NOT CMAKE_CONFIGURATION_TYPES)
  message(STATUS
  "Setting build type to '${default_build_type}' as none was specified.")
  set(CMAKE_BUILD_TYPE "${default_build_type}" CACHE
      STRING "Choose the type of build." FORCE)
  # Set the possible values for build type for cmake-gui
  set_property(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS
    "Debug" "Release" "MinSizeRel" "RelWithDebInfo")
endif()
```

System Introspection

- find_* commands
  - find_file
  - find_library
  - find_package
  - find_path
  - find_program

System Introspection

- try_compile
- Macros to help with common tests
  - CheckIncludeFileCXX.cmake
  - CheckCSourceCompiles.cmake
  - CheckIncludeFiles.cmake
  - CheckCSourceRuns.cmake
  - CheckLibraryExists.cmake
  - CheckCXXCompilerFlag.cmake
  - CheckCXXSourceCompiles.cmake
  - CheckSizeOf.cmake
- try_run, but only if not cross-compiling

System Introspection (cont.)

– CheckCXXSourceRuns.cmake
– CheckStructHasMember.cmake
– CheckSymbolExists.cmake
– CheckTypeSize.cmake
– CheckFunctionExists.cmake
– CheckIncludeFile.cmake
– CheckVariableExists.cmake

Step 5 - System Introspection

Cooley:

```
  soft add +cmake-3.17.3
```

Theta:

```
  module load cmake/3.16.2
  module swap craype-mic-knl craype-broadwell
```

- Follow the directions in Step5 of the CMake Tutorial
- Run cmake or cmake-gui to configure the project and then build it with your chosen build tool
- Run the built Tutorial executable
- Which function gives better results now, Step1’s sqrt or Step5’s mysqrt?

OUTPUT CONTROLS

Build Name Controls

- You can control the build output name of targets
  - Controlled at the target level
  - Can be customized on a per config basis

Suffix Debug Libraries

```
add_library(root  OBJECT root.cxx)
add_library(trunk STATIC trunk.cxx)
add_library(leaf  SHARED leaf.cxx)
set_target_properties(leaf trunk root
             PROPERTIES DEBUG_POSTFIX "d")
```

```
set(CMAKE_DEBUG_POSTFIX "d")
add_library(root  OBJECT root.cxx)
add_library(trunk STATIC trunk.cxx)
add_library(leaf  SHARED leaf.cxx)
```

Static and Shared libraries

```
add_library(leaf_shared SHARED leaf.cxx)
add_library(leaf_static STATIC leaf.cxx)

set_target_properties(leaf_shared
                PROPERTIES OUTPUT_NAME "leaf")
set_target_properties(leaf_static
                PROPERTIES OUTPUT_NAME "leaf")
```

Build Versioning Libraries

library versions on UNIX

```
add_library(leaf SHARED leaf.cxx)
set_target_properties(leaf PROPERTIES VERSION   "1.12"
                                      SOVERSION "8")
```

This results in the following library and symbolic links:

libleaf.so.1.12

libleaf.so.8 -> libleaf.so.1.12

libleaf.so -> libleaf.so.8

Build Location Controls

- You can control the build output location of:
  - Executables
  - Libraries
  - Archives
  - PDB Files

- Can be controlled globally, or at the target level
- Can be customized on a per config basis

Build Output Controls

- Executables = CMAKE_RUNTIME_OUTPUT_DIRECTORY
- Shared Libraries = CMAKE_LIBRARY_OUTPUT_DIRECTORY
- Static Libraries = CMAKE_ARCHIVE_OUTPUT_DIRECTORY
- PDB Files = CMAKE_PDB_OUTPUT_DIRECTORY
- DLL's import .lib = CMAKE_ARCHIVE_OUTPUT_DIRECTORY
- At the Target Level (properties)
  - RUNTIME_OUTPUT_DIRECTORY
  - RUNTIME_OUTPUT_DIRECTORY_<Config>

CUSTOM COMMANDS

Custom commands

- `add_custom_command`
  - Allows you to run arbitrary commands before, during, or after a target is built

- Can be used to generate new files
- Can be used to move or fixup generated or compiled files

Custom commands

- All outputs of the `add_custom_command` need to be explicitly listed.
- `add_custom_command` output must be consumed by a target in the same scope
  - `add_custom_target` can be used for this

Custom command example

```
add_library(CudaPTX OBJECT kernelA.cu kernelB.cu)
set_target_properties(CudaPTX PROPERTIES CUDA_PTX_COMPILATION ON)

set(output_file ${CMAKE_CURRENT_BINARY_DIR}/embedded_objs.h)
add_custom_command(
  OUTPUT "${output_file}"
  COMMAND ${CMAKE_COMMAND}
  "-DBIN_TO_C_COMMAND=${bin_to_c}"
  "-DOBJECTS=$<TARGET_OBJECTS:CudaPTX>"
  "-DOUTPUT=${output_file}"
  -P ${CMAKE_CURRENT_SOURCE_DIR}/bin2c_wrapper.cmake
  VERBATIM
  DEPENDS $<TARGET_OBJECTS:CudaPTX>
  COMMENT "Converting Object files to a C header"
)

add_executable(CudaOnlyExportPTX main.cu ${output_file})
add_dependencies(CudaOnlyExportPTX CudaPTX)
```

CMake Scripts

- cmake –E command
  - Cross platform command line utility for:
  - Copy file, Remove file, Compare and conditionally copy, time, create symlinks, others
- cmake –P script.cmake
  - Cross platform scripting utility
  - Does not generate CMakeCache.txt
  - Ignores commands specific to generating build environment

Step 6 - Custom Command and Generated File

Cooley:

```
   soft add +cmake-3.17.3
```

Theta:

```
   module load cmake/3.16.2
   module swap craype-mic-knl craype-broadwell
```

- Follow the directions in Step6 of the CMake Tutorial
- Run cmake or cmake-gui to configure the project and then build it with your chosen build tool
- Run the built Tutorial executable