Automake

Automake part of a set of tools called *The Autotools*. *The Autotools* are tools that will create a *GNU Build System* for your package.

Automake is a *tool* for automatically generating **Makefile.in**s from files called **Makefile.am**. The generated **Makefile.in**s are compliant with the *GNU Makefile standards*.

The goal of Automake is to remove the burden of Makefile maintenance from the back of the individual GNU maintainer.

The typical Automake input file is simply a series of variable definitions. Each such file is processed to create a **Makefile.in**.

Automake does constrain a project in certain ways; for instance, it assumes that the project uses **Autoconf** (see Introduction in The Autoconf Manual), and enforces certain restrictions on the **configure.ac** contents.

At the beginning of this tuturial, we shall install the following tools:

1. GNU Automake https://ftp.gnu.org/gnu/automake/

   ```
   wget -c https://ftp.gnu.org/gnu/automake/automake-1.16.1.tar.gz %linux
   ```

   Or:

   ```
   curl -O http://ftp.gnu.org/automake/automake-1.16.1.tar.gz  %mac
   ```

   ```
   tar -xzvf automake-1.16.1.tar.gz
   ```

   ```
   cd automake-1.16.1
   ./configure
   make
   make install
   ```

   ```
   info Automake
   ```

2. GNU Autoconf http://mirrors.nju.edu.cn/gnu/autoconf/

   ```
   wget -c http://ftpmirror.gnu.org/autoconf/autoconf-latest.tar.gz
   Or:
   curl -O http://ftpmirror.gnu.org/autoconf/autoconf-latest.tar.gz
   ```

Here we just use one simple example of **hello.c**.

Example 1: hello.c

Firstly, we can see one picture as following.

- Preprocessing phase. The preprocessor (cpp) modifies the original C program according to directives that begin with the # character. For example, the #include <stdio.h> command in line 1 of hello.c tells the preprocessor to read the contents of the system header file stdio.h and insert it directly into the program text. The result is another C program, typically with the .i suffix.
- Compilation phase. The compiler (cc1) translates the text file hello.i into the text file hello.s, which contains an assembly-language program. Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.
- Assembly phase. Next, the assembler (as) translates hello.s into machine- language instructions, packages them in a form known as a relocatable object program, and stores the result in the object file hello.o. The hello.o file is a binary file whose bytes encode machine language instructions rather than characters. If we were to view hello.o with a text editor, it would appear to be gibberish.
- Linking phase.Notice that our hello program calls the printf function, which is part of the standard C library provided by every C compiler. The printf function resides in a separate precompiled object file called printf.o, which must somehow be merged with our hello.o program. The linker (ld) handles this merging. The result is the hello file, which is an executable object file (or simply executable) that is ready to be loaded into memory and executed by the system.

1. Writing hello.c

```
#include <stdio.h>
int main(void)
{
    printf("Hello, Automake!\n")
    return 0;

}
```

2. Editing Makefile.am

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```

3. configure.scan -> configure.ac

```
autoscan
vim configure.scan
mv configure.scan configure.ac
```

4. configure

```
aclocal
autoconf
```

5. Automake

```
automake
```

6. ./configure ->Makefile

```
./configure
```

7. make

```
make
```

Example 2: amhello

1. Creating amhello-1.0.tar.gz

The package is simple enough so that we will only need to write *5* files. (You may copy them from the final **amhello-1.0.tar.gz** that is distributed with **Automake** if you do not want to write them.)

- **configure.ac**

  This file is read by both `autoconf` (to create configure) and `automake` (to create the various Makefile.ins). It contains a series of M4 macros that will be expanded as shell code to finally form the configure script.

```
AC_INIT([amhello], [1.0], [bug-automake@gnu.org])
AM_INIT_AUTOMAKE([-Wall -Werror foreign])
AC_PROG_CC
AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([
 Makefile
 src/Makefile
])
AC_OUTPUT
```

The macros prefixed with `AC_` are Autoconf macros, documented in the Autoconf manual (see Autoconf Macro Index in The Autoconf Manual). The macros that start with `AM_` are Automake macros, documented later in this manual (see Macro Index).

The first two lines of configure.ac initialize Autoconf and Automake. `AC_INIT` takes in as parameters the name of the package, its version number, and a contact address for bug-reports about the package.

The argument to `AM_INIT_AUTOMAKE` is a list of options for `automake` (see Options). -Walland -Werror ask `automake` to turn on all warnings and report them as errors.

Using **-Wall -Werror** is a safe setting when starting to work on a package: you do not want to miss any issues.

The **foreign** option tells Automake that this package will not follow the GNU Standards.

The `AC_PROG_CC` line causes the `configure` script to search for a C compiler and define the variable `CC` with its name.

The `AC_CONFIG_HEADERS([config.h])` invocation causes the `configure` script to create a config.h file gathering '#define's defined by other macros in configure.ac.

The `AC_CONFIG_FILES` macro declares the list of files that `configure` should create from their *.in templates. Automake also scans this list to find the Makefile.am files it must process.

- **src/Makefile.am**

```
bin_PROGRAMS = hello
hello_SOURCES = main.c
```

Variables that end with `_PROGRAMS` are special variables that list programs that the resulting Makefile should build. `_PROGRAMS` suffix is called a*primary*; Automake recognizes other primaries such as `_SCRIPTS`, `_DATA`, `_LIBRARIES`, etc. corresponding to different types of files.

The 'bin' part of the `bin_PROGRAMS` tells `automake` that the resulting programs should be installed in bindir

- **Makefile.am**

```
SUBDIRS = src
dist_doc_DATA = README
```

`SUBDIRS` is a special variable listing all directories that `make` should recurse into before processing the current directory.

The line `dist_doc_DATA = README` causes README to be distributed and installed in docdir

- **src/main.c**

```
#include <config.h>
#include <stdio.h>

int
main (void)
{
    puts ("Hello World!");
    puts ("This is" PACKAGE_STRING ".");
    return 0;
}
```

- **README**

```
This is a demonstration package for GNU Automake.
Type 'info Automake' to read the Automake manual.
```

Then we can do the command:

```
autoreconf --install
```

`autoreconf` created four other files: configure, config.h.in, Makefile.in, and src/Makefile.in. The latter three files are templates that will be adapted to the system by `configure` under the names config.h, Makefile, and src/Makefile.

```
./configure
make
src/hello
```

2. Install program produced by Automake and Autoconf.

Tar file: amhello-1.0.tar.gz

```
tar -xzvf amhello-1.0.tar.gz
cd amhello-1.0

./configure --prefix ~/usr
make
make check
make install
make installcheck
```

Here is a list of the most useful targets that the GNU Coding Standards specify.

- `make all`

  Build programs, libraries, documentation, etc. (same as `make`).

- `make install`

  Install what needs to be installed, copying the files from the package's tree to system-wide directories.

- `make install-strip`

  Same as `make install`, then strip debugging symbols. Some users like to trade space for useful bug reports...

- `make uninstall`

  The opposite of `make install`: erase the installed files. (This needs to be run from the same build tree that was installed.)

- `make clean`

  Erase from the build tree the files built by `make all`.

- `make distclean`

  Additionally erase anything `./configure` created.

- `make check`

  Run the test suite, if any.

- `make installcheck`

  Check the installed programs or libraries, if supported.

- `make dist`

  Recreate package-version.tar.gz from all the source files.

Autoscan

The **autoscan** program can help you create and /or maintain a **configure .ac** file.

Reference

- CSAPP
- Wiki

  https://en.wikipedia.org
- GNU Automake manual https://www.gnu.org/software/automake/manual/automake.html#Why-Autotools

Tips

- **m4** is a general-purpose macro processorincluded in all UNIX-like operating systems, and is a component of the POSIX standard.
- .tar

  ```
  tar -cvf filename.tar filename
  tar -xvf filename.tar
  ```

  .tar.gz

  ```
  tar -czvf filename.tar.gz filename
  tar -xzvf filename.tar.gz
  ```

  .tar.xz

  ```
  xz -d filename.tar.xz
  tar -xvf filename.tar
  ```

- **GNU Build System**

  The GNU Coding Standards (see The Release Process in The GNU Coding Standards) explains how each package of the GNU project should have a configure script, and the minimal interface it should have. The Makefile too should follow some established conventions. The result? A unified build system that makes all packages almost indistinguishable by the installer. In its simplest scenario, all the installer has to do is to unpack the package, run `./configure && make && make install`, and repeat with the next package to install.
- **GNU** project

A tax-exempt charity started by Richard Stallman in 1984, with the ambitious goal of developing a complete Unix-like system whose source code is unencumbered by restrictions on how it can be modified or distributed. The GNU project has developed an environment with all the major components of a Unix operating system, except for the kernel, which was developed separately by the Linux project. The GNU environment includes the emacs editor, gcc compiler, gdb debugger, assembler, linker, utilities for manipulating binaries, and other components. The gcc compiler has grown to support many different languages, with the ability to generate code for many different machines. Supported languages include C, C++, Fortran, Java, Pascal, Objective-C, and Ada.