Managed by Fermi Research Alliance, LLC for the U.S. Department of Energy Office of Science

Obtaining and building code: Setting up for development

James Amundson

art/LArSoft course

2015-08-03

Overview

- The goal of this lecture is to give you the background to understand Exercise 2: Building and Running Your First Module.
- You will be using git to check out the code.
  - A few git tips will help.
- You will be using cetbuildtools to build the code.
  - Understanding the context for cetbuildtools will help.

git

- git has recently become *the* industry standard for tracking revisions of source code.
  - Plus: There is a wealth of git documentation available on the web.
  - Plus: git has many, many features.
  - Minus: You have to determine which small set of git's features are appropriate for you.
- git is a distributed system.
  - Everyone has his or her own copy of the repository
- All git commands are of the form

  git cmd [options]
  - Use, e.g., "*man git-clone*" to get the man page for "*git clone*"
    - works everywhere, "*man git clone*" works on some systems, but not others.

Fermilab

More git

- *git clone* makes a local copy of a git repository

  git clone http://cdcvs.fnal.gov/projects/art-workbook

  – The original repository becomes “origin”
- *git checkout –b* creates a branch based on *something*

  git checkout –b work origin/August2015

  – creates the branch “work”

  – “work” is based on the branch *August2015*
- *git branch –a* lists all branches
- *git tag –l* lists all tags

  – interface consistency is not git’s strong suit

Fermilab

Now for something completely different: git

- *git pull [remote] [branch]* gets updates from other repositories and merges them into our working branch
  - *pull = fetch + merge*

    git fetch <remote>

    git merge <remote>/<branch>
- *git push [remote [localref:remoteref]* sends updates to remote repository

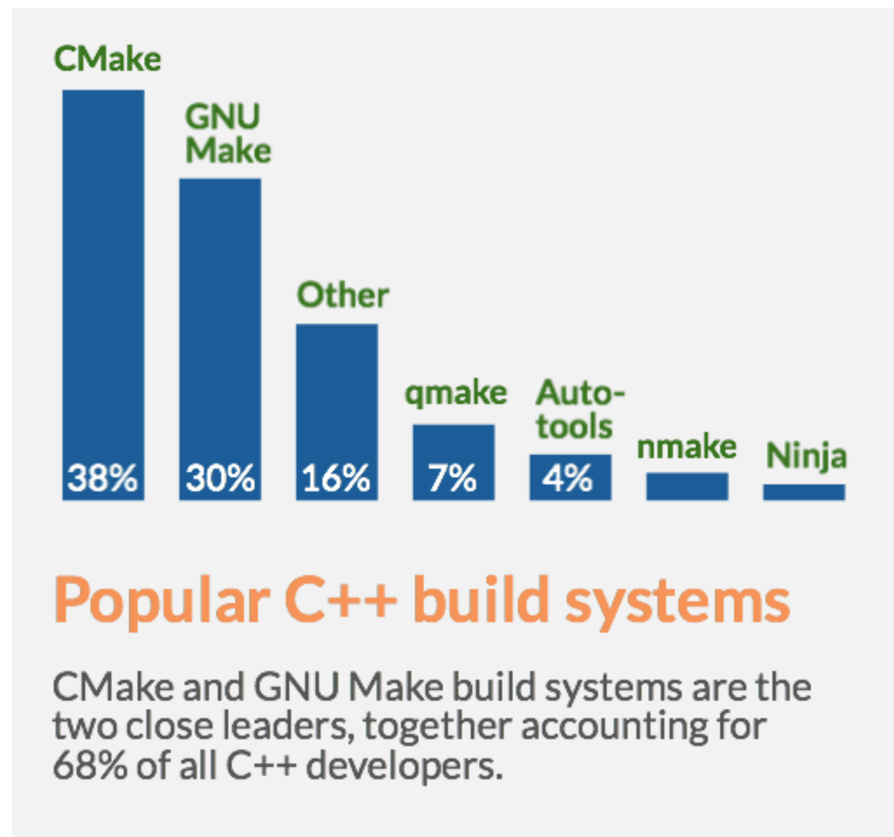
Fermilab

Systems for building code

- Some steps necessary to compile code
  - find external packages
    - header files
    - libraries
  - determine compiler flags
    - optimization settings, etc.
  - compile source files
  - link object files
  - install build products
- In addition, properly supporting incremental builds is crucial
  - Faster is always better…
  - …unless it is too fast
    - Inconsistent builds are easy to create, awful to deal with

Industry standards

- Interesting survey information about contemporary C++ development: http://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/



Fermilab

CMake and cetbuildtools

- CMake is the most popular C++ build tool today.
  - Still, it is not completely dominant.
- CMake creates low-level build scripts
  - Make, Ninja, etc.
    - Run *cmake* once, then use *make* (or *ninja*…); see below.
- CMake can be used with various integrated development environments (IDEs).
  - Outside the scope of this course.
- cetbuildtools is built on top of CMake.
  - https://cdcvs.fnal.gov/redmine/projects/cetbuildtools/wiki
  - Simplifies and enforces consistency.
  - *buildtool* replaces *cmake* and *make* (or *ninja*).
    - Users always use the same command.

CMake basics

- Build description is stored in *CMakeLists.txt* .
  - One *CMakeLists.txt* per directory.
- Trivial raw CMake example. Two *CMakeLists.txt* files:

parent directory

```
cmake_minimum_required(VERSION 2.8.11)
project(HELLO)
add_subdirectory(Hello)
```

subdirectory Hello

```
add_executable(helloDemo helloDemo.cc)
```

- Language features:
  - Commands do not return values; they do modify arguments.
  - Commands (e.g., *add_subdirectory*) are case insensitive; keywords (e.g., *VERSION*) are case sensitive.
  - Users can write new commands.
    - Most of the content of cetbuildtools is new CMake commands.
    - I do not recommend end-users start writing new commands.

Build systems, CMake and cetbuildtools

- The completely trivial CMake example does not display the true usefulness of CMake
  - Could have done something nearly as simple with plain Make.
    - Would not have had automatic header dependency discovery, among other things.
- Real development projects become complicated very quickly.
- A completely trivial cetbuildtools example would not display the true usefulness of cetbuildtools.
  - See Example 2.

Fermilab

Using cetbuildtools

- We always separate source and build directories
  - It is optional to do so with plain CMake.
  - Separation is good practice.
    - Multiple builds from same source (e.g., optimized and debug).
    - Delete all build products without touching source.
- In Example 2, you will do

```
|alcourse>source ../art-workbook/ups/setup_for_development -p
$ART_WORKBOOK_QUAL
The working build directory is /home/amundson/work/build-prof2
The source code directory is /home/amundson/work/art-workbook
----------- check this block for errors -----------------------
----------------------------------------------------------------
<snip>
|alcourse>buildtool -j4
```

- The first command locates the source files and sets the hooks for the various dependencies

Fermilab

Using cetbuildtools

- The command

  buildtool –j4

  performs the actual build, including running CMake and the resulting build files
- The flag *–j4* tells *buildtool* to use up to four parallel processes.
  - More is generally better.
  - Limitations come from memory usage, shared resource problems, etc., as well as the fundamental size of the build.
- The command

  buildtool –-help (note: two dashes)

  will display help for buildtool commands.

Get Started

- Work on Exercise 2 (Chapter 10) of the *art* Workbook
  https://web.fnal.gov/project/ArtDoc/Shared%20Documents/art-documentation.pdf

Fermilab