Using Autotools to Configure, Make, and Install a Program

7 minute read Updated: July 11, 2023


Kasper Siig

In this Series

Table of contents

We're Earthly. We make building software simpler and therefore faster using containerization. This article is about autotools and make. If you're interested in a different approach to building and packaging software then check us out.

Autotools is one of the most widely adopted code packaging and shipping tools available to developers on Linux. While there are alternatives, such as CMake, SCons, and BJam, they don't quite match Autotools in ease of use, power, and versatility.

At its base, Autotools can help make your application more portable, give it the versatility to be installed on many different systems, and can automatically procure scripts to check where elements are, like the compiler for your program. In this article, you will learn how to use Autotools to package up an application and ship it.

Components of Autotools

Autotools is made up of three unique components:

- `autoconf`
- `automake`
- `aclocal`

Using these tools, you will create two files, `configure` and `Makefile.in`. These files are present in any project shipped using Autotools and are usually quite large and complex. Luckily, you don't have to write them yourself—instead you'll be writing the files `configure.ac` and `Makefile.am`, which will automatically generate the files you need.

Autoconf

Autoconf is written in **M4sh**, using `m4` macros. If you've heard of this before, then great! If you haven't, no worries. `m4sh` provides some macros you can use when creating your `configure.ac` script and are part of why you can generate a massive `configure` script without having to write too much actual code.

The way it works is that you create a `configure.ac` script in which you define various settings like release name, version, which compiler to use, and where it should output files. Once you've written your script, you run it through `autoconf` to create your final `configure` script. The purpose of `autoconf` is to collect information from your system to populate the `Makefile.in` template, which is created using `automake`.

Automake

The `Makefile.am` script creates `Makefile.in`. The principles behind this are the same as with `configure.ac`: write a simple script in order to create a complex file. Automake is the component you'll use to create the Makefile, a template that can then be populated with `autoconf`.

Automake does so using variables and **primaries**. An example of such a primary is `bin_PROGRAMS = helloworld`, where the primary is the `_PROGRAMS` suffix. This

primary gives `automake` some knowledge about your program, like where you want the produced binary to be installed.

In this case, you're telling `automake` to install the `helloworld` binary in the path defined by the `bindir`. You may notice we didn't define a `bindir` variable, because that variable is built into `automake` and is typically the default binary directory of your system.

Other examples of primaries are `_SCRIPTS`, which you can use when you want a script, rather than a binary to be installed somewhere, and `_DATA`, when you have extra data files you want included in your installation. There are many more that you will find once you start using Autotools and figure out what your needs are.

One last thing to mention is that although `Makefile.in` is special in that it contains all of these primaries, it's still a regular Makefile. Meaning you can write your own custom make targets if you want. For example, if you want to have a custom `clean` target that deletes specific files, you can do so easily.

`Aclocal`

This is the smallest component in the Autotools suite, but it's very important. You learned in the previous section that `autoconf` uses `m4` macros to be configured. But where do these `m4` macros come from? They're generated by running the `aclocal` command. Simple as that. If you don't run `aclocal` before running `autoconf`, you'll get an error complaining about missing macros.

Using Autotools

Now that you know the basic principles of how the Autotools suite is put together, it's time to see it all in action and create a small C program that you can compile and ship.

Writing the Source Program

The first thing you need is the program you want to compile and ship. Autotools is compatible with many different projects and languages, but for this example

you'll be working in **C**, which is most commonly used. If you're not familiar with **C**, don't worry, it's very simple. Here's your sample code:

```
#include <stdio.h>

int
main(int argc, char* argv[])
{
  printf("Hello World\n");
  return 0;
}
```

As you can see, it simply includes a standard in/standard out library and then prints **Hello World\n**. Let's start with **autoconf** to configure this project.

Configuring `configure.ac`

When writing your **configure.ac** file, there are a lot of options to choose from. You can get very specific about how you want your script to be configured, but some configurations need to be set. The first of these is **AC_INIT**. This tells **autoconf** what the name of your application is, what version it is, and who's the maintainer. For this example, you'll write:

```
AC_INIT([helloworld], [0.1], [maintainer@example.com])
```

While **autoconf** is generally used alongside **automake**, it's not necessary, so you need to initialize that by writing:

```
AM_INIT_AUTOMAKE
```

Now that the generic options are initialized, you can get more specific with what you want. You need to specify what compiler you want the **configure** script to use. You do this by writing:

```
AC_PROG_CC
```

This will tell the **`configure`** script to look for a **`C`** **compiler**. For other applications, you may need more dependencies to build your program. By using the **`AC_PATH_PROG`** macro, you can make **`autoconf`** look for specific programs in a user's **`PATH`** . At this point, there are only two steps needed to finish your basic **`configure.ac`** script:

```
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

`AC_CONFIG_FILES` tells **`autoconf`** that it should find a file called **`Makefile.in`** and replace placeholders according to what we've specified. This can be things like version or maintainer. **`AC_OUTPUT`** is the last thing you want to put in your **`configure.ac`** script, as it tells **`autoconf`** to output the final **`configure`** script. In the end, your **`configure.ac`** file should contain the following:

```
AC_INIT([helloworld], [0.1], [maintainer@example.com])
AM_INIT_AUTOMAKE
AC_PROG_CC
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Making the Makefile

When using **`automake`** , you'll have to adhere to a set of standards. One of these is that source files for a project are located in the **`src`** folder. In this project, you have a single **`main.c`** file in our root directory, so you need to tell **`automake`** that:

```
AUTOMAKE_OPTIONS = foreign
```

You need to tell **`automake`** what you want your compiled binary to be called. In this case, you want it to be called **`helloworld`** , so write the following:

```
bin_PROGRAMS = helloworld
```

Only one thing left, and that is to tell `automake` what files are needed to compile your application. Do this by writing:

```
helloworld_SOURCES = main.c
```

Notice how the first part is the name of your application followed by the `SOURCES` primary. Now `automake` knows all that it needs to know, and your `Makefile.am` is ready to use.

Creating Final Scripts

Once you've written your `configure.ac` and `Makefile.am`, it's relatively straightforward to distribute your application. Remember to start by running `aclocal` so you can run `autoconf`. Once you've run `autoconf`, you can run `automake --add-missing` to build your `Makefile.in`.

The reason for the `--add-missing` flag is to tell `automake` to automatically generate all of the additional files required, as usually you need more than just `Makefile.am` and would have to manually enter in the other files.

At this point, you have all you need to distribute your program. Before moving on on, here's a short recap showing the commands you should've run by now:

```
aclocal
autoconf
automake --add-missing
```

Distributing the Program

Distributing your application can seem like a daunting task, but Autotools makes it super easy. All you have to do is run `make dist` after you've run the configure scripts above. This will produce a tarball, which you can then ship to your customers.

Conclusion

Now you're able to use Autotools to compile and distribute your application, and you're able to do it in a way that ensures it's portable across a variety of systems. From here, you can start looking into automating this procedure and other ways to integrate Autotools directly into your daily development.

The great advantage of using something like Autotools is that you'll be using a system that has been in place for many years, is well-documented, and widely used. Many developers are comfortable installing applications using what Autotools produces, so it can make your application much more familiar and accessible.

If you are looking for a solution to avoid the complexities of Makefile, check out **Earthly**. **Earthly** takes the best ideas from Makefile and Dockerfile, and provides understandable and repeatable build scripts, minus the head-scratching parts of the Makefile.

Kasper Siig

As a DevOps engineer, Kasper Siig is used to working with a variety of exciting technologies, from automating simple tasks to CI/CD to Docker.

Updated: July 11, 2023

Published: June 8, 2021

You may also enjoy

A biased take on the ROI of fast
9 minute read

Fast builds with Earthly can lead to significant cost savings in CI/CD infrastructure and increased developer productivity, resulting in a 13X greater value....

Introducing Earthly: build automation for the container era
3 minute read

Introducing Earthly, a build automation tool for the container era. Learn how Earthly brings modern capabilities like reproducibility, determinism, and paral...

Can We Build Better?
4 minute read

Learn how to solve the problem of reproducible builds with Earthly, an open-source tool that encapsulates your build process in a Docker-like syntax. With Ea...

The Platform Values of Earthly
11 minute read

Learn about the platform values of Earthly, a new approach to build automation. Discover the principles that guide Earthly’s design, including versatility, a...

Earthly used by Phoenix Project
less than 1 minute read

Learn how Earthly is revolutionizing the CI pipeline for the popular Phoenix project, making testing and continuous integration easier than ever. Discover ho...

Better Together - Earthly + Github Actions

15 minute read

Learn how Earthly and Github Actions can work together to improve your Continuous Integration (CI) process. Discover the benefits of Earthly's local CI pipel...

What makes Earthly fast

13 minute read

Earthly makes CI/CD builds faster by reusing computation from previous runs for unchanged parts of the build. It is particularly effective in speeding up CI ...

Better Dependency Management in Python

6 minute read

Learn how Earthly can simplify dependency management in Python projects, ensuring consistency across different environments and streamlining the build and de...