

# CSE471 - Free and Open Source Software

Dayanand V

`v_dayanand@cb.amrita.edu`

Dept. of Computer Science and Engineering  
Amrita School of Engineering  
Amrita Vishwa Vidyapeetham

March 16, 2016

# Credits

The credits for the contents of this presentation goes to Karl Fogel and his book “Producing Open Source Software”. You are as free to reuse and redistribute the contents of this presentation as I am in using Fogel’s book’s content.

# Contents

- 1 Introduction
- 2 History
- 3 Overview of the development process
- 4 Technical Infrastructure
- 5 Packaging, Releasing, and Daily Development
- 6 Licenses, Copyrights and Patents
- 7 Introduction to GNU Autotools

# Introduction

- About 90-95% free software projects fail. Why?
  - Unrealistic requirements
  - Vague specifications
  - Poor resource management
  - Insufficient design phases
- Aren't these the same reasons why closed-source projects fail?
  - Yes, and there's more.
- Sometimes, the developers do not appreciate the problems unique to open source development. Like?

# Introduction

- More reasons why open source projects fail.
  - ① Expecting hordes of oompa-loompas to magically volunteer for you.
  - ② Expecting that releasing the source will be a cure of all its ills.
  - ③ Skimping on presentation and packaging to develop “more important stuff”.
  - ④ Expecting that the same management practices used for in-house development will work equally well on an open source project.
  - ⑤ Failures of “Cultural Navigation”.

*“Open source does work, but it is most definitely not a panacea. If there’s a cautionary tale here, it is that you can’t take a dying project, sprinkle it with the magic pixie dust of “open source,” and have everything magically work out. Software is hard. The issues aren’t that simple.” -Jamie Zawinski, The Mozilla Project*

# Contents

- 1 Introduction
- 2 History
- 3 Overview of the development process
- 4 Technical Infrastructure
- 5 Packaging, Releasing, and Daily Development
- 6 Licenses, Copyrights and Patents
- 7 Introduction to GNU Autotools

# History

- At the dawn of commercial computers, ‘Software’ sounded closer to an accessory than a business asset.
- Customers (scientists, technicians etc) distributed patches to other customers and it was encouraged by the manufacturers.
- There was no hardware standardization and the manufacturer wanted machine-specific code and knowledge to spread as widely as possible.
- There was no Internet, and widespread frictionless sharing as we know it today, was not possible.

# The Rise of Proprietary Software and Free Software

- The advent of better 'hardware' and 'high level' programming languages.
- Selling software began looking like a good strategy.
- If the user had freedom to modify the software and improve it, the 'added value' patches would be of lesser significance.
- Shared code can reach competitors.
- Ironically, the Internet was getting off the ground around the same time.



# The Rise of Proprietary Software and Free Software - Conscious Resistance

- “Richard Stallman” happened.

*“We did not call our software ”free software”, because that term did not yet exist; but that is what it was. Whenever people from another university or a company wanted to port and use a program, we gladly let them. If you saw someone using an unfamiliar and interesting program, you could always ask to see the source code, so that you could read it, change it, or cannibalize parts of it to make a new program.”*  
-Richard Stallman

- He went ahead and started the GNU Project and Free Software Foundation (FSF).
- GNU was aimed at developing a completely free and open computer operating system and body of application s/w.
- “The GNU General Public License (GPL) says that the code may be copied and modified without restriction and that both copies and derivative works must be distributed *under the same license as the original with no additional restrictions.*”

# The Rise of Proprietary Software and Free Software - Conscious Resistance


- GNU slowly began gaining ground and was able to produce free replacements for many critical components of an OS.
- GNU Emacs came into being.
- The compiler collection GCC.
- By 1990s, GNU had produced most parts of a free OS except for the kernel.
- Linus Torvalds enters the scene with Linux, a kernel completed with the help of volunteers around the world.
- Becomes GNU Linux, the world's first free and open source operating system.

# The Rise of Proprietary Software and Free Software - Accidental Resistance

- Berkeley Software Distribution (BSD) - University of California, Berkeley.
- Non ideological practice of free software development.
- X Window System developed at MIT allowed proprietary extensions over free software, but by itself was free.
- T<sub>E</sub>X, developed by Donald Knuth offering free publishing quality typesetting system was released open source with just some naming restrictions on derivatives.

This presentation is built using ‘Beamer’ which is an extension of L<sup>A</sup>T<sub>E</sub>X, built on T<sub>E</sub>X.

# Free Versus Open Source

- Free? As in *free food*?
- Free as in *freedom*.
- Freedom to share and modify the source code for any purpose.
- Case study : Battle of browsers (1990s), Netscaps vs IE.
- Free software - Morality vs Profitability.
- Conflict of opinion.
- 1998 - the word “open source” is coined by Open Source Initiative (OSI ) to disambiguate “free” and aimed at giving ‘free software’ a good marketing.

# The Situation Today

- Free software has become a culture of choice.
- So what persuades all these people to stick together long enough to produce something useful?
- The feeling that their connection to a project, and influence over it, is directly proportional to their contributions.
- Clearly, projects with corporate sponsorship and/or salaried developers need to be especially careful in this regard.

# Contents

- 1 Introduction
- 2 History
- 3 Overview of the development process**
- 4 Technical Infrastructure
- 5 Packaging, Releasing, and Daily Development
- 6 Licenses, Copyrights and Patents
- 7 Introduction to GNU Autotools

# Look Around

*“Every good work of software starts by scratching a developer’s personal itch.”*  
-Eric Raymond, ‘The Cathedral and the Bazaar’.

- A good software results when the programmer has a personal interest in seeing the problem solved.
- Today, we also have the phenomenon of organizations-for-profit, corporations, governments, non-profits, etc starting large, centrally-conceived open source projects from scratch.
- Case Study : Kuali Foundation
- Identifying potential problems that are faced by a large community.
- So, first, *look around!* [github.com](https://github.com), [openhub.net](https://openhub.net), [sourceforge.net](https://sourceforge.net), [directory.fsf.org](https://directory.fsf.org)

# Start from What You Have

- Understanding clearly what the project is and what its not.
- Need for Documentation : Founders Vs New Comers.
- Investments : *hacktivation energy*.



# Start from What You Have

- Choose a Good Name.
  - Gives an idea about what the project does.
  - Easy to remember.
  - Good manners, good legal sense.
  - Getting top-level domains with forwarding to a central home site.
  - Handle availability in Twitter, FB.

# Start from What You Have

- Clear mission statement.
- State that the Project is 'Free'.
- A brief list of features and requirements.

# Start from What You Have

- Development Status
  - Should reflect reality.
  - Alpha, Beta releases.
  - Availability for download.
  - Version Control and Bug Tracker Access.
    - GitHub.com - based on Git.
    - Rate of bug filing.
- Communication Channels
  - Mailing List.
  - Chat room.
  - IRC Channel.
- Developer Guidelines.

# Start from What You Have

- Documentation.
  - Minimal criteria:
    - Should tell the reader clearly how much technical expertise they're expected to have.
    - Tells clearly how to setup, and make sure that they've installed the software correctly.
    - Give one tutorial-style example of how to do a common task.
    - Label the areas where the documentation is known to be incomplete.
  - Developer documentation.
- Demos, Screenshots, Videos, and Example Output.
- Hosting.

# Choosing a License and Applying It

- The “Do-Anything” licenses (MIT License).
  - Asserts nominal copyright (without actually restricting copying).
  - Code comes with no warranty.
- The GNU General Public License (GPL).
  - Translates loosely to - “If you use my code, make sure you let others use yours.”

# Choosing a License and Applying It

- Applying a license - Steps :
  - State the license clearly on the project's front page.
  - Put the full license text in a file called COPYING (or LICENSE) included with the source code.
  - Put a short notice in a comment at the top of each source file, naming :
    - Copyright date
    - Holder
    - The kind of license
    - Where to find the full text of the license.

# Setting the Tone

- Avoid private discussions ( $1 \ll n$ ).
- Zero tolerance to rudeness.
- Practice conspicuous code review.
- Be open from day one!
- Opening a formerly closed project.
- Announcing.

# Contents

- 1 Introduction
- 2 History
- 3 Overview of the development process
- 4 Technical Infrastructure**
- 5 Packaging, Releasing, and Daily Development
- 6 Licenses, Copyrights and Patents
- 7 Introduction to GNU Autotools



# Overview

- Website
- Mailing lists / Message forums
- Version control
- Bug tracking
- Real-time chat

# Version Control and Website

- Canned Hosting Services.
- Github.com
- Setting up a Github account.
  - ① Go to <https://github.com/>
  - ② Sign up - Get yourself an account.
  - ③ Go to <https://guides.github.com/activities/hello-world/>
  - ④ Learn how to :
    - ① Create a 'repository'.
    - ② Open an 'issue'.
    - ③ Create a 'branch'.
    - ④ Make a 'commit'.
    - ⑤ Open a 'pull request'.
    - ⑥ 'Merge' a pull request.
  - ⑤ Go to <https://pages.github.com/> and follow the steps given there to setup a website for your project.

# Mailing list, Chat and Wiki

- Google groups. *<https://www.groups.google.com>*
- Internet Relay Channel (IRC).  
*<http://www.irchelp.org/irchelp/altircfaq.html>*
- Mediawiki. *<https://www.mediawiki.org>*

# Contents

- 1 Introduction
- 2 History
- 3 Overview of the development process
- 4 Technical Infrastructure
- 5 Packaging, Releasing, and Daily Development**
- 6 Licenses, Copyrights and Patents
- 7 Introduction to GNU Autotools

# Introduction

- Highway Repair analogy.
- Mutually independent tasks are parallelized.
- Release works can happen in parallel to development.

# Releasing

- When does a release happen?
  - Some old bugs have been fixed and new ones added.
  - New features added.
  - New configuration options might have been added, changing the installation procedure.
  - Incompatible changes might have been introduced.

# Releasing Numbering

- Unambiguously communicates the ordering of the release.
- Indicates the degree and nature of the changes in the release.
- Be consistent in choosing a numbering scheme, and stick with it.
- Some examples would be:
  - “Project 1.2.3”
  - “Project 1.2.3 (Alpha)” with the qualifier.
  - “Project 1.2.3 (Alpha 2)” with meta-qualifier.
- Qualifiers : Alpha, Beta, Stable, Unstable, Development, and RC (for “Release Candidate”) etc.
- “RC” always includes a numeric meta-qualifier, “Project 1.2.3 (RC 1)” is possible, whereas “Project 1.2.3 (RC)” is not.

# Releasing Numbering

- $\langle \text{Project Name} \rangle \langle \text{Major Numbering} \rangle . \langle \text{Minor Numbering} \rangle . \langle \text{Micro Numbering} \rangle$
- “Project 2.10.17” is the eighteenth micro release (or patch release) in the eleventh minor release line within the second major release series.



# The Simple Strategy

- Adopted by *APR project (Apache)*.
- Changes to the micro number only must be both forward- and backward-compatible.
  - Only bug fixes, or very small enhancements to existing features.
  - No new features introduced in a micro release.
- Changes to the minor number must be backward-compatible, but not necessarily forward-compatible.
  - New features can be introduced, but not too many.
- Changes to the major number mark compatibility boundaries. A new major release can be forward- and backward-incompatible.
  - Entire new feature sets can be expected.

# The Even/Odd Strategy

- Parity of the minor number defines stability of the software.
- Even means stable, Odd means unstable.
- Increments in the micro number still indicate bug fixes (no new features).
- Increments in the major number still indicate big changes, new feature sets, etc.
- Compatibility guidelines are still preserved.

# Release Branches

- The entire development tree will not be clean and ready for release at any time.
- A release branch is just a branch in the version control system (see branch), on which the code destined for this release can be isolated from mainline development.
- When a release branch is ‘stabilized’, it is time to tag a snapshot (a micro release) from the branch.
- ‘Stabilization’ is the process of deciding which changes will be in the release, which will not, and shaping the branch content accordingly.
- The ‘Release Owner’ sets the release rules and has the final say in the case of any conflict.
- Voting on changes.
- ‘Release Manager’.

# Packaging

- Free software is distributed as source code or pre-built binary packages.
- Shipping formats:
  - \*nix OSes - TAR compressed by /emphcompress, gzip, bzip, or bzip2.
  - MS Windows - /emphzip format.
- Name of the package contains the software's name plus the release number, plus the format suffixes appropriate for the archive type.

# Layout

- Once uncompressed, the root directory should contain:
  - A 'README' plain text file describing :
    - what the software does.
    - what release this is.
    - link to the project's website etc.
  - An 'INSTALL' file giving instructions on how to build and install the software for all supporting operating systems.
  - A COPYING or LICENSE file, giving the software's terms of distribution.
  - A CHANGES file (sometimes called NEWS), explaining what's new in this release.

# Compilation and Installation

- For compiled languages, in \*NIX systems, the installation procedure of tarred source code typically would be like:
  - \$ ./configure (auto-detects the environment and prepares the build process)
  - \$ make (builds the software)
  - \$ make install (installs the software on the system)
- Installation of binary packages:
  - RPM system for RedHat GNU/Linux distros.
  - APT system for Debian GNU/Linux distros.

# Testing

- Regression testing + Manual Testing.
- Once all check-list criteria are met, the packages are digitally signed using *GnuPG* or *PGP* etc.
- Once approved, the release should be placed into the project's download area, accompanied by the digital signatures, and by MD5/SHA1 checksums.
- The purpose of all this signing and checksumming is to give users a way to verify that the copy they receive has not been maliciously tampered with.

# Candidate Releases

- Remember the qualifier 'RC'?
- Candidates releases are put out first before official release of a new version.
- The aim is to subject the code to wide testing before blessing it as an official release.
- If problems are found, they are fixed on the release branch and a new candidate release is rolled out.
- If there are no unacceptable bugs left, the last candidate release becomes the official release.



# Announcing Releases

- The announcement should include:
  - The URL to the downloadable release tarball should always be accompanied with the MD5/SHA1 checksums and pointers to the digital signatures file.
    - Announcing in multiple forums/ mailing lists.
    - Reassurance on security.
  - The relevant portion of the CHANGES file.
  - Credits where it is due.

# Maintaining Multiple Release Lines

- Most mature projects maintain multiple release lines in parallel.
- After 1.0.0 comes out, that line should continue with micro (bugfix) releases 1.0.1, 1.0.2, etc., until the project explicitly decides to end the line.
- Announcing the ‘end of life’ of release.
- Some projects set a window of time during which they pledge to support the previous release line.
- Some projects waits for the bug reports percentage to drops below a certain point to stop support.
- For each release, make sure to have a target version or target milestone available in the bug tracker, so people filing bugs will be able to do so against the proper release.

# Security Releases

- A security release is a release made **SOLELY** to close a security vulnerability.
- This is friendly to the administrators who may need to deploy the security fix, but whose upgrade policy stipulates that they not deploy any other changes at the same time.
- An extra component can be added to an existing release to indicate that it contains only security changes (Eg: 1.1.2.1).

# Releases and Daily Development

- Release Stabilization.
- Have each commit to be a single logical change, and don't mix unrelated changes in the same commit.
- If a change is too big or too disruptive to do in one commit, break it across N commits.
- Each commit is a well-partitioned subset of the overall change, and includes nothing unrelated to the overall change.

# Planning Releases

- Deadlines, Schedules.
- Discussions, Compromises.
- Depending on the complexity of the release process, and the nature of the project, somewhere between every three and six months is usually about the right gap between releases.
- Maintenance lines may put out micro releases a bit faster, if there is demand for them.

# Contents

- 1 Introduction
- 2 History
- 3 Overview of the development process
- 4 Technical Infrastructure
- 5 Packaging, Releasing, and Daily Development
- 6 Licenses, Copyrights and Patents**
- 7 Introduction to GNU Autotools

# Aspects of Licenses

- Differences between licenses boil down to a few often recurring issues :
  - Compatibility with proprietary licenses.
  - Compatibility with other types of free licenses.
  - Enforcement of crediting.
  - Protection of trademark.
  - Patent snapback.
  - Protection of “artistic integrity”.

# GPL and License Compatibility

- GPL Requirements:
  - Any derivative work must itself be distributed under the GPL.
  - No additional restrictions may be placed on the redistribution of either the original work or a derivative work.
- Incompatibilities:
  - Credit enforcement as a new policy cannot be applied on a GPL licensed code.
- Free Software Foundation maintains a *list* showing which licenses are compatible with the GPL and which are not.



# Choosing a License

- Use one of the widely-used, well-recognized existing licenses.
- Advantages:
  - People won't feel they have to read the legalese in order to use your code, because they will have already done so for that license a long time ago.
  - They are the products of much thought and experience and the modern versions represent a great deal of accumulated legal and technical wisdom.

# Choosing a License

- A list of licenses that meet the criteria: <sup>1</sup>
  - *GNU General Public License version 3 (GPL-3.0)*
  - *GNU Affero General Public License version 3 (AGPL-3.0)*
  - *Mozilla Public License 2.0 (MPL-2.0)*
  - *GNU Library or “Lesser” General Public License version 3 (LGPL-3.0)*
  - *Eclipse Public License 1.0 (EPL-1.0)*
  - *MIT license (MIT)*
  - *Apache License 2.0 (Apache-2.0)*
  - *BSD 2-Clause (“Simplified” or “FreeBSD”) license*

---

<sup>1</sup> **Assignment #3 : Study in detail and submit a report about the differences between the above licenses (with a clear tabular column) and where they would deem best to be used.**

# Contributor Agreements

- Handling copyright ownership for free code and documentation that were contributed to by many people:
  - ① Ignore the issue of copyright entirely.
  - ② Collect a contributor license agreement (CLA) from each person who works on the project, explicitly granting the project the right to use that person's contributions.
  - ③ Get actual copyright assignment (CA from contributors, so that the project (i.e., some legal entity, usually a nonprofit) is the copyright owner for everything.
- Even under centralized copyright ownership, the code remains free, because open source licenses do not give the copyright holder the right to retroactively proprietize all copies of the code.

## *Developer Certificates of Origin (DCO)*

- An attestation that the contributor intends to contribute the enclosed code under the project's license, and that the contributor has the right to do so.
- The contributor indicates her understanding of the DCO once, early on, for example by emailing its text from her usual contribution address to a special archive at the project.
- The contributor includes a “Signed-Off-By:” line in her patches or commits, using the same identity, to indicate that the corresponding contribution is certified under the DCO.
- The DCO relies on the project's native open source license for any trademark or patent provisions, which in most cases is fine.

# Proprietary Re-licensing

- An open source version of the software is available under the usual open source terms, while a proprietary version is available for a fee.
- Two types:
  - ① Selling exceptions to copyleft requirements.
    - Sells users a promise to not enforce the redistribution requirements of the open source version's license.
    - Eg:- MySQL Database Engine. GPL v2.
  - ② The freemium or open core model.
    - Uses an open source version to drive sales of a presumably fancier proprietary version.
- The catch here is that GPL's terms are something the copyright holder imposes on everyone else; the owner is therefore free to decide not to apply those terms to itself.

# Proprietary Re-licensing

- Problems with proprietary re-licensing:
  - ① Discourages the normal dynamics of open source projects.
  - ② Any code contributors from outside the company are now effectively contributing to two distinct entities, one which they usually do not prefer.
  - ③ there is a deep motivational problem for open source projects that operate in the shadow of a proprietary re-licensed version.
  - ④ Hybrid “shakedown” models: anyone who makes commercially significant use of the code ends up being pressured to purchase a proprietary license as a way of protecting their commercial revenue stream from harassment.

- A trademarked name or symbol is a way for an entity (who owns or controls that trademark), to signal, in an easily recognizable way, that they approve of a particular product.
- Trademarks do not restrict copying, modification, or redistribution.
- Trademark is unrelated to copyright, and does not govern the same actions that copyright governs.
- Case study: Mozilla Firefox, the Debian Project, and Iceweasel.
- Case study: The GNOME Logo and the Fish Pedicure Shop.

# Patents

- The only real threat against which the free software community cannot defend itself.
- Once someone has accused a free software project of infringing a patent, the project must either stop implementing that particular feature, or expose the project and its users to expensive and time-consuming lawsuits.
- Modern free software licenses generally have clauses to combat, or at least mitigate, the dangers arising from software patents.
- Usually these clauses work by automatically revoking the overall open source license for any party who makes a patent infringement claim based on either the work as a whole, or based on the claimant's code contributions to the project.



# Contents

- 1 Introduction
- 2 History
- 3 Overview of the development process
- 4 Technical Infrastructure
- 5 Packaging, Releasing, and Daily Development
- 6 Licenses, Copyrights and Patents
- 7 Introduction to GNU Autotools**

# Introduction to GNU Autotools

# Introduction to GNU Autotools

- Outcome of the demand for a de-facto standard for compiling and installing software.

# Introduction to GNU Autotools

- Outcome of the demand for a de-facto standard for compiling and installing software.
- Autoconf, Automake, and Libtool were developed separately, to make tackling the problem of software configuration more manageable by partitioning it.

# Introduction to GNU Autotools

- Outcome of the demand for a de-facto standard for compiling and installing software.
- Autoconf, Automake, and Libtool were developed separately, to make tackling the problem of software configuration more manageable by partitioning it.
- By-products of the GNU/Linux project.

# GNU Autotools and what they do!

- Autoconf:

# GNU Autotools and what they do!

- Autoconf:
  - A tool that makes your packages more portable by performing tests to discover system characteristics before the package is compiled.

# GNU Autotools and what they do!

- Autoconf:
  - A tool that makes your packages more portable by performing tests to discover system characteristics before the package is compiled.
  - The source code adapts to these differences.



# GNU Autotools and what they do!

- Autoconf:
  - A tool that makes your packages more portable by performing tests to discover system characteristics before the package is compiled.
  - The source code adapts to these differences.
- Automake:

# GNU Autotools and what they do!

- Autoconf:
  - A tool that makes your packages more portable by performing tests to discover system characteristics before the package is compiled.
  - The source code adapts to these differences.
- Automake:
  - A tool for generating ‘Makefile’s—descriptions of what to build—that conform to a number of standards.

# GNU Autotools and what they do!

- Autoconf:
  - A tool that makes your packages more portable by performing tests to discover system characteristics before the package is compiled.
  - The source code adapts to these differences.
- Automake:
  - A tool for generating ‘Makefile’s—descriptions of what to build—that conform to a number of standards.
  - It substantially simplifies the process of describing the organization of a package.

# GNU Autotools and what they do!

- Autoconf:
  - A tool that makes your packages more portable by performing tests to discover system characteristics before the package is compiled.
  - The source code adapts to these differences.
- Automake:
  - A tool for generating ‘Makefile’s—descriptions of what to build—that conform to a number of standards.
  - It substantially simplifies the process of describing the organization of a package.
  - Performs additional functions such as dependency tracking between source files.

# GNU Autotools and what they do!

- Autoconf:
  - A tool that makes your packages more portable by performing tests to discover system characteristics before the package is compiled.
  - The source code adapts to these differences.
- Automake:
  - A tool for generating ‘Makefile’s—descriptions of what to build—that conform to a number of standards.
  - It substantially simplifies the process of describing the organization of a package.
  - Performs additional functions such as dependency tracking between source files.
- Libtool:

# GNU Autotools and what they do!

- Autoconf:
  - A tool that makes your packages more portable by performing tests to discover system characteristics before the package is compiled.
  - The source code adapts to these differences.
- Automake:
  - A tool for generating ‘Makefile’s—descriptions of what to build—that conform to a number of standards.
  - It substantially simplifies the process of describing the organization of a package.
  - Performs additional functions such as dependency tracking between source files.
- Libtool:
  - A command line interface to the compiler and linker that makes it easy to portably generate static and shared libraries, regardless of the platform it is running on.

# History of Autotools

- The advent and popularity of Unix.

# History of Autotools

- The advent and popularity of Unix.
- POSIX standards.



# History of Autotools

- The advent and popularity of Unix.
- POSIX standards.
- Need for an organized approach to handle the differences.

# History of Autotools

- The advent and popularity of Unix.
- POSIX standards.
- Need for an organized approach to handle the differences.
- Earlier tools :

# History of Autotools

- The advent and popularity of Unix.
- POSIX standards.
- Need for an organized approach to handle the differences.
- Earlier tools :
  - The Metaconfig program, by Larry Wall, Harlan Stenn, and Raphael Manfredi.
  - The Cygnus 'configure' script, by K. Richard Pixley, and the original GCC 'configure' script, by Richard Stallman.
  - The GNU Autoconf package, by David MacKenzie.
  - Imake, part of the X Window system.

# History of Autotools

- The advent and popularity of Unix.
- POSIX standards.
- Need for an organized approach to handle the differences.
- Earlier tools :
  - The Metaconfig program, by Larry Wall, Harlan Stenn, and Raphael Manfredi.
  - The Cygnus ‘configure’ script, by K. Richard Pixley, and the original GCC ‘configure’ script, by Richard Stallman.
  - The GNU Autoconf package, by David MacKenzie.
  - Imake, part of the X Window system.
- All these tools had a *configuration step* and a *build step*.

# History of Autotools

- The advent and popularity of Unix.
- POSIX standards.
- Need for an organized approach to handle the differences.
- Earlier tools :
  - The Metaconfig program, by Larry Wall, Harlan Stenn, and Raphael Manfredi.
  - The Cygnus ‘configure’ script, by K. Richard Pixley, and the original GCC ‘configure’ script, by Richard Stallman.
  - The GNU Autoconf package, by David MacKenzie.
  - Imake, part of the X Window system.
- All these tools had a *configuration step* and a *build step*.
- The configuration step would generate ‘Makefile’s, and perhaps other files, which would then be used during the build step using the ‘make’ command.

# History of Autotools

- The advent and popularity of Unix.
- POSIX standards.
- Need for an organized approach to handle the differences.
- Earlier tools :
  - The Metaconfig program, by Larry Wall, Harlan Stenn, and Raphael Manfredi.
  - The Cygnus ‘configure’ script, by K. Richard Pixley, and the original GCC ‘configure’ script, by Richard Stallman.
  - The GNU Autoconf package, by David MacKenzie.
  - Imake, part of the X Window system.
- All these tools had a *configuration step* and a *build step*.
- The configuration step would generate ‘Makefile’s, and perhaps other files, which would then be used during the build step using the ‘make’ command.
- The use of ‘feature tests’ made software compatible across OS variants without change in source code.

# History of Autotools

- By 1994, Autoconf was a solid framework for handling the differences between Unix variants. But,

# History of Autotools

- By 1994, Autoconf was a solid framework for handling the differences between Unix variants. But,
  - Program developers still had to write large 'Makefile.in' files in order to use it.



# History of Autotools

- By 1994, Autoconf was a solid framework for handling the differences between Unix variants. But,
  - Program developers still had to write large ‘Makefile.in’ files in order to use it.
  - The ‘configure’ script generated by autoconf would transform the ‘Makefile.in’ file into a ‘Makefile’ used by the make program.

# History of Autotools

- By 1994, Autoconf was a solid framework for handling the differences between Unix variants. But,
  - Program developers still had to write large ‘Makefile.in’ files in order to use it.
  - The ‘configure’ script generated by autoconf would transform the ‘Makefile.in’ file into a ‘Makefile’ used by the make program.

# History of Autotools

- Since most programs are built in much the same way, there was a great deal of duplication in ‘Makefile.in’ files.

# History of Autotools

- Since most programs are built in much the same way, there was a great deal of duplication in ‘Makefile.in’ files.
- Also, the GNU project developed a reasonably complex set of standards for ‘Makefile’s, and it was easy to get some of the details wrong.

# History of Autotools

- Since most programs are built in much the same way, there was a great deal of duplication in ‘Makefile.in’ files.
- Also, the GNU project developed a reasonably complex set of standards for ‘Makefile’s, and it was easy to get some of the details wrong.
- These factors led to the development of Automake.

# History of Autotools

- Since most programs are built in much the same way, there was a great deal of duplication in ‘Makefile.in’ files.
- Also, the GNU project developed a reasonably complex set of standards for ‘Makefile’s, and it was easy to get some of the details wrong.
- These factors led to the development of Automake.
  - The developer writes files named ‘Makefile.am’; these use a simpler syntax than ordinary ‘Makefile’s. automake reads the ‘Makefile.am’ files and produces ‘Makefile.in’ files.

# History of Autotools

- Since most programs are built in much the same way, there was a great deal of duplication in ‘Makefile.in’ files.
- Also, the GNU project developed a reasonably complex set of standards for ‘Makefile’s, and it was easy to get some of the details wrong.
- These factors led to the development of Automake.
  - The developer writes files named ‘Makefile.am’; these use a simpler syntax than ordinary ‘Makefile’s. automake reads the ‘Makefile.am’ files and produces ‘Makefile.in’ files.
  - A script generated by autoconf converts these ‘Makefile.in’ files into ‘Makefile’s.

# History of Autotools

- Since most programs are built in much the same way, there was a great deal of duplication in ‘Makefile.in’ files.
- Also, the GNU project developed a reasonably complex set of standards for ‘Makefile’s, and it was easy to get some of the details wrong.
- These factors led to the development of Automake.
  - The developer writes files named ‘Makefile.am’; these use a simpler syntax than ordinary ‘Makefile’s. automake reads the ‘Makefile.am’ files and produces ‘Makefile.in’ files.
  - A script generated by autoconf converts these ‘Makefile.in’ files into ‘Makefile’s.
  - automake also adds any rules required by the GNU ‘Makefile’ standards.



# History of Autotools

- Since most programs are built in much the same way, there was a great deal of duplication in ‘Makefile.in’ files.
- Also, the GNU project developed a reasonably complex set of standards for ‘Makefile’s, and it was easy to get some of the details wrong.
- These factors led to the development of Automake.
  - The developer writes files named ‘Makefile.am’; these use a simpler syntax than ordinary ‘Makefile’s. automake reads the ‘Makefile.am’ files and produces ‘Makefile.in’ files.
  - A script generated by autoconf converts these ‘Makefile.in’ files into ‘Makefile’s.
  - automake also adds any rules required by the GNU ‘Makefile’ standards.

# History of Autotools

- Unix added support for shared libraries.

# History of Autotools

- Unix added support for shared libraries.
- Advantages of using shared libraries.

# History of Autotools

- Unix added support for shared libraries.
- Advantages of using shared libraries.
- Development of Libtool began.

# History of Autotools

- Unix added support for shared libraries.
- Advantages of using shared libraries.
- Development of Libtool began.
  - Libtool is a collection of shell scripts which handle the differences between shared library generation and use on different systems.
  - It is closely tied to Automake, although it is possible to use it independently.
- Over time, Libtool has been enhanced to support more Unix variants and to provide an interface for standardizing shared library features.

# History of Autotools

- In 1995, Microsoft released Windows 95 which became widely popular.

# History of Autotools

- In 1995, Microsoft released Windows 95 which became widely popular.
- Autoconf and Libtool provided a framework to support portability (from a single source code base) to Windows as well.

# History of Autotools

- In 1995, Microsoft released Windows 95 which became widely popular.
- Autoconf and Libtool provided a framework to support portability (from a single source code base) to Windows as well.
- To support this, the GNU bash shell was ported to Windows as Cygwin which implements the basic Unix API in Windows.



# History of Autotools

- In 1995, Microsoft released Windows 95 which became widely popular.
- Autoconf and Libtool provided a framework to support portability (from a single source code base) to Windows as well.
- To support this, the GNU bash shell was ported to Windows as Cygwin which implements the basic Unix API in Windows.
- Autoconf and Libtool support Windows directly, using either the Cygwin interface or the Visual C++ tools from Microsoft.

# History of Autotools

- In 1995, Microsoft released Windows 95 which became widely popular.
- Autoconf and Libtool provided a framework to support portability (from a single source code base) to Windows as well.
- To support this, the GNU bash shell was ported to Windows as Cygwin which implements the basic Unix API in Windows.
- Autoconf and Libtool support Windows directly, using either the Cygwin interface or the Visual C++ tools from Microsoft.
- Automake has also been ported to Windows but it requires Perl to be installed.

# Running ‘configure’

- Command line options.

- [https://www.sourceware.org/autobook/autobook/autobook\\_14.html#Configuring](https://www.sourceware.org/autobook/autobook/autobook_14.html#Configuring)

# Running ‘configure’

- Command line options.
  - [https://www.gnu.org/autobook/autobook/autobook\\_14.html#Configuring](https://www.gnu.org/autobook/autobook/autobook_14.html#Configuring)
  - ‘--cache-file=file’

# Running ‘configure’

- Command line options.
  - [https://www.gnu.org/autobook/autobook/autobook\\_14.html#Configuring](https://www.gnu.org/autobook/autobook/autobook_14.html#Configuring)
  - ‘--cache-file=file’
  - ‘--help’

# Running ‘configure’

- Command line options.
  - [https://www.gnu.org/autobook/autobook/autobook\\_14.html#Configuring](https://www.gnu.org/autobook/autobook/autobook_14.html#Configuring)
  - ‘--cache-file=file’
  - ‘--help’
  - ‘--quiet’ or ‘--silent’

# Running ‘configure’

- Command line options.
  - [https://www.gnu.org/autobook/autobook/autobook\\_14.html#Configuring](https://www.gnu.org/autobook/autobook/autobook_14.html#Configuring)
  - ‘--cache-file=file’
  - ‘--help’
  - ‘--quiet’ or ‘--silent’
  - ‘--disable-⟨feature⟩’

# Running ‘configure’

- Command line options.
  - [https://www.gnu.org/autobook/autobook/autobook\\_14.html#Configuring](https://www.gnu.org/autobook/autobook/autobook_14.html#Configuring)
  - ‘--cache-file=file’
  - ‘--help’
  - ‘--quiet’ or ‘--silent’
  - ‘--disable-⟨feature⟩’
  - ‘--enable-feature[=arg]’etc..
- Source tree, Build tree and Install tree.



# Files generated by configure

- 'config.cache'

## Files generated by configure

- 'config.cache'
- 'config.log'

# Files generated by configure

- 'config.cache'
- 'config.log'
- 'config.status'

# Files generated by configure

- 'config.cache'
- 'config.log'
- 'config.status'
- 'config.h'

# Files generated by configure

- 'config.cache'
- 'config.log'
- 'config.status'
- 'config.h'
- 'Makefile'

# Files generated by configure

- 'config.cache'
- 'config.log'
- 'config.status'
- 'config.h'
- 'Makefile'

# Makefiles and targets

- Makefiles contain specifications of dependencies between files and how to resolve those dependencies
- Targets - the task to be performed by the Makefile:
  - make *all*
  - make *check*
  - make *install*
  - make *clean* etc.

# Make files

- The ‘make’ program attempts to bring a target up to date by bring all of the target’s dependencies up to date (the dependent has a more recent timestamp than the target).
- Dependencies and dependency graph.

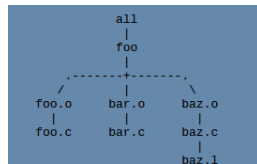
```
all: foo

foo: foo.o bar.o baz.o

.c.o:
    $(CC) $(CFLAGS) -c $< -o $@

.l.c:
    $(LEX) $< && mv lex.yy.c $@
```

(a) Makefile



(b) Dependency Graph

- When leaf nodes are found in the dependency graph, the ‘Makefile’ must include a set of shell commands to bring the dependent up to date with the dependency.



# Makefiles

- Each of the shell commands are run in their own sub-shell and, unless the 'Makefile' instructs make otherwise.
- Each command must exit with an exit code of 0 to indicate success.
- Target rules can be written which are executed unconditionally (by specifying that the target has no dependents).

# Writing Makefiles

- Syntax:

```
target1: dep1 dep2 ... depN
<tab>    cmd1
<tab>    cmd2
<tab>    ...
<tab>    cmdN

target2: dep4 dep5
<tab>    cmd1
<tab>    cmd2

dep4 dep5:
<tab>    cmd1
```

- Command prefixes - '@', '-'
- Macros
  - Starts with a dollar sign.
  - Eg:- \$CC
  - Define a make variable using a 'var=value' syntax: 'CC = ec++' (Default value of \$CC is 'cc')
  - '\$@' and '\$/' represent the names of the target and the first dependency for the rule in which they appear

# Writing Makefiles

- Example:

```
all:    dummy
        @echo "$@" depends on dummy"

dummy:
        touch $@
```

(c) File

```
$ make
touch dummy
all depends on dummy
```

(d) Output

- Suffix Rules: wildcard pattern that can match targets
- Example:

```
.c.o:    $(CC) $(CFLAGS) -c $< -o $@
```

- This rule will match any target that ends in ‘.o’ and are said to always be dependent on ‘.c’