Building GCC from source

Darren Smith
Jun 2, 2019 · 9 min read

In this guide we are going to walk through the steps required to build GCC 9.1.0 from source. Additionally we shall see how the sequence of commands can be collected into a single build script, making the whole process easily repeatable. The entire build script is available via:

> *https://github.com/darrenjs/howto/blob/master/build_scripts/build_gcc_9.sh*

GCC is the de facto compiler for GNU/Linux. This means that some version of GCC comes already installed with the operating system and is used to compile core components such as kernel modules, programs and libraries. Any C/C++ developer working on GNU/Linux will already be familiar with the system compiler; it's what gets invoked when you enter commands like *gcc* and *g++*.

It's great that GNU/Linux comes with GCC installed, however this has a down-side. The system version typically trails behind the latest version of GCC, often being many years out of date. This is illustrated in the table below; the leading professional distributions, Redhat & Suse, are shipped with quite old versions of GCC (as of early 2019), ones that barely offer access to C++11, let alone C++17:

- Redhat/Centos 7 — GCC 4.8 (experimental C++11)
- Redhat/Centos 6 — GCC 4.4 (C++03)
- Suse 12 — GCC 4.8 (experimental C++11)
- Suse 11 — GCC 4.3 (C++03)
- Ubuntu 18.04 — GCC 7.4 (full C++14; experimental C++17)

To check your current GCC version, do `gcc --version`

So if we want access to the latest C++14, C++17, C++2x language features and compiler improvements, how can we install a more up-to-date GCC?

In a corporate setting you could turn to a system administrator, but they may only be able to help if an official RPM or DEB package is available, which often there won't be. Or you could pay Redhat / Suse for so-called developer extensions, but again the GCC version they offer might not be the latest.

A better approach is to take matters into your own hands. GCC is open source software, so you can download the source code, configure & build, and install into a directory of your choice. No financial cost; no root permissions or privileges required; you can choose the compiler options most suitable to your needs; and you learn something about compiler build systems on the way.

This is what we will look at in this guide. We will go through the sequence of steps required to install GCC by building from source code, without having to rely on the system package manager. The whole procedure is also written up as a bash script, for which this guide serves as documentation. If you are in a hurry to build the latest GCC, you can skip this guide and run the build script directly.

Tip — review the release notes for each release. In particular, GCC error messages have steadily approved over time, with an additional improvements in GCC 9.

Overview

Building GCC from source involves the following sequence of steps:

1. choose a GCC version (and version of dependencies)

2. obtain source tarballs and unpack into appropriate directories
3. create a clean build environment
4. configure the source code
5. compile the source code
6. install the built artefacts
7. usage

We will go through each step in turn and refer to the corresponding code in the build script.

Choosing a GCC version

Usually we want to build the latest version of GCC (which is 9.1.0 as of mid 2019) so that we can access the latest C/C++ language features and compiler improvements. However the approach described here generally applies to earlier versions of GCC, and will likely work for later ones also.

Tip — get alerted to new GCC versions by signing up for announcements.

Early on in the build script a variable is defined which stores the version of GCC we are going to build:

```
gcc_version=9.1.0
```

This variable is used to select which GCC source code tarball to download. The version number is also embedded into the names of temporary and installation directories, allowing several different versions of GCC to be installed side-by-side.

Versions of dependencies

GCC depends on several other open source projects. These are typically support libraries, packaged separately to the GCC source code, that must also be downloaded and built. The list of required projects and recommended minimum versions can be found on the GCC prerequisites web page, although the version numbers listed there are often just a guide; in practice it might be required to bump the numbers slightly to get everything successfully working together.

In the build script each dependency and the version to use is defined as a variable:

```
gmp_version=6.1.2
mpfr_version=3.1.4
mpc_version=1.0.3
isl_version=0.18
```

You may wonder: *can I just use yum/yast to install these packages?* — No, is the general answer. The libraries that come shipped with your operating system will unlikely match the version requirements for the GCC you are trying to build.

Download

Each source code package should be obtained from a valid distribution origin, ideally from its project website. The build script defines a helper function called *__wget* which performs the download using the *wget* GNU/Linux utility; it is called for each package in turn:

```
__wget https://gmplib.org/download/gmp
$gmp_tarfile
__wget https://ftp.gnu.org/gnu/mpfr
$mpfr_tarfile
__wget http://www.multiprecision.org/downloads
$mpc_tarfile
__wget ftp://gcc.gnu.org/pub/gcc/infrastructure
$isl_tarfile
__wget ftp://ftp.gnu.org/gnu/gcc/gcc-${gcc_version}
$gcc_tarfile
```

Unpack

GCC's build system has a helpful feature for building these dependencies. If the source code for a dependency is placed within the GCC source code folder, it will be automatically discovered and compiled during the main build of GCC. This is the approach recommended by the GCC website and adopted in the build script (an alternative approach is to manually build and install each dependency separately and then configure GCC to find them).

The build script defines a utility function (*__untar*) to open a tarfile and then move its contents into the GCC source code directory. The following example is repeated for each dependency:

```
__untar "$source_dir/gcc-${gcc_version}"
"$tarfile_dir/$mpfr_tarfile"

mv -v $source_dir/gcc-${gcc_version}/mpfr-
${mpfr_version} $source_dir/gcc-${gcc_version}/mpfr
```

Clean your shell

An important consideration when building any C/C++ project is to clean and control the shell environment. Environment variables present during compilation can have significant side effects on the build process, possibly causing compile or link failures, and later run-time errors when the built artefacts are used. So before proceeding to the configure and build steps we must first obtain a clean shell environment.

There are several ways to create a clean environment. The simplest is run the command: `env -i bash -norc`. This creates a new shell with minimum variables, although possibly too minimal for compiling software.

Alternatively we can loop over and unset each variable found in the current shell; this is the approach taken in the build script, performed by the following code:

```
for i in $(env | awk -F"=" '{print $1}') ; do
  unset $i || true # ignore unset fails
done
```

Another approach, more commonly encountered in corporate settings, is to create a user account dedicated for the purpose of compiling software, i.e. a *build* user; the point is that production builds should not be undertaken from a developer's own account, where the shell environment may be uncertain and inconsistent between successive builds.

Configuration

After the source code packages have been unzipped they must next be configured.

Configuration allows options to be applied to the source code in order to customise the compiler and other tools that are built. For example, we can specify the set of programming languages that GCC should support; what compiler tuning options to enable; whether to create static and shared libraries; and also the location where the final GCC artefacts get installed. GCC is a broad compiler platform, with many options and several languages, so please consult the GCC website for the full list of options, including advanced options for language and/or platform tuning.

The options chosen in the build script are typical ones for C/C++ projects, and only those two languages are enabled.

A key configuration choice is the final installation location. This is the directory where the built artefacts — the compiler binary, libraries, include files, man pages etc. — will be copied to after the build completes. This must be somewhere where you have write access, for

example it could be under your home or team directory.

The build script defaults this to be a path under the user's home directory, with the GCC version number appearing as part of directory name. Placing the version number in the path name allows multiple versions to be installed side-by-side.

```
install_dir=${HOME}/opt/gcc-${gcc_version}
```

It is the configuration step that usually causes the most trouble. Problems typically relate to missing system build tools and libraries. In such cases you may need to get missing packages installed via your system package manager.

Build

After a successful configure step we are now ready to build. Ensure you have around 5 GB of disk space free. Prefer to build on a local disk rather than a network share, for faster completion.

We next choose whether to proceed with a parallel build, and if so how many jobs to run concurrently. A parallel build will complete sooner but can drastically eat up system resources, causing all other running programs to behave sluggishly. A conservative choice is to use two CPUs, by proving the option *-j 2* to make.

The build script uses a variable to control this setting; by default it is commented out, resulting in a single threaded build:

```
# make_flags="-j 2"
```

The build is started by invoking *make*. It can take up to several hours to finish.

Install

After the build and optional checks have successfully completed, the next step is to install the built artefacts into their destination location. This location is specified via the `--prefix` option provided during the configuration step.

The script performs installation by running `make install`

Usage

Assuming no errors, the compiler is now installed and can be invoked by providing its full path. For example, if it was installed at ~/opt/gcc-9.1.0 we can run:

```
$ ~/opt/gcc-9.1.0/bin/g++
```

The system now has two versions of GCC available (the other being the system compiler), and this can present challenges when using the new version.

Problems arise if the two are accidentally mixed. For example,

compiling a program with the new compiler but linking at run-time to the shared libraries belonging to the other. It is very easy to do this accidentally.

Here is an illustration of this problem, showing a mix up between GCC 9.1.0 and the system compiler on Centos 7 (GCC 4.8.5). We start with a C++17 source file which we compile with the new compiler:

```
centos7> ~/opt/gcc-9.1.0/bin/g++ --std=c++17 demo.cc
-o demo
```

The compilation succeeds, producing a binary file named *demo*. However running this binary throws up a dynamic linking error:

```
centos7> ./demo

./demo: /lib64/libstdc++.so.6: version `CXXABI_1.3.9'
not found (required by ./demo)

./demo: /lib64/libstdc++.so.6: version
`GLIBCXX_3.4.21' not found (required by ./demo)
```

This error is caused by the demo program linking, at run-time, to the system C++ library (located under the system directory /lib64). It links to that particular library because it is the only candidate library found when the program is invoked. And because that library is not compatible with the version of C++ used during compilation, the

program fails to start.

There are a few approaches to resolving such errors.

We can modify the set of paths the dynamic linker uses to search for candidates libraries, so that it finds the libraries belonging to the new compiler instead of the system libraries. This is done by setting the environment variable LD_LIBRARY_PATH to the library folder under the GCC 9.1.0 installation. This change must be made before running the binary:

```
centos7> export LD_LIBRARY_PATH=$HOME/opt/gcc-
9.1.0/lib64

centos7> ./demo
```

An alternative mechanism for providing library search paths is RPATH. This is a compile time approach, allowing a compiled binary to internally store a set of paths that will be searched for run-time libraries:

```
centos7> ~/opt/gcc-9.1.0/bin/g++ -Wl,-
rpath=$HOME/opt/gcc-9.1.0/lib64 --std=c++17 demo.cc -
o demo
```

Another approach is to link the GCC libraries statically rather than dynamically, so that they don't need to be loaded at run-time; instead

they are combined with the final binary:

```
centos7> ~/opt/gcc-9.1.0/bin/g++ --std=c++17 demo.cc
-o demo -static-libstdc++ -static-libgcc
```

During the GCC installation step the build script creates an environment helper script named *activate,* placed at the root of the installation location. This script can be *sourced* to activate the compiler, and resolve dynamic linking issues, for the current user shell. It does this by making the following changes to the current shell:

- update PATH to find the new compiler binaries
- update LD_LIBRARY_PATH to find the new libraries
- update MANPATH to find the new man-pages

The helper script must be sourced to affect your shell:

```
$ source $HOME/opt/gcc-9.1.0/activate
```

The man pages of the new compiler can now be accessed, and running the compiler no longer needs specifying the full installation path:

```
$ man g++

$ g++ --std=c++17 demo.cc -o demo

$ ./demo
```

Summary

If you have followed these steps, or just run the build script, you should now have a production ready build of the latest version of GCC, and with it access to the latest C++ 17/2x language features.

You have also been introduced to the basics of the GCC build process, so building the next version of GCC should be a relatively simple task of re-running these steps, with just a few minor changes (e.g., change the gcc_version and possibly the dependency versions).

And because the entire process can be scripted, you can try building GCC with different options and install them alongside each other. Different tuning options can be explored, which is particularly important when aiming for software optimised for some constraint, e.g., a specific CPU instruction set, or fastest run-time speed.

Programming Gcc Compilers Linux C