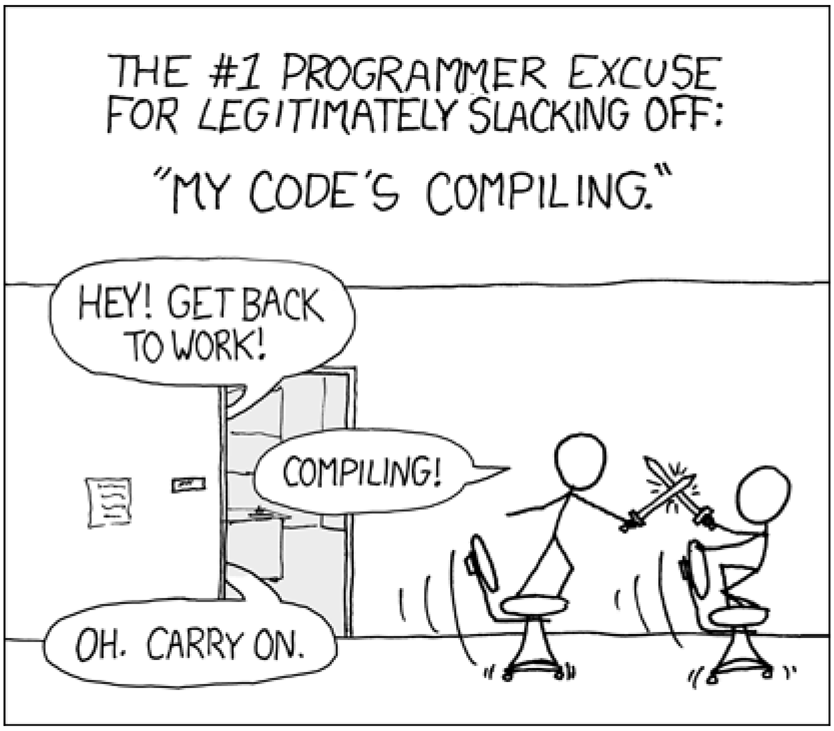
Compiling from Source in Linux

Presentation to Linux Users of Victoria, 21st April, 2017, by Lev Lafayette



Slide One: What Are You Doing?

- Source code is any collection of computer instructions usually written as plain text and usually in a high-level programming language (e.g., Java, C, C++, Fortran, Python etc). Source code is often transformed by an assembler, interpreter, or compiler into binary machine code which can be executed by the computer.
- In the earliest, first-generation, computers programs were entered directly in binary (i.e., no distinction between source code and machine code). Historically, hardware licensing also meant providing the source code. For example, IBM distributed source code as part of its license until 1983.
- Proprietary software became prevalent in the 1980s onwards where users received only the binary executable. The free and open-source software movement is a counter and a response to this - even when binary packages are offered.

Slide Two: Why Would You Do This?

- It is much easier to install packaged software in almost all cases. They are typically designed for the operating system and distribution you are using, and the installer or package manager (e.g., apt, yum, rpm, pacman, synaptic, portage) will usually manage any dependencies and updates.
- In some cases however, a package hasn't been written (too old or too new). An operator may also prefer a source version to ensure control over the dependencies included, to ensure security

patches are updated, or they may want multiple versions of the same software installed on their system.
- Software sources are absolutely required if you want to do any development on an application, whether for debugging, expanding functionality, or testing on different architectures. Compiled software is also necessary for optimisation to particular hardware.
- Performance of compiled software is often *much* faster than packaged software. Here is an example of a newer source version of BIND operating 10 times as fast as the default packaged version. `https://www.singlehop.com/blog/linux-series-stock-rpm-or-compile-from-source-tough-decisions/`

Slide Three: How Would You Do This? (Part I: Developer Tools)

- To install software from source, you will at a minimum need the appropriate developer tools on your system, such as compiler (e.g., GCC, G++, Intel, Make, Bison, Flex, Libtool etc). You will need to ensure all the dependencies are installed - and often these dependencies will have dependencies.
- For the purposes of this tutorial (and with time in mind) - and with a sense of irony - packages of build tools can be installed.

** For Debian, Ubuntu etc: `sudo apt-get install build-essential` ** For Red Hat, CentOS etc: `sudo yum groupinstall "Development Tools"` ** For ArchLinux: `sudo pacman -Sy base-devel`

*** Note for RH/CentOS 7 To install all the packages belonging to a package group called “Development Tools” use the following command to ensure optional packages are also installed. `sudo yum --setopt=group_package_types=mandatory,default,optional groupinstall "Development Tools"`

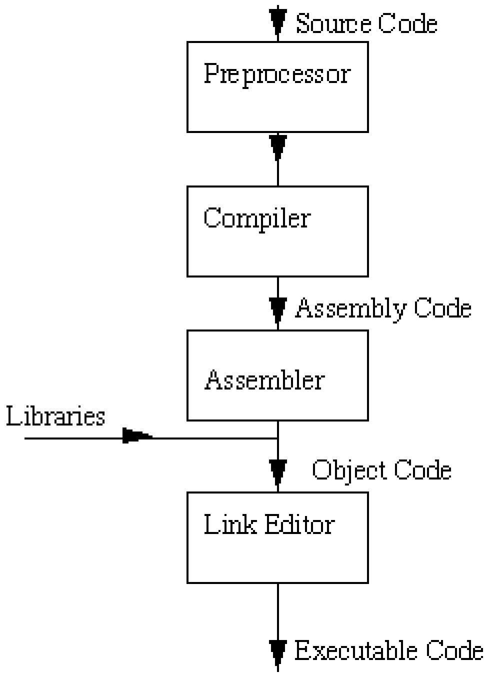
Slide Four: How Would You Do This? (Part II: Build Automation)

- The notion of a "build" is the conversion of the source code to machine code, carrying out activities such compilation, installation, and testing. Automated build tools aid a great deal in this process. If build automation tools do not exist the various checks and options have to be included in the compilation process itself or through shell scripts.
- The most common system is the GNU Autotools. This typically includes a `configure` process which scans the environment and generates an `Makefile` from the `config.status` it generates. This `Makefile` then can build the program (`make`) or alternatively carry out target instructions (e.g., `make clean`) where available. As root the program can be installed (`make install`).
- An alternative is CMake, which is cross-platform and compiler-independent. Compilation consists of two stages; build files are created from configuration files and then the build tools create the application. Each project contains a CMakeLists.txt file of commands.
- Some other build automation software includes: Apache Ant, which is implemented and largely designed for Java projects and uses XML to describe the process and targets., SCons, which generates project configurations and the build process as Python scripts. A relative newcomer is

Meson, a Python3-based system which is being adopted by several GNOME projects.

Slide Five: Simple GNU Autoconf Example (Hello World)

- A simple program does not need build automation; it can simply be compiled. For example in C, the source code starts with accepting the preprocessor directives (e.g., `#include`, `#define`) and removing comments. The compiler translates the source to assembly code, then the assembler creates object code (represented by the .o suffix. The link editor combines and references library functions in other source files, creating an executable.



```
#include <stdio.h>
int main()
{
    printf( "I am alive!  Beware.\n" );
/*    return 0; */
}
```

- If this source file is named HelloWorld.c it can be compiled with:

```
gcc -g -Wall HelloWorld.c -o Hello
```

(Notice the warning - fix it!)

- This could have a simple makefile (not normally necessary for something so small) e.g.,

```
CC      = gcc
CFLAGS  = -g -Wall
RM      = rm -f

default: all
```

```
all: Hello

Hello: HelloWorld.c
    $(CC) $(CFLAGS) -o Hello HelloWorld.c

clean veryclean:
    $(RM) Hello
```

- Large projects should have an `configure` as well as a `Makefile`! (Yes, I am looking at you LAMMPS : `http://lammps.sandia.gov/`)

Slide Six: GNU Autoconf Examples (Valgrind and GDB)

- Valgrind is a debugging suite that automatically detects many memory management and threading bugs. A suggested process for compiling from source follows:

```
sudo -s
mkdir -p /usr/local/src/VALGRIND
cd /usr/local/src/VALGRIND
wget http://www.valgrind.org/downloads/valgrind-3.10.1.tar.bz2
tar xvjf valgrind-3.10.1.tar.bz2
cd valgrind-3.10.1
./configure --prefix=/usr/local/$(basename $(pwd) | sed 's#-#/#')
make
make install
```

(Note - in these examples I am compiling and installing these applications as the root user - this is not best practise. An account specifically for installations is preferred c.f., EasyBuild)

- The GNU Debugger (GDB) is the standard debugger for the GNU software system. It is a portable debugger that runs on many Unix-like systems

```
sudo -s
mkdir -p /usr/local/src/GDB
cd /usr/local/src/GDB
wget ftp://ftp.gnu.org/gnu/gdb/gdb-7.12.1.tar.gz
tar xvf gdb-7.12.1.tar.gz
cd gdb-7.12.1
./configure --prefix=/usr/local/$(basename $(pwd) | sed 's#-#/#')
make
make install
```

In the HelloWorld example we compiled the program with the -g flags - which incorporates debugging information which can be used by GDB - good practise!

Slide Seven: CMake Example (GROMACS)

- The GROningen MAchine for Chemical Simulations (GROMACS) is a molecular dynamics simulation package that is very fast and has support for different force fields.

```
sudo -s
mkdir -p /usr/local/src/GROMACS
cd /usr/local/src/GROMACS
wget ftp://ftp.gromacs.org/pub/gromacs/gromacs-4.6.tar.gz
tar xvf gromacs-4.6.tar.gz
mkdir build
cd build
cmake -DGMX_X11=ON -DGMX_BUILD_OWN_FFTW=ON -DGMX_MPI=OFF
-DCMAKE_INSTALL_PREFIX=/usr/local/gromacs/4.6 ../gromacs-4.6
make -j 2
make install-mdrun
```

Slide Eight: Bugs, Modules, and Build Environments

- There is an enormous variety of things that can go wrong when building from source. You're in control and you have to make sure that the environment is suitable for the software you want to build. Some of the most common include:

** Missing dependecies and/or libraries - you'll need to install the dependency. ** Library or dependency not in path - you'll need to explicitly state the path (e.g., export LD_LIBRARY_PATH=$FFTWINSTALL/lib:$LD_LIBRARY_PATH)

- Once you have programs compiled from source you will want some way of accessing them. This can be done by adding them to your path manually (not recommended), adding them to a path via .bash_profile or equivalent (better), or - especially for multiple versions of the same program - by using a environment modules system which dynamically changes the path as needed (e.g., LMod `https://www.tacc.utexas.edu/research-development/tacc-projects/lmod`).
- When software is installed, it is valuable to keep a some sort of record of the build environment and procedure used. This can be as simple as a text file of notes, a shell script for configuration, or a fully fledged system such as EasyBuild (`https://easybuild.readthedocs.io/`), which includes a build recipe with options and specifies the compiler toolchain used; it also automatically builds the environment modules.

Slide Nine: Acknowledgements

- XKCD image from Randall Munroe
- Source compilation flowchart from David Marshall, Professor of Computer Vision, University of Cardiff