How to Build & Package Open-Source Software (OSS)

Blue Waters Webinar Series

Prentice Bisbal

Princeton Plasma Physics Laboratory

February 6, 2019

PPPL PRINCETON PLASMA PHYSICS LABORATORY

PRINCETON UNIVERSITY

Objectives

- Teach how you how to compile and install open-source software (OSS)
  - Provide an overview of the process
  - Identify common issues and how to deal with them
- Provide advice for those of you writing OSS that will be compiled and installed by others
- Focus on OSS that uses a configure script create with GNU Autoconf
- Focus on the GCC compilers

Outline

- My bio / motivation
- Why this is important
- Building OSS
  - Intro. to the build process
  - Background information
    - Environment variables
    - Filesystem Hierarchy
    - The compiling process
  - The configuration process
  - The make && make install
  - Additional optional steps
    - make check
    - make clean
- Tips for packaging OSS

My background

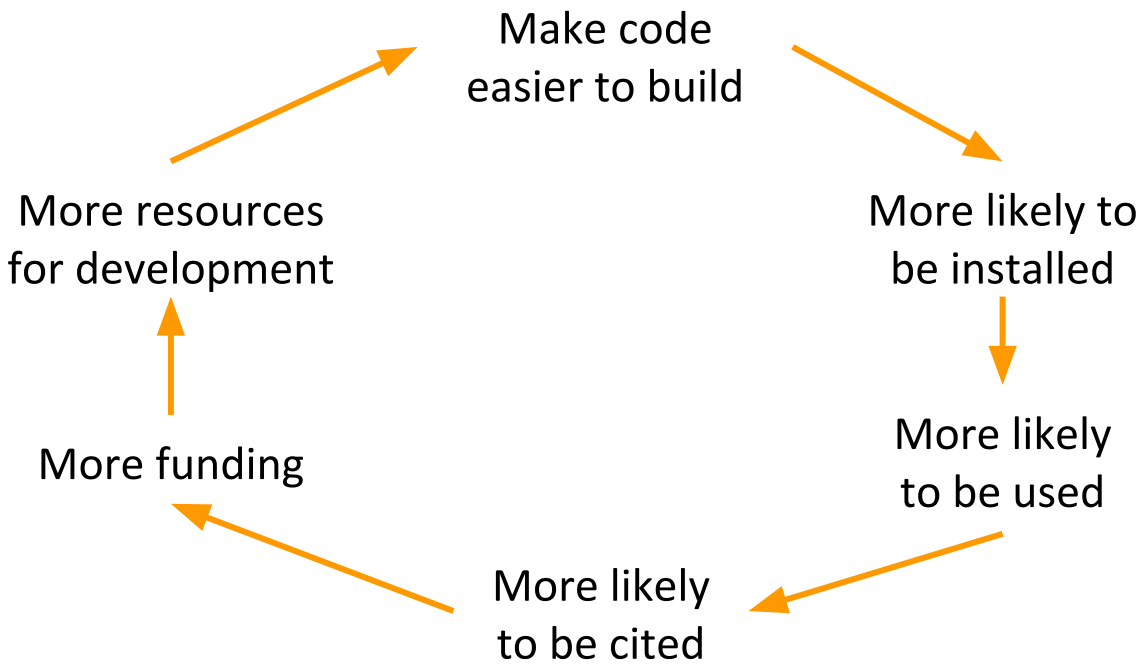
- BS in Chemical Engineering (not CS)
- Linux System administrator / HPC Specialist
- 20 years of experience
- Installed OSS thousands of times
- Significant experience supporting users
- Summer Program in Computational Astrophysics (2009)
  - Students required to build all the software themselves
  - Mailing list for students to help each other
  - Most-dominant topic: building the software
  - Broke down explanation of build process step-by-step for students
  - Very positive feedback from students.

Why do you need to know how to build OSS?

- Computational software is one of the tools-of-the-trade for computational scientists. Having access to the latest or best tools can help you be more productive.
- Most Linux distros provide little-or-no scientific software
- May not have adequate support staff at home institution to install software you need in a timely manner
- May want to have software on your laptop for working at home or when travelling.

Why should you care about packaging OSS?

- Only necessary for those who write code
- Beyond “publish or perish”: how often is your research cited?



The “Open-Source 5-step”

1. tar xvf foo-1.2.3.tar.gz
2. cd foo-1.2.3/
3. ./configure
4. make
5. make install

But...

- Running configure and building in source directory no longer proper practice
- How do you tell it where to install the software?
- How do you enable/disable different features?
- What if something goes wrong?

But first, some prerequisites

1. Environment variables
2. The filesystem hierarchy
3. The compiling process

Environment variables

Environment variables may need to be set before starting the build process, and will need to be set afterwards.

Environment variables

- Global variables available to a shell and its child processes
- Names are typically in ALL CAPS
- Examples: PATH, HOME, USER
- Programs often look at environment variables to determine how to run
- How to set an environment variable
  - Bash -> export
  - csh/tch -> setenv
- Can be set in login scripts (.bashrc, .bash_profile, .cshrc, .login, etc.)

Environment variables

- Configure will look at environment variable to determine desired behavior
  - CC
  - CXX
  - CPP
  - FC
  - CFLAGS
  - CXXFLAGS
  - CPPFLAGS
  - etc.
- Use `configure --help` to see list of environment variables for a specific application

Environment variables

- Certain environment variables will need to be set after installing software before you can use it:
  - PATH
  - LD_LIBRARY_PATH
  - MANPATH
  - Other application-specific variables
- These variables can be defined in your login scripts so they are set correctly every time you login or start a new shell
- Environment modules can be used to set these correctly

Environment variables

Demos

- env and printenv commands
- hello.sh
- ./configure --help
- edit .bashrc to change path

Linux Filesystem Hierarchy Standard (FHS)

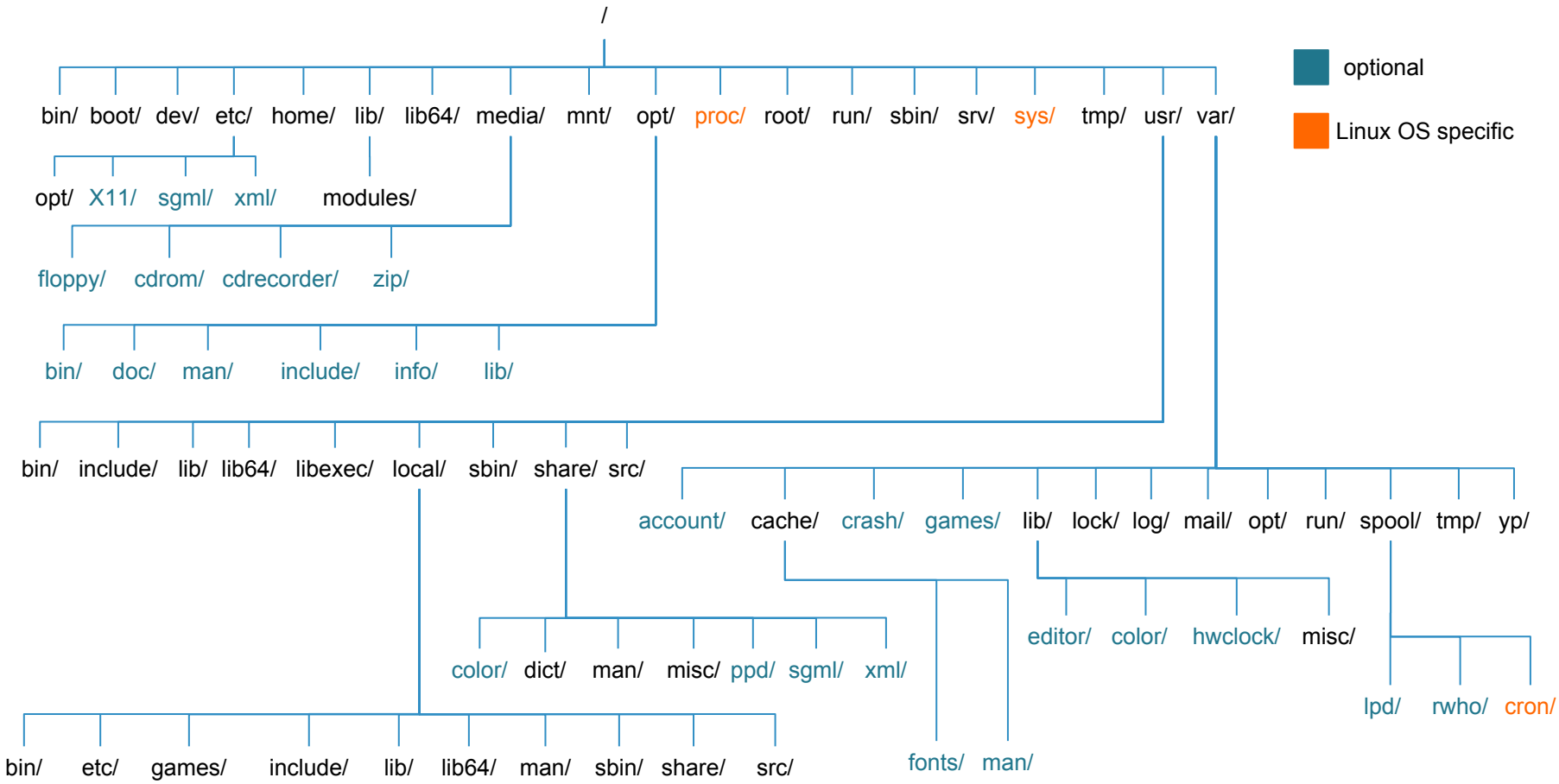
Since most building issues are related to search path issues, understanding where files are most like to be found can help resolve these issues.

Also, when installing software, it is best to install software in locations consistent with existing standards.

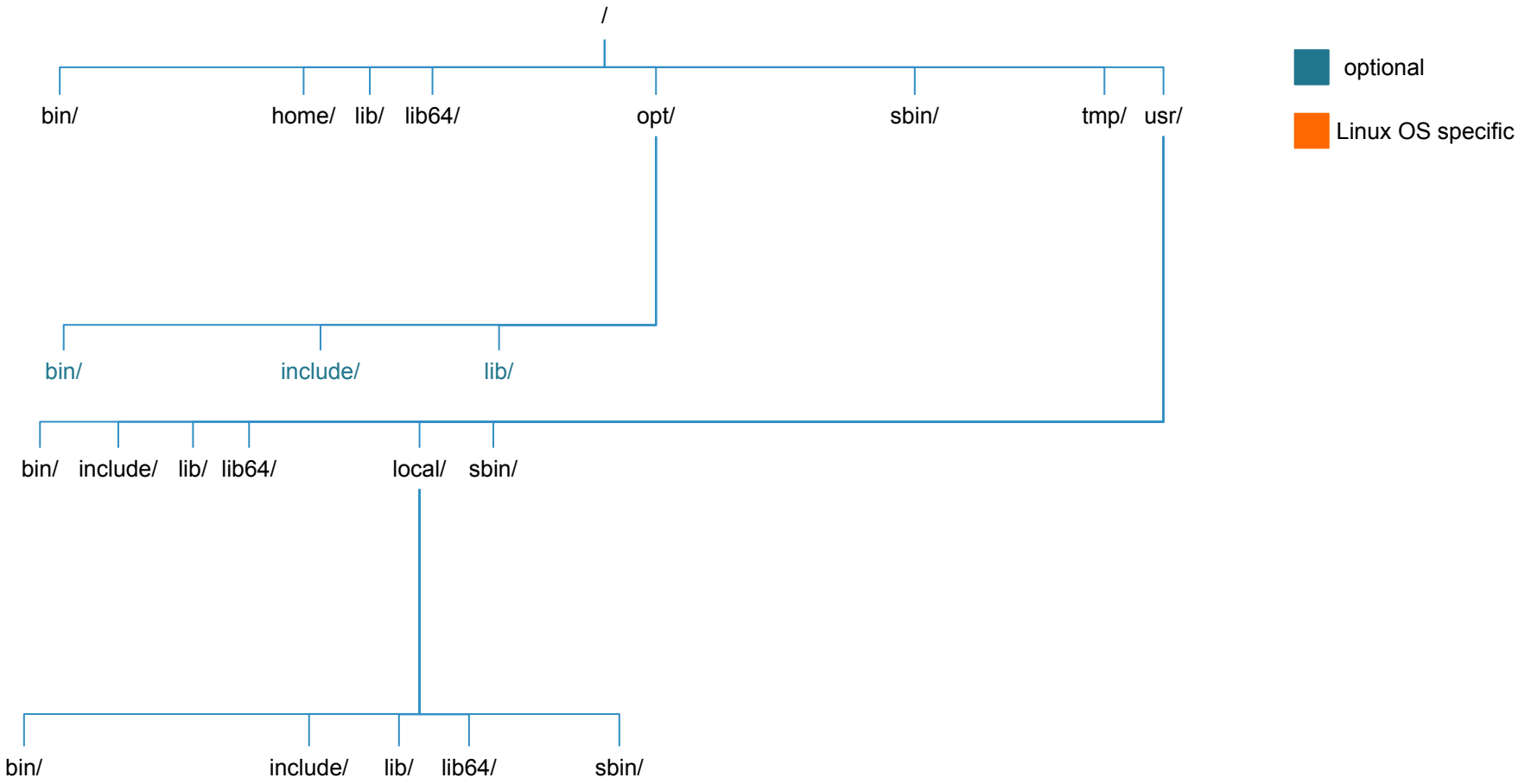
Linux Filesystem Hierarchy

- The Filesystem Hierarchy Standard (FHS) is an industry standard most Linux distros adhere to.
- https://refspecs.linuxfoundation.org/fhs.shtml
- "Local placement of local files is a local issue, so FHS does not attempt to usurp system administrators."
  - For example, all locally install software at PPPL is installed in /usr/pppl
  - Some sites install to /usr/local
  - Some commercial software prefers to install in /opt

Filesystem Hierarchy Standard (FHS)



LInux Filesystem Hierarchy Standard



Linux Filesystem Hierarchy

- Header files should be in a directory named “include”
  - /usr/include
  - /usr/local/include
  - /opt/include
  - Other site-specific locations
- Library files should be in directories named “lib” or “lib64”
  - /usr/lib
  - /usr/lib64
  - /lib
  - /lib64
  - /usr/local/lib
  - /opt/lib
  - Other site-specific locations

Linux Filesystem Hierarchy

- Commands should be in directories names “bin” or “sbin”
  - /usr/bin
  - /usr/sbin
  - /bin
  - /sbin
  - /usr/local/bin
  - /opt/bin
  - Other site-specific locations

The $HOME directory

You can do just about whatever you want in your own home directory, including install software without needing administrative rights. This allows you manage your own software needs on systems where you do not have administrative rights.

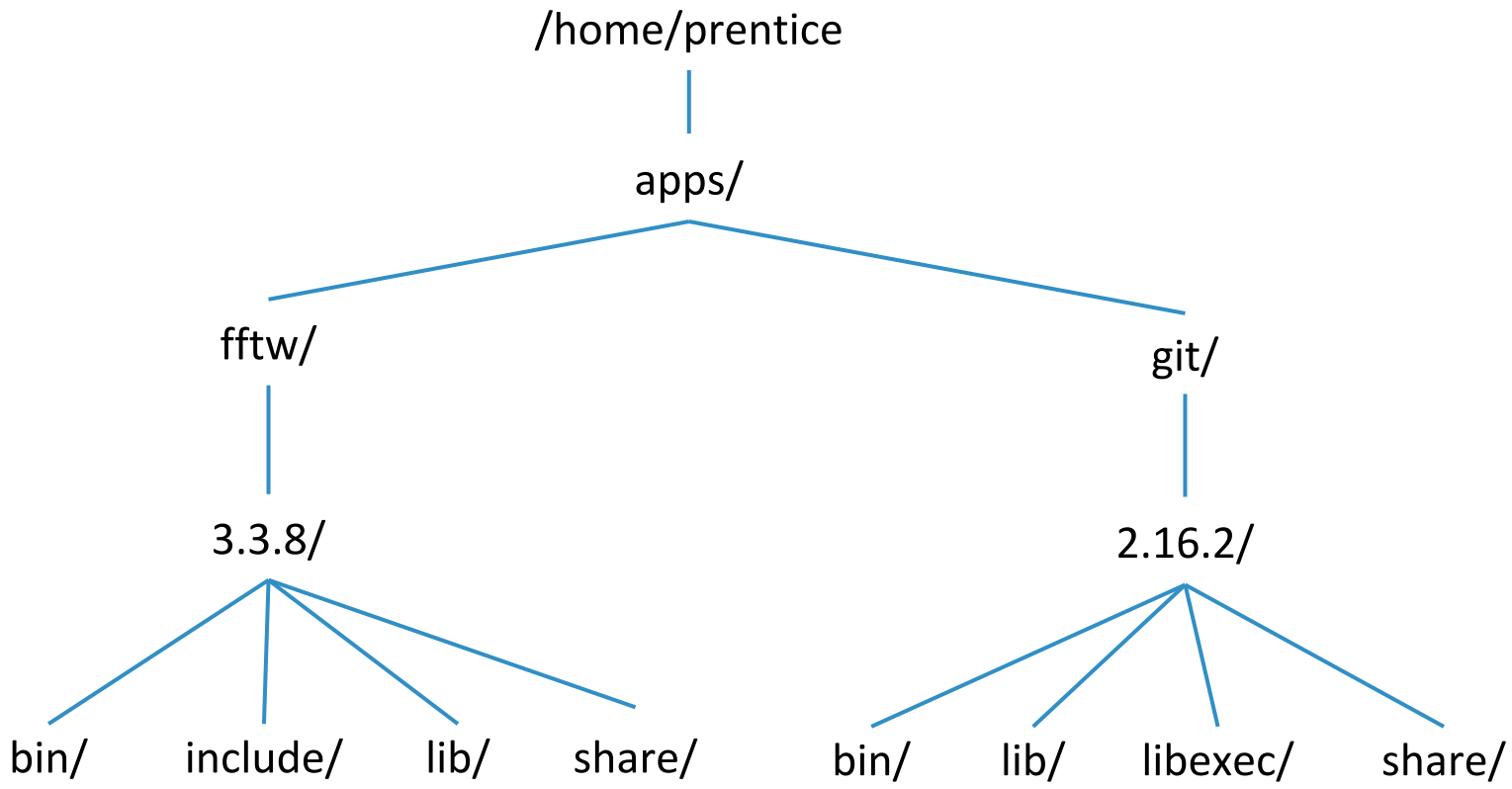
My recommendation, install software in your home directory in this way:

$HOME/apps/<app. name>/<app. version>

Examples:

- /home/prentice/apps/fftw/3.3.8
- /home/prentice/apps/git/2.16.2

Linux Filesystem Hierarchy



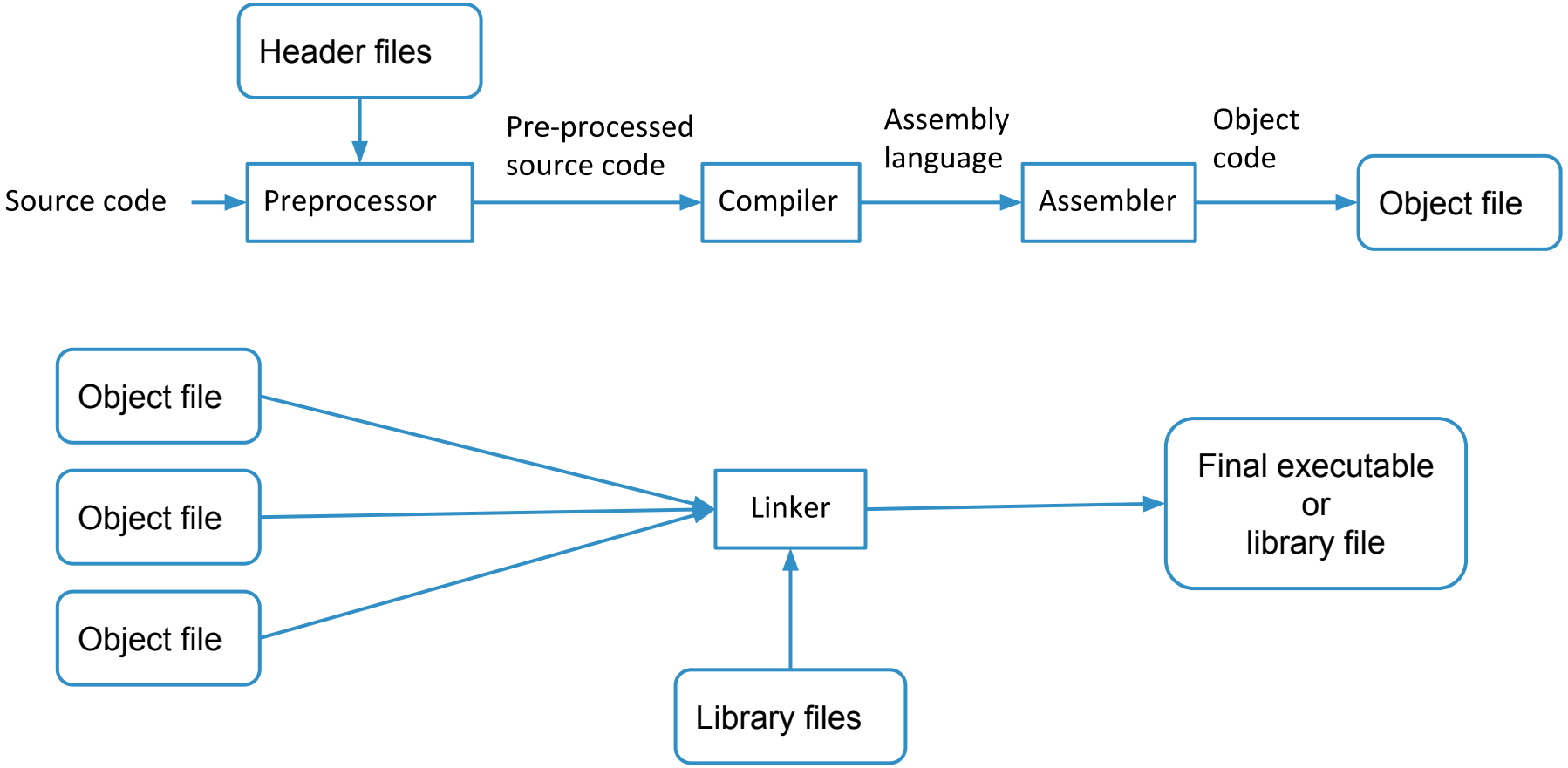
Tying it all (so far) together

```
FFTW_DIR=$HOME/apps/fftw/3.3.8
GIT_DIR=$HOME/apps/git/2.16.2

export PATH=$GIT_DIR/bin:$FFTW_DIR/bin:$PATH

export MANPATH=$GIT_DIR/share/man:$FFTW_DIR/share/man:$MANPATH

export LD_LIBRARY_PATH=$GIT_DIR/lib:$FFTW_DIR/lib:$LD_LIBRARY_PATH

export CPATH=$FFTW_DIR/include:$CPATH

export C_INCLUDE_PATH=$FFTW_DIR/include:$C_INCLUDE_PATH
```

The compiling process

Understanding the compiling process, and what errors happen at each stage are key to troubleshooting the build process.

The Compiling Process



The compiling process

- You execute the compiler command (gcc, g++, etc.) and the compiler calls the preprocessor, the compiler itself, the assembler, and the linker as necessary.
- The `-I` switch can be used multiple times to specify the paths to header files
- The `-L` switch can be used multiple times to specify the paths to library files
- The `-l` switch is used to specify the names of libraries to link to (`-l`<library name>)
- The order of libraries listed with the `-l` switches is important - the library that supplies a function should come after any libraries that call that function

The compiling process

- The GCC preprocessor will look at the following environment variables for header file search paths:
  - CPATH - header files for all languages
  - C_INCLUDE_PATH - header files for C
  - CPLUS_INCLUDE_PATH - header files for C++
  - OBJC_INCLUDE_PATH - header files for Objective C
- The GCC linker will look at the LIBRARY_PATH environment variable for library search paths

The compiling process

Example:

```
$ gcc -o example -I$HOME/apps/fftw/3.3.8/include \
    -L$HOME/apps/fftw/3.3.8/lib -lfftw3 example.c
```

Is equivalent to:

```
CFLAGS="-I$HOME/apps/fftw/3.3.8/include"
LDFLAGS="-L$HOME/apps/fftw/3.3.8/lib -lfftw3"
```

Or

```
CPATH=$HOME/apps/fftw/3.3.8/include
LIBRARY_PATH=/$HOME/apps/fftw/3.3.8/lib
```

Note: LIBRARY_PATH only handles path elements (`-L`), not individual libraries (`-l`)

Preprocessor errors - No such file or directory

Example:

```
$ gcc -o example example.c
example.c:2:10: fatal error: example.h: No such file or directory
 #include "example.h"
          ^~~~~~~~~~
```

How to fix:

1. Make sure the required package is installed (the “-dev” or “-devel” packages include the header files, or
2. Find the location of header files, and then
   a. Specify the location with the -I switch to the compiler, or
   b. Add the path the environment variables used by the C preprocessor like CPATH or C_INCLUDE_PATH

Linker errors - cannot find -l<librayname>

Example:

```
$ gcc -o myname -I./include  -lmyname myname.c
/usr/bin/ld: cannot find -lmyname
```

How to fix:

1. Make sure the required package is installed on your system (this is especially true for static libraries)
2. Find the correct location of the library file and then
   a. Specify the location to directory containing the library with the `-L` switch, or
   b. Add the directory containing the library to the LIBRARY_PATH environment variable

Linker errors - unresolved symbols

Example:

```
$ gcc -o liborder -I ./include -L ./lib -lfunction2 -lfunction3
liborder.c
./lib/libfunction2.so: undefined reference to `function1'
collect2: ld returned 1 exit status
```

How to fix:

1. Do an internet search to see what library provides the symbol, you might missing a library from your compiler command
2. Make sure the version of the library is correct. The symbol (function) may be new, or obsoleted.
3. Check the order of your libraries in your command.

Demos

- Using `-I` switch (example.c)
- Setting CPATH (example.c)
- Setting C_INCLUDE_PATH (example.c)
- Using `-L` switch (liborder.c)
- Setting LIBRARY_PATH (liborder.c)

The “Open-Source 5-step”

1. tar xvf foo-1.2.3.tar.gz
2. cd foo-1.2.3/
3. ./configure
4. make
5. make install

The configure script

- Created by the software's developer(s) using GNU Autoconf
- Most popular OSS configuration method
- Takes arguments to customize how your software is built
  - Specify installation location
  - Enable/disable features
  - Specify correct paths to dependencies
- Checks your environment to determine if prerequisites are available
- Creates the *Makefiles* that will guide the rest of the build process with the correct settings

The configure script - tips

- Always check README and INSTALL files if provided
- Always run `configure --help` to see all available options, and what environment variables configure will look at
- Common options to set:
  - --prefix
  - --enable-shared
  - --enable-static
  - --disable-silent-rules
- Good idea to pipe output to a log file using 'tee':

```
$ ./configure … 2>&1 | tee configure.log
```

Executing the configure script - example

```
./configure \
  --prefix=$HOME/apps/fftw/3.3.8 \
  --disable-silent-rules \
  --enable-shared \
  --enable-static \
  --enable-mpi \
  --enable-openmp \
  --enable-threads \
  CC=gcc \
  MPICC=mpicc \
  F77=gfortran \
  2>&1 | tee compile.log
```

make

This step actually calls the compiler to compile the source code

- No arguments necessary
- Use '`make -j <number>`' to run in parallel (optional)
- Good idea to pipe output to log file using 'tee':

```
$ make 2>&1 | tee make.log
```

make check (optional)

- Makes sure application produces the correct results
- Not every package provides this function
- Test results are not always reliables - see README and INSTALL documents
- XFAIL = Expected Failure
- Sometimes `make test` instead of `make check`

make clean (optional)

- Used to provide a “clean start” if need to start over
  - Removes all object files and executables
- `Make distclean` provides even more thorough cleaning
  - Removes configuration information, etc.
- Not every package provides these features

make install

- Copies the files to the location(s) you specified when you ran configure
- Sets the correct permissions
- Needs to be run as root, unless you are installing into your home directory.
- Good idea to pipe output to log file using ‘tee’:

```
$ make install 2>&1 | tee install.log
```

Demo

Go through process of building Hello 2.10

Packaging software

My observations on scientific OSS packaging

- Much harder to manage than general-purpose OSS
- Poor or no installation documentation
- No configuration process
- Files must be manually copied to installation location
- All file types (headers, libraries, binaries and data) in a single directory
- No version information
- Indecipherable filenames

Provide Consistent Version information

- Why?
  - It helps you keep track of your updates more easily
  - Helps the user determine whether they're using the latest version
  - Helps the user determine if bug reports or documentation applies to their version
- Several different versioning schemes commonly used:
  - Major.minor.bugfix (ex: 12.03.1)
  - Year.minor.bugfix (ex. 2018.01.03)
  - Choose one and stick with it.
- See https://semver.org/ for information on Semantic Versioning
- Should be easy to check version number with a command (--version switch, etc.)
- Downloadable tarballs should have version information in filename (master.tar.gz is NOT an acceptable filename)

Follow Linux Filesystem Hierarchy Standard

- Why?
  - To follow standard conventions
  - To put files in correct locations for required read-write privileges
- Additional file locations for developers
  - /tmp
  - /var/tmp
  - /var/log
  - /var/lib
  - /var/run
  - $HOME (for dot files)

Effective Installation Documentation

- Why?
  - Makes it easier for the user to install
- README and/or INSTALL file
  - README - general information about application, new features (release notes)
  - INSTALL - detailed installation instructions
- Should be readable from the command-line
  - ASCII text
  - Do not rely on web pages

Follow existing standards and convention

- Why?
  - Will make code maintenance easier
  - Fosters collaboration
  - Fosters portability
- Organize files in a hierarchy
- Code formatting (indenting, etc.)
- Variable naming
- The GNU Coding standards would be a good candidate

Configuration and Installation Automation Tools

- Why?
  - Makes it easier for you to make it easier for them
- Use a tool to automate configuration/installation
  - GNU Autoconf is by far the most popular
  - CMake is the second most popular, but far behind GNU Autoconf
  - SCons

Resources for building OSS

- Filesystem Hierarchy Standard: http://refspecs.linuxfoundation.org/fhs.shtml
- “An Intro to GCC” by Brian Gough: http://www.network-theory.co.uk/docs/gccintro/
- GNU Coding Standards, Section 7.1: How Configuration Should Work: https://www.gnu.org/prep/standards/html_node/Configuration.html#Configuration

Resources for Packaging OSS

- Filesystem Hierarchy Standard: http://refspecs.linuxfoundation.org/fhs.shtml
- GNU Coding Standards: https://www.gnu.org/prep/standards/
- GNU Autoconf: https://www.gnu.org/software/autoconf/autoconf.html
- GNU Automake: https://www.gnu.org/software/automake
- Information for maintainers of GNU software: https://www.gnu.org/prep/maintain/
- GNU Hello https://www.gnu.org/software/hello/

Resources for Packaging OSS

- “How To Make Package Managers Cry”, by Kenneth Hoste (FOSDEM 2018)
  https://www.youtube.com/watch?v=NSemlYagjIU
- CMake:
  https://cmake.org/
- SCons
  https://scons.org/

Questions / Feedback

“Feedback is a gift”

Please e-mail questions and feedback to pbisbal@pppl.gov