If you’ve used any flavour of Unix for development, you’ve probably installed software from source with this magic incantation:

+

./configure

make

make install

+

I know I’ve typed it a lot, but in my early days using Linux I didn’t really understand what it meant, I just knew that if I wanted to install software this was the spell to recite.

+

Recently I’ve been building my own Unix tools, and I wanted to tap into this standard install process; not only is it familiar to many Unix users, it’s also a great starting point for building a package for Homebrew and the various Linux and BSD package managers. It was time to dig into the Unix Grimoire and find out what the incantation does.

[**What does all of this do**](https://robots.thoughtbot.com/the-magic-behind-configure-make-make-install#what-does-all-of-this-do)

+

There are three distinct steps in this process:

+

**1. Configure the software**

+

The configure script is responsible for getting ready to build the software on your specific system. It makes sure all of the dependencies for the rest of the build and install process are available, and finds out whatever it needs to know to use those dependencies.

+

Unix programs are often written in C, so we’ll usually need a C compiler to build them. In these cases the configure script will establish that your system does indeed have a C compiler, and find out what it’s called and where to find it.

+

**2. Build the software**

+

Once configure has done its job, we can invoke make to build the software. This runs a series of tasks defined in a Makefile to build the finished program from its source code.

+

The tarball you download usually doesn’t include a finished Makefile. Instead it comes with a template called Makefile.in and the configure script produces a customised Makefile specific to your system.

+

**3. Install the software**

+

Now that the software is built and ready to run, the files can be copied to their final destinations. The make install command will copy the built program, and its libraries and documentation, to the correct locations.

+

This usually means that the program’s binary will be copied to a directory on your PATH, the program’s manual page will be copied to a directory on your MANPATH, and any other files it depends on will be safely stored in the appropriate place.

+

Since the install step is also defined in the Makefile, where the software is installed can change based on options passed to the configure script, or things the configure script discovered about your system.

+

Depending on where the software is being installed, you might need escalated permissions for this step so you can copy files to system directories. Using sudo will often do the trick.

[**Where do these scripts come from**](https://robots.thoughtbot.com/the-magic-behind-configure-make-make-install#where-do-these-scripts-come-from)

+

All of this works because a configure script examines your system, and uses the information it finds to convert a Makefile.in template into a Makefile, but where do the configure script and the Makefile.in template come from?

+

If you’ve ever opened up a configure script, or associated Makefile.in, you will have seen that they are thousands of lines of dense shell script. Sometimes these supporting scripts are longer than the source code of the program they install.

+

Even starting from an existing configure script, it would be very daunting to manually construct one. Don’t worry, though: these scripts aren’t built by hand.

+

Programs that are built in this way have usually been packaged using a suite of programs collectively referred to as [*autotools*](http://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html#Autotools-Introduction). This suite includes autoconf, automake, and many other programs, all of which work together to make the life of a software maintainer significantly easier. The end user doesn’t see these tools, but they take the pain out of setting up an install process that will run consistently on many different flavours of Unix.

[**Hello world**](https://robots.thoughtbot.com/the-magic-behind-configure-make-make-install#hello-world)

+

Let’s take a simple “Hello world” C program, and see what it would take to package it with autotools.

+

Here’s the source of the program, in a file called main.c:

+

#include <stdio.h>

int

main(int argc, char\* argv[])

{

printf("Hello world\n");

return 0;

}

[**Creating the configure script**](https://robots.thoughtbot.com/the-magic-behind-configure-make-make-install#creating-the-configure-script)

+

Instead of writing the configure script by hand, we need to create a configure.ac file written in m4sh—a combination of [m4](http://en.wikipedia.org/wiki/M4_%28computer_language%29) macros and POSIX shell script—to describe what the configure script needs to do.

+

The first m4 macro we need to call is [AC\_INIT](https://www.gnu.org/software/autoconf/manual/autoconf-2.69/html_node/Initializing-configure.html), which will initialise autoconf and set up some basic information about the program we’re packaging. The program is called helloworld, the version is 0.1, and the maintainer is george@thoughtbot.com:

+

AC\_INIT([helloworld], [0.1], [george@thoughtbot.com])

+

We’re going to use automake for this project, so we need to initialise that with the [AM\_INIT\_AUTOMAKE](http://www.gnu.org/software/automake/manual/html_node/Public-Macros.html) macro:

+

AM\_INIT\_AUTOMAKE

+

Next, we need to tell autoconf about the dependencies our configure script needs to look for. In this case, the configure script only needs to look for a C compiler. We can set this up using the [AC\_PROG\_CC](https://www.gnu.org/software/autoconf/manual/autoconf-2.69/html_node/C-Compiler.html) macro:

+

AC\_PROG\_CC

+

If there were other dependencies, then we’d use other m4 macros here to discover them; for example the [AC\_PATH\_PROG](http://www.gnu.org/software/autoconf/manual/autoconf-2.68/html_node/Generic-Programs.html) macro looks for a given program on the user’s PATH.

+

Now that we’ve listed our dependencies, we can use them. We saw earlier that a typical configure script will use the information it has about the user’s system to build a Makefile from a Makefile.in template.

+

The next line used the [AC\_CONFIG\_FILES](https://www.gnu.org/software/autoconf/manual/autoconf-2.69/html_node/Configuration-Files.html) macro to tell autoconf that the configure script should do just that: it should find a file called Makefile.in, substitute placeholders like @PACKAGE\_VERSION@ with values like 0.1, and write the results to Makefile.

+

AC\_CONFIG\_FILES([Makefile])

+

Finally, having told autoconf everything our configure script needs to do, we can call the [AC\_OUTPUT](http://www.gnu.org/software/autoconf/manual/autoconf-2.69/html_node/Output.html) macro to output the script:

+

AC\_OUTPUT

+

Here’s the whole thing. Not bad, compared to the 4,737 line configure script it’s going to produce!

+

AC\_INIT([helloworld], [0.1], [george@thoughtbot.com])

AM\_INIT\_AUTOMAKE

AC\_PROG\_CC

AC\_CONFIG\_FILES([Makefile])

AC\_OUTPUT

+

We’re almost ready to package up and distribute our program, but we’re still missing something. Our configure script will expect a Makefile.in file that it can substitute all of those system-specific variables into, but so far, we’ve not created that file.

[**Creating the Makefile**](https://robots.thoughtbot.com/the-magic-behind-configure-make-make-install#creating-the-makefile)

+

As with the configure script, the Makefile.in template is very long and complex. So instead of writing it by hand, we write a shorter Makefile.am file, which automake will use to generated the Makefile.in for us.

+

First, we need to set some options to tell automake about the layout of the project. Since we’re not following the standard layout of a GNU project, we warn automake that this is a [foreign](http://www.gnu.org/software/automake/manual/automake.html#List-of-Automake-options) project:

+

AUTOMAKE\_OPTIONS = foreign

+

Next, we tell automake that we want the Makefile to build a program called helloworld:

+

bin\_PROGRAMS = helloworld

+

There’s a lot of information packed into this line, thanks to automake’s [uniform naming scheme](http://www.gnu.org/software/automake/manual/html_node/Uniform.html).

+

The PROGRAMS suffix is called a *primary*. It tells automake what properties the helloworld file has. For example, PROGRAMS need to be built, whereas SCRIPTS and DATA files don’t need to be built.

+

The bin prefix tells automake that the file listed here should be installed to the directory defined by the variable bindir. There are various directories defined for us by autotools—including bindir, libdir, and pkglibdir—but we can also define our own.

+

If we wanted to install some Ruby scripts as part of our program, we could define a rubydir variable and tell automake to install our Ruby files there:

+

rubydir = $(datadir)/ruby

ruby\_DATA = my\_script.rb my\_other\_script.rb

+

Additional prefixes can be added before the install directory to further nuance automake’s behaviour.

+

Since we’ve defined a PROGRAM, we need to tell automake where to find its source files. In this case, the prefix is the name of the program these source files build, rather than the place where they will be installed:

+

helloworld\_SOURCES = main.c

+

Here’s the whole Makefile.am file for our helloworld program. As with the configure.ac and the configure script, it’s a lot shorter than the Makefile.in that it generates:

+

AUTOMAKE\_OPTIONS = foreign

bin\_PROGRAMS = helloworld

helloworld\_SOURCES = main.c

[**Putting it all together**](https://robots.thoughtbot.com/the-magic-behind-configure-make-make-install#putting-it-all-together)

+

Now we’ve written our config files, we can run autotools and generate the finished configure script and Makefile.in template.

+

First, we need to generate an m4 environment for autotools to use:

+

aclocal

+

Now we can run autoconf to turn our configure.ac into a configure script, and automake to turn our Makefile.am into a Makefile.in:

+

autoconf

automake --add-missing

[**Distributing the program**](https://robots.thoughtbot.com/the-magic-behind-configure-make-make-install#distributing-the-program)

+

The end user doesn’t need to see our autotools setup, so we can distribute the configure script and Makefile.in without all of the files we used to generate them.

+

Fortunately, autotools will help us with distribution too. The Makefile contains all kinds of interesting targets, including one to build a tarball of the project containing all of the files we need to distribute:

+

./configure

make dist

+

You can even test that the distribution tarball can be installed under a variety of conditions:

+

make distcheck

[**Overview**](https://robots.thoughtbot.com/the-magic-behind-configure-make-make-install#overview)

2

Now we know where this incantation comes from and how it works!

+

On the maintainer’s system:

+

aclocal # Set up an m4 environment

autoconf # Generate configure from configure.ac

automake --add-missing # Generate Makefile.in from Makefile.am

./configure # Generate Makefile from Makefile.in

make distcheck # Use Makefile to build and test a tarball to distribute

+

On the end-user’s system:

+

./configure # Generate Makefile from Makefile.in

make # Use Makefile to build the program

make install # Use Makefile to install the program