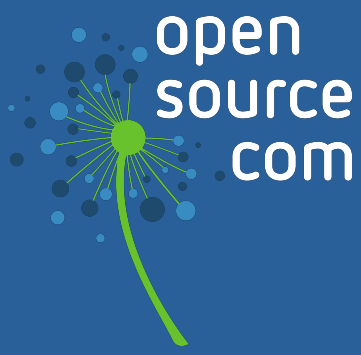


An open source developer's guide to building applications

by J. Huttanagoudar, S. Kenlon, S. Avenwedde

We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com

Supported by
Red Hat

Table of Contents

Jayashree Huttanagoudar

Jayashree Huttanagoudar is a Senior Software Engineer at Red Hat. She works with the Middleware OpenJDK team. She is always curious to learn new things, and enjoys writing about the seemingly infinite aspects of programming.

Steven Avenwedde

Stephan is a technology enthusiast who appreciates open source for the deep insight of how things work. Stephan works as a full time support engineer in the mostly proprietary area of industrial automation software. If possible, he works on his Python-based open source projects, writing articles, or driving motorbike.

Seth Kenlon

Seth Kenlon is a UNIX geek and free culture advocate. He has worked in the film and computing industry, often at the same time. He is a writer, editor, and community advocate for Opensource.com.

Anyone can compile open source code in these three simple steps

By Seth Kenlon

There are many ways to install software, but you get an option not available elsewhere with open source: You can compile the code yourself. This is the classic three-step process to compile source code:

```
$ ./configure
$ make
$ sudo make install
```

Thanks to these commands, you might be surprised to find that you don't need to know how to write or even read code to compile it.

Install commands to build software

As this is your first time compiling, there's a one-time preparatory step to install the commands for building software. Specifically, you need a compiler. A compiler, such as GCC or LLVM, turns source code that looks like this—

```
#include <iostream>

using namespace std;

int main() {
  cout << "hello world";
}
```

—into *machine language*, the instructions that a CPU uses to process information. You can look at machine code, but it wouldn't make any sense to you (unless you're a CPU).

You can get the GNU C compiler (GCC) and the LLVM compiler, along with other essential commands for compiling on Fedora, CentOS, Mageia, and similar distributions, using your package manager:

```
$ sudo dnf install @development clang
```

On Debian, Elementary, Mint, and similar distributions:

```
$ sudo apt install build-essential clang
```

With your system set up, there are a few tasks that you'll repeat each time you want to compile your software:

1. Download the source code
2. Unarchive the source code
3. Compile

You have all the commands you need, so now you need some software to compile.

1. Download source code

Obtaining source code for an application is much like getting any downloadable software. You go to a website or a code management site like GitLab, SourceForge, or GitHub. Typically, open source software is available in both a work-in-progress ("current" or "nightly") form as well as a packaged "stable" release version. Use the stable version when possible unless you have reason to believe otherwise or are good enough with code to fix things when they break. The term **stable** suggests the code got tested and that the programmers of the application feel confident enough in the code to package it into a `.zip` or `.tar` archive, give it an official number and sometimes a release name, and offer it for download to the general non-programmer public.

For this exercise, I'm using Angband, an open source (GPLv2) ASCII dungeon crawler. It's a simple application with just enough complications to demonstrate what you need to consider when compiling software for yourself.

Download the source code from the website.

2. Unarchive the source code

Source code is often delivered as an archive because source code usually consists of multiple files. You have to extract it before interacting with it, whether it's a tarball, a zip file, a 7z file, or something else entirely.

```
$ tar --extract --file Angband-x.y.z.tar.gz
```

Once you've unarchived it, change the directory into the extracted directory and have a look around. There's usually a `README` file at the top level of the directory. This file, ideally, contains guidance on what you need to do to compile the code. The `README` often contains information on these important aspects of the code:

- **Language:** What language the code is in (for instance, C, C++, Rust, Python).
- **Dependencies:** What other software you need to have installed on your system for this application to build and run.
- **Instructions:** The literal steps you need to take to build the software. Occasionally, they include this information within a dedicated file intuitively entitled `INSTALL`.

If the `README` file doesn't contain that information, consider filing a bug report with the developer. You're not the only one who needs an introduction to source code. Regardless of how experienced they are, everyone is new to source code they've never seen before, and documentation is important!

Angband's maintainers link to online instructions to describe how to compile the code. This document also describes what other software you need to have installed, although it doesn't exactly spell it out. The site says, "There are several different front ends that you can optionally build (GCU, SDL, SDL2, and X11) using arguments to configure such as `--enable-sdl`, `--disable-x11`, etc." This may mean something to you or look like a foreign language, but this is the kind of stuff you get used to after compiling code frequently. Whether or not you understand what X11 or SDL2 is, they're both requirements that you see pretty often after regularly compiling code over a few months. You get comfortable with the idea that most software needs other software libraries because they build upon other technologies. In this case, though, Angband is very flexible and compiles with or without these optional dependencies, so for now, you can pretend that there are no additional dependencies.

3. Compile the code

The canonical steps to build code are:

```
$ ./configure
$ make
$ sudo make install
```

Those are the steps for projects built with Autotools, which is a framework created to standardize how source code is delivered. Other frameworks (such as Cmake) exist, however, and they require different steps. When projects stray from Autotools or Cmake, they tend to warn you in the `README` file.

Configure

Angband uses Autotools, so it's time to compile code! In the Angband directory, first, run the configuration script included with the source:

```
$ ./configure
```

This step scans your system to find the dependencies that Angband requires to build correctly. Some dependencies are so basic that your computer wouldn't be running without them, while others are specialized. At the end of the process, the script gives you a report on what it has found:

```
[...]
configure: creating ./config.status
config.status: creating mk/buildsys.mk
config.status: creating mk/extra.mk
config.status: creating src/autoconf.h

Configuration:

  Install path:    /usr/local
  binary path:     /usr/local/games
  config path:     /usr/local/etc/angband/
  lib path:        /usr/local/share/angband/
  doc path:        /usr/local/share/doc/angband/
  var path:        (not used)
  (save and score files in ~/.angband/Angband/)

-- Frontends --
- Curses            Yes
```

```
- X11               Yes
- SDL2              Disabled
- SDL               Disabled
- Windows           Disabled
- Test              No
- Stats             No
- Spoilers          Yes

- SDL2 sound        Disabled
- SDL sound         Disabled
```

Some of that output may make sense to you, some of it may not. Either way, you probably notice that SDL2 and SDL are marked as `Disabled`, and both Test and Stats are marked with `No`. Although negative, this isn't necessarily a bad thing. This, essentially, is the difference between a **Warning** and an **Error**. Had the configure script encountered something that would prevent it from building the code, it would have alerted you with an error.

If you want to optimize your build a little, you can choose to resolve these negative messages. By searching through the Angband documentation, you might decide that Test and Stats aren't actually of interest to you (they're developer options, specific to Angband). However, with a little online research, you might discover that SDL2 would be a nice feature to have.

To resolve a dependency when compiling code, you need to install the missing component and the *development libraries* for that missing component. In other words, Angband needs SDL2 to play sound, but it needs `SDL2-devel` (called `libsdl2-dev`, on Debian systems) to build. Install both with your package manager:

```
$ sudo dnf install sdl2 sdl2-devel
```

Try the configuration script again:

```
$ ./configure --enable-sdl2
[...]
Configuration:
[...]
- Curses                                Yes
- X11                                   Yes
- SDL2                                  Yes
- SDL                                   Disabled
- Windows                               Disabled
- Test                                  No
- Stats                                 No
```

```
- Spoilers                                  Yes

- SDL sound                                 Disabled
- SDL2 sound                                Yes
```

Make

Once everything's configured, run the `make` command:

```
$ make
```

This usually takes a while, but it provides lots of visual feedback, so you'll know code is getting compiled.

Install

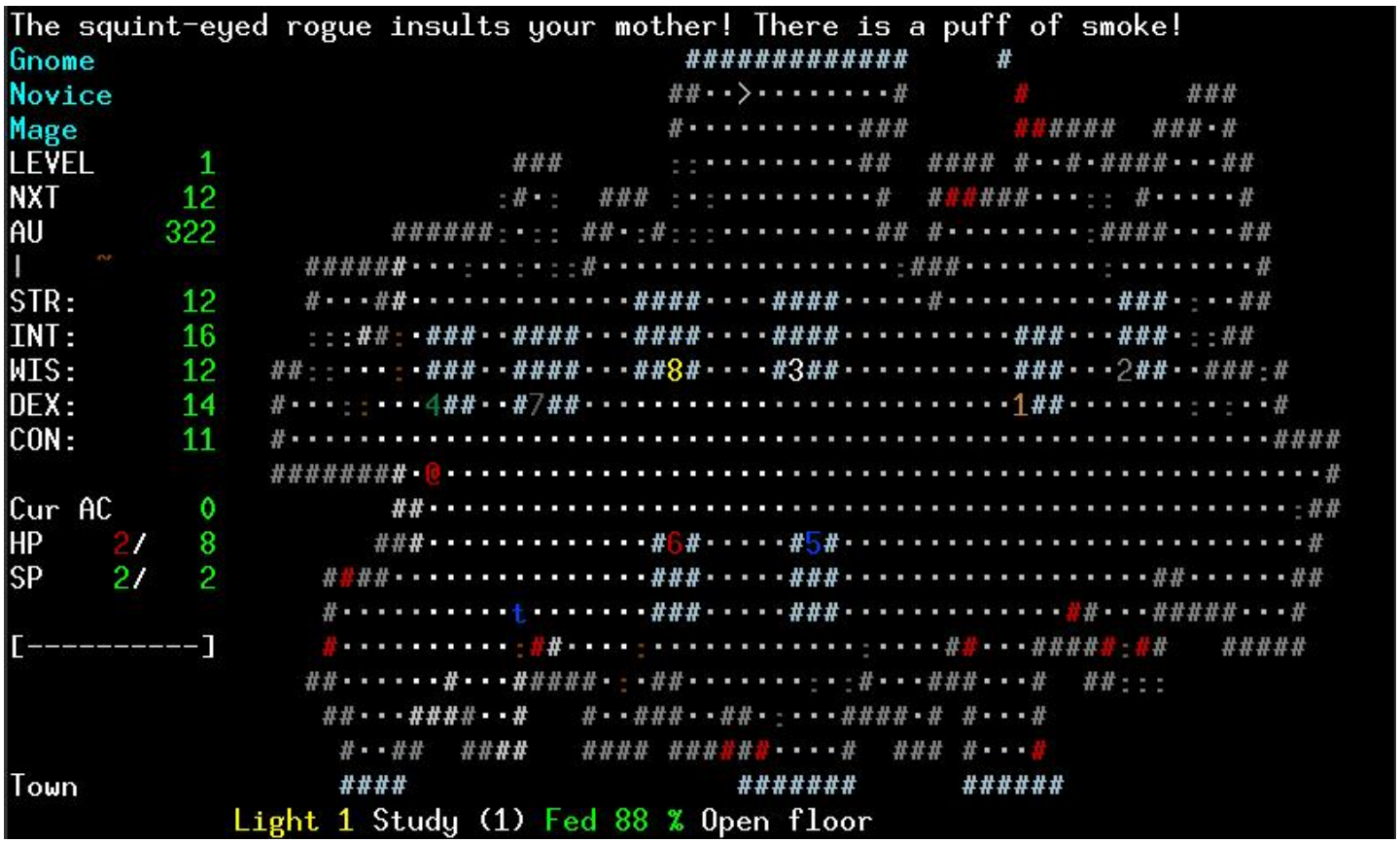
The final step is to install the code you've just compiled. There's nothing magical about installing code. All that happens is that lots of files get copied to very specific directories. That's true whether you're compiling from source code or running a fancy graphical install wizard. Because the code is getting copied to system-level directories, you must have root (administrative) privileges, which get granted by the `sudo` command.

```
$ sudo make install
```

Run the application

Once the application gets installed, you can run it. According to the Angband documentation, the command to start the game is `angband`, so try it out:

```
$ angband
```



(Seth Kenlon, CC BY-SA 4.0)

Compiling code

I compile most of my own applications, whether on my Slackware desktop computer or my CentOS laptop using NetBSD's pkgsrc system. I find that by compiling software myself, I can be as particular as I want to be about the features included in the application, how it's configured, which library version it uses, and so on. It's rewarding, and it helps me keep up to date with new releases and, because I sometimes find bugs in the process, it helps me get involved with lots of different open source projects.

It's rare that you have no other option but to compile software. Most open source projects provide both the source code (that's why it's called "open source") and installable packages. Compiling from source code is a choice you get to make for yourself, maybe because you want new features not yet available in the latest release or just because you prefer to compile code yourself.

Homework

Angband can use either Autotools or Cmake, so if you want to experience another way of building code, try this:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

You can also try compiling with the LLVM compiler instead of the GNU C compiler. For now, I'll leave that as an exercise for you to investigate on your own (hint: try setting the CC environment variable).

Once you finish exploring the source code of Angband and at least a few of its dungeons (you've earned some downtime), have a look at some other codebases. Many will use Autotools or Cmake, while others may use something different. See what you can build!

A programmer's guide to GNU C Compiler

By Jayashree Huttanagoudar

C is a well-known programming language, popular with experienced and new programmers alike. Source code written in C uses standard English terms, so it's considered human-readable. However, computers only understand binary code. To convert code into machine language, you use a tool called a *compiler*.

A very common compiler is GCC (GNU C Compiler). The compilation process involves several intermediate steps and adjacent tools.

Install GCC

To confirm whether GCC is already installed on your system, use the `gcc` command:

```
$ gcc --version
```

If necessary, install GCC using your packaging manager. On Fedora-based systems, use `dnf`:

```
$ sudo dnf install gcc libgcc
```

On Debian-based systems, use `apt`:

```
$ sudo apt install build-essential
```

After installation, if you want to check where GCC is installed, then use:

```
$ whereis gcc
```

Simple C program using GCC

Here's a simple C program to demonstrate how to compile code using GCC. Open your favorite text editor and paste in this code:

```
// hellogcc.c
#include <stdio.h>

int main() {
        printf("Hello, GCC!\n");
return 0;
}
```

Save the file as `hellogcc.c` and then compile it:

```
$ ls
hellogcc.c

$ gcc hellogcc.c

$ ls -1
a.out
hellogcc.c
```

As you can see, `a.out` is the default executable generated as a result of compilation. To see the output of your newly compiled application, just run it as you would any local binary:

```
$ ./a.out
Hello, GCC!
```

Name the output file

The filename `a.out` isn't very descriptive, so if you want to give a specific name to your executable file, you can use the `-o` option:

```
$ gcc -o hellogcc hellogcc.c

$ ls
a.out  hellogcc  hellogcc.c

$ ./hellogcc
Hello, GCC!
```

This option is useful when developing a large application that needs to compile multiple C source files.

Intermediate steps in GCC compilation

There are actually four steps to compiling, even though GCC performs them automatically in simple use-cases.

1. Pre-Processing: The GNU C Preprocessor (`cpp`) parses the headers (**#include** statements), expands macros (**#define** statements), and generates an intermediate file such as `hellogcc.i` with expanded source code.

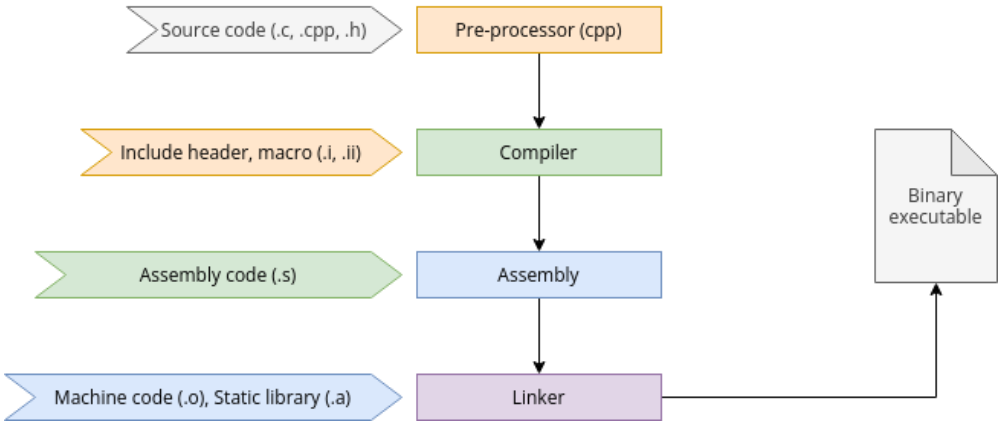
2. Compilation: During this stage, the compiler converts pre-processed source code into assembly code for a specific CPU architecture. The resulting assembly file is named with a `.s` extension, such as `hellogcc.s` in this example.

3. Assembly: The assembler (`as`) converts the assembly code into machine code in an object file, such as `hellogcc.o`.

4. Linking: The linker (`ld`) links the object code with the library code to produce an executable file, such as `hellogcc`.

When running GCC, use the `-v` option to see each step in detail.

```
$ gcc -v -o hellogcc hellogcc.c
```



(Jayashree Huttanagoudar, CC BY-SA 4.0)

Manually compile code

It can be useful to experience each step of compilation because, under some circumstances, you don't need GCC to go through all the steps. First, delete the files generated by GCC in the current folder, except the source file.

```
$ rm a.out hellogcc.o

$ ls
hellogcc.c
```

Pre-processor

First, start the pre-processor, redirecting its output to `hellogcc.i`:

```
$ cpp hellogcc.c > hellogcc.i

$ ls
hellogcc.c  hellogcc.i
```

Take a look at the output file and notice how the pre-processor has included the headers and expanded the macros.

Compiler

Now you can compile the code into assembly. Use the `-S` option to set GCC just to produce assembly code.

```
$ gcc -S hellogcc.i

$ ls
hellogcc.c  hellogcc.i  hellogcc.s

$ cat hellogcc.s
```

Take a look at the assembly code to see what's been generated.

Assembly

Use the assembly code you've just generated to create an object file:

```
$ as -o hellogcc.o hellogcc.s

$ ls
hellogcc.c  hellogcc.i  hellogcc.o  hellogcc.s
```

Linking

To produce an executable file, you must link the object file to the libraries it depends on. This isn't quite as easy as the previous steps, but it's educational:

```
$ ld -o hellogcc hellogcc.o
ld: warning: cannot find entry symbol _start; defaulting to 0000000000401000
ld: hellogcc.o: in function `main`:
hellogcc.c:(.text+0xa): undefined reference to `puts'
```

An error referencing an `undefined puts` occurs after the linker is done looking at the `libc.so` library. You must find suitable linker options to link the required libraries to resolve this. This is no small feat, and it's dependent on how your system is laid out.

When linking, you must link code to core runtime (CRT) objects, a set of subroutines that help binary executables launch. The linker also needs to know where to find important system libraries, including libc and libgcc, notably within special start and end instructions. These instructions can be delimited by the `--start-group` and `--end-group` options or using paths to `crtbegin.o` and `crtend.o`.

This example uses paths as they appear on a RHEL 8 install, so you may need to adapt the paths depending on your system.

```
$ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 \
-o hello /usr/lib64/crt1.o /usr/lib64/crti.o \
--start-group -L/usr/lib/gcc/x86_64-redhat-linux/8 \
-L/usr/lib64 -L/lib64 hello.o -lgcc \
--as-needed -lgcc_s \
--no-as-needed -lc -lgcc \
--end-group
/usr/lib64/crtn.o
```

The same linker procedure on Slackware uses a different set of paths, but you can see the similarity in the process:

```
$ ld -static -o hello -L/usr/lib64/gcc/x86_64-slackware-linux/11.2.0/ \
/usr/lib64/crt1.o /usr/lib64/crti.o hello.o /usr/lib64/crtn.o \
--start-group -lc -lgcc -lgcc_eh \
--end-group
```

Now run the resulting executable:

```
$ ./hello
Hello, GCC!
```

Some helpful utilities

Below are a few utilities that help examine the file type, symbol table, and the libraries linked with the executable.

Use the `file` utility to determine the type of file:

```
$ file hellogcc.c
hellogcc.c: C source, ASCII text

$ file hellogcc.o
hellogcc.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

$ file hellogcc
hellogcc: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=bb76b241d7d00871806e9fa5e814fee276d5bd1a, for GNU/Linux 3.2.0, not
stripped
```

The use the `nm` utility to list symbol tables for object files:

```
$ nm hellogcc.o
0000000000000000 T main
                         U puts
```

Use the `ldd` utility to list dynamic link libraries:

```
$ ldd hellogcc
linux-vdso.so.1 (0x00007ffe3bdd7000)
libc.so.6 => /lib64/libc.so.6 (0x00007f223395e000)
/lib64/ld-linux-x86-64.so.2 (0x00007f2233b7e000)
```

Wrap up

In this article, you learned the various intermediate steps in GCC compilation and the utilities to examine the file type, symbol table, and libraries linked with an executable. The next time you use GCC, you'll understand the steps it takes to produce a binary file for you, and when something goes wrong, you know how to step through the process to resolve problems.

How dynamic linking for modular libraries works on Linux

By Jayashree Huttanagoudar

When you write an application using the C programming language, your code usually has multiple source files.

Ultimately, these files must be compiled into a single executable. You can do this by creating either *static* or *dynamic* libraries (the latter are also referred to as *shared* libraries). These two types of libraries vary in how they are created and linked. Both have advantages and disadvantages, depending on your use case.

Dynamic linking is the most common method, especially on Linux systems. Dynamic linking keeps libraries modular, so just one library can be shared between any number of applications. Modularity also allows a shared library to be updated independently of the applications that rely upon it.

In this article, I demonstrate how dynamic linking works.

Linker

A linker is a command that combines several pieces of a program together and reorganizes the memory allocation for them.

The functions of a linker include:

- Integrating all the pieces of a program
- Figuring out a new memory organization so that all the pieces fit together
- Reviving addresses so that the program can run under the new memory organization
- Resolving symbolic references

As a result of all these linker functionalities, a runnable program called an *executable* is created. Before you can create a dynamically linked executable, you need some libraries to link *to* and an application to compile. Get your favorite text editor ready and follow along.

Create the object files

First, create the header file `mymath.h` with these function signatures:

```
int add(int a, int b);
int sub(int a, int b);
int mult(int a, int b);
int divi(int a, int b);
```

Create `add.c`, `sub.c` , `mult.c` and `divi.c` with these function definitions. I'm placing all of the code in one code block, so divide it up among four files, as indicated in the comments:

```
// add.c
int add(int a, int b){
return (a+b);
}

//sub.c
int sub(int a, int b){
return (a-b);
}

//mult.c
int mult(int a, int b){
return (a*b);
}

//divi.c
int divi(int a, int b){
return (a/b);
}
```

Now generate object files `add.o`, `sub.o`, `mult.o`, and `divi.o` using GCC:

```
$ gcc -c add.c sub.c mult.c divi.c
```

The `-c` option skips the linking step and creates only object files.

Creating a shared object file

Dynamic libraries are linked during the execution of the final executable. Only the name of the dynamic library is placed in the final executable. The actual linking happens during runtime, when both executable and library are placed in the main memory.

In addition to being sharable, another advantage of a dynamic library is that it reduces the size of the final executable file. Instead of having a redundant copy of the library, an application using a library includes only the name of the library when the final executable is created.

You can create dynamic libraries from your existing sample code:

```
$ gcc -Wall -fPIC -c add.c sub.c mult.c divi.c
```

The option `-fPIC` tells GCC to generate *position-independent code* (PIC). The `-Wall` option isn't necessary and has nothing to do with how the code is compiling. Still, it's a valuable option because it enables compiler warnings, which can be helpful when troubleshooting.

Using GCC, create the shared library `libmymath.so`:

```
$ gcc -shared -o libmymath.so add.o sub.o mult.o divi.o
```

You have now created a simple example math library, `libmymath.so`, which you can use in C code. There are, of course, very complex C libraries out there, and this is the process their developers use to generate the final product that you or I install for use in C code.

Next, you can use your new math library in some custom code, then link it.

Creating a dynamically linked executable

Suppose you've written a command for mathematics. Create a file called `mathDemo.c` and paste this code into it:

```
#include <mymath.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int x, y;
  printf("Enter two numbers\n");
  scanf("%d%d",&x,&y);

  printf("\n%d + %d = %d", x, y, add(x, y));
  printf("\n%d - %d = %d", x, y, sub(x, y));
  printf("\n%d * %d = %d", x, y, mult(x, y));

  if(y==0){
    printf("\nDenominator is zero so can't perform division\n");
      exit(0);
  }else{
      printf("\n%d / %d = %d\n", x, y, divi(x, y));
      return 0;
  }
}
```

Notice that the first line is an `include` statement referencing, by name, your own `libmymath` library. To use a shared library, you must have it installed. If you don't install the library you use, then when your executable runs and searches for the included library, it won't be able to find it. Should you need to compile code without installing a library to a known directory, there are ways to override default settings. For general use, however, it's expected that libraries exist in known locations, so that's what I'm demonstrating here.

Copy the file `libmymath.so` to a standard system directory, such as `/usr/lib64`, and then run `ldconfig`. The `ldconfig` command creates the required links and cache to the most recent shared libraries found in the standard library directories.

```
$ sudo cp libmymath.so /usr/lib64/
$ sudo ldconfig
```

Compiling the application

Create an object file called `mathDemo.o` from your application source code (`mathDemo.c`):

```
$ gcc -I . -c mathDemo.c
```

The `-I` option tells GCC to search for header files (`mymath.h` in this case) in the directory listed after it. In this case, you're specifying the current directory, represented by a single dot (`.`). Create an executable, referring to your shared math library by name using the `-l` option:

```
$ gcc -o mathDynamic mathDemo.o -lmymath
```

GCC finds `libmymath.so` because it exists in a default system library directory. Use `ldd` to verify the shared libraries used:

```
$ ldd mathDemo
    linux-vdso.so.1 (0x00007fffe6a30000)
    libmymath.so => /usr/lib64/libmymath.so (0x00007fe4d4d33000)
    libc.so.6 => /lib64/libc.so.6 (0x00007fe4d4b29000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fe4d4d4e000)
```

Take a look at the size of the `mathDemo` executable:

```
$ du ./mathDynamic
24   ./mathDynamic
```

It's a small application, of course, and the amount of disk space it occupies reflects that. For comparison, a statically linked version of the same code is 932K!

```
$ ./mathDynamic
Enter two numbers
25
5

25 + 5 = 30
25 - 5 = 20
25 * 5 = 125
25 / 5 = 5
```

You can verify that it's dynamically linked with the `file` command:

```
$ file ./mathDynamic
./mathDynamic: ELF 64-bit LSB executable, x86-64,
dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
with debug_info, not stripped
```

Success!

Dynamically linking

A shared library leads to a lightweight executable, as the linking happens during runtime. Because it resolves references during runtime, it does take more time for execution. However, since the vast majority of commands on everyday Linux systems are dynamically linked and on modern hardware, the time saved is negligible. Its inherent modularity is a powerful feature for developers and users alike.

Dynamically linking libraries while compiling code

By Seth Kenlon

Compiling software is something that developers do a lot, and in open source some users even choose to do it themselves. Linux podcaster Dann Washko calls source code the "universal package format" because it contains all the components necessary to make an application run on any platform. Of course, not all source code is written for all systems, so it's only "universal" within the subset of targeted systems, but the point is that source code is extremely flexible. With open source, you can decide how code is compiled and run.

When you're compiling code, you're usually dealing with multiple source files. Developers tend to keep different classes or modules in separate files so that they can be maintained separately, and possibly even used by different projects. But when you're compiling these files, many of them get compiled into a single executable.

This is usually done by creating shared libraries, and then dynamically linking back to them from the executable. This keeps the executable small by keeping modular functions external, and ensures that libraries can be updated independently of the applications that use them.

Locating a shared object during compilation

When you're compiling with GCC, you usually need a library to be installed on your workstation for GCC to be able to locate it. By default, GCC assumes that libraries are in a system library path, such as `/lib64` and `/usr/lib64`. However, if you're linking to a library of your own that's not yet installed, or if you need to link to a library that's not installed in a standard location, then you have to help GCC find the files.

There are two options significant for finding libraries in GCC:

- `-L` (capital L) adds an additional library path to GCC's search locations.
- `-l` (lowercase L) sets the name of the library you want to link against.

For example, suppose you've written a library called `libexample.so`, and you want to use it when compiling your application `demo.c`. First, create an object file from `demo.c`:

```
$ gcc -I ./include -c src/demo.c
```

The `-I` option adds a directory to GCC's search path for header files. In this example, I assume that custom header files are in a local directory called `include`. The `-c` option prevents GCC from running a linker, because this task is only to create an object file. And that's exactly what happens:

```
$ ls
demo.o   include/   lib/    src/
```

Now you can use the `-L` option to set a path for your library, and compile:

```
$ gcc -L`pwd`/lib -o myDemo demo.o -lexample
```

Notice that the `-L` option comes *before* the `-l` option. This is significant, because if `-L` hasn't been added to GCC's search path before you tell GCC to look for a non-default library, GCC won't know to search in your custom location. The compilation succeeds as expected, but there's a problem when you attempt to run it:

```
$ ./myDemo
./myDemo: error while loading shared libraries:
libexample.so: cannot open shared object file:
No such file or directory
```

Troubleshooting with ldd

The `ldd` utility prints shared object dependencies, and it can be useful when troubleshooting issues like this:

```
$ ldd ./myDemo
        linux-vdso.so.1 (0x00007ffe151df000)
        libexample.so => not found
        libc.so.6 => /lib64/libc.so.6 (0x00007f514b60a000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f514b839000)
```

You already knew that `libexample` couldn't be located, but the `ldd` output at least affirms what's expected from a *working* library. For instance, `libc.so.6` has been located, and `ldd` displays its full path.

LD_LIBRARY_PATH

The `LD_LIBRARY_PATH` environment variable defines the path to libraries. If you're running an application that relies on a library that's not installed to a standard directory, you can add to the system's library search path using `LD_LIBRARY_PATH`.

There are several ways to set environment variables, but the most flexible is to place them before you run a command. Look at what setting `LD_LIBRARY_PATH` does for the `ldd` command when it's analyzing a "broken" executable:

```
$ LD_LIBRARY_PATH=`pwd`/lib ldd ./
   linux-vdso.so.1 (0x00007ffe515bb000)
   libexample.so => /tmp/Demo/lib/libexample.so (0x0000...
   libc.so.6 => /lib64/libc.so.6 (0x00007eff037ee000)
   /lib64/ld-linux-x86-64.so.2 (0x00007eff03a22000)
```

It applies just as well to your custom command:

```
$ LD_LIBRARY_PATH=`pwd`/lib myDemo
hello world!
```

If you move the library file or the executable, however, it breaks again:

```
$ mv lib/libexample.so ~/.local/lib64
$ LD_LIBRARY_PATH=`pwd`/lib myDemo
./myDemo: error while loading shared libraries...
```

To fix it, you must adjust the `LD_LIBRARY_PATH` to match the library's new location:

```
$ LD_LIBRARY_PATH=~/.local/lib64 myDemo
hello world!
```

When to use LD_LIBRARY_PATH

In most cases, `LD_LIBRARY_PATH` isn't a variable you need to set. By design, libraries are installed to `/usr/lib64` and so applications naturally search it for their required libraries. You may need to use `LD_LIBRARY_PATH` in two cases:

- You're compiling software that needs to link against a library that itself has just been compiled and has not yet been installed. Good build systems, such as Autotools and CMake, can help handle this.

- You're bundling software that's designed to run out of a single directory, with no install script or an install script that places libraries in non-standard directories. Several applications have releases that a Linux user can download, copy to `/opt`, and run with "no install." The `LD_PATH_LIBRARY` variable gets set through wrapper scripts so the user often isn't even aware it's been set.

Compiling software gives you a lot of flexibility in how you run your system. The `LD_LIBRARY_PATH` variable, along with the `-L` and `-l` GCC options, are components of that flexibility.

How to handle dynamic and static libraries in Linux

By Stephan Avenwedde

Linux, in a way, is a series of static and dynamic libraries that depend on each other. For new users of Linux-based systems, the whole handling of libraries can be a mystery. But with experience, the massive amount of shared code built into the operating system can be an advantage when writing new applications.

To help you get in touch with this topic, I prepared a small application example that shows the most common methods that work on common Linux distributions (these have not been tested on other systems). To follow along with this hands-on tutorial using the example application, open a command prompt and type:

```
$ git clone https://github.com/hANSIc99/library_sample
$ cd library_sample/
$ make
cc -c main.c -Wall -Werror
cc -c libmy_static_a.c -o libmy_static_a.o -Wall -Werror
cc -c libmy_static_b.c -o libmy_static_b.o -Wall -Werror
ar -rsv libmy_static.a libmy_static_a.o libmy_static_b.o
ar: creating libmy_static.a
a - libmy_static_a.o
a - libmy_static_b.o
cc -c -fPIC libmy_shared.c -o libmy_shared.o
cc -shared -o libmy_shared.so libmy_shared.o
$ make clean
rm *.o
```

After executing these commands, these files should be added to the directory (run `ls` to see them):

```
my_app
libmy_static.a
libmy_shared.so
```

About static linking

When your application links against a static library, the library's code becomes part of the resulting executable. This is performed only once at linking time, and these static libraries usually end with a `.a` extension.

A static library is an archive (ar) of object files. The object files are usually in the ELF format. ELF is short for Executable and Linkable Format, which is compatible with many operating systems.

The output of the `file` command tells you that the static library `libmy_static.a` is the `ar` archive type:

```
$ file libmy_static.a
libmy_static.a: current ar archive
```

With `ar -t`, you can look into this archive; it shows two object files:

```
$ ar -t libmy_static.a
libmy_static_a.o
libmy_static_b.o
```

You can extract the archive's files with `ar -x <archive-file>`. The extracted files are object files in ELF format:

```
$ ar -x libmy_static.a
$ file libmy_static_a.o
libmy_static_a.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not
stripped
```

About dynamic linking

Dynamic linking means the use of shared libraries. Shared libraries usually end with `.so` (short for "shared object").

Shared libraries are the most common way to manage dependencies on Linux systems. These shared resources are loaded into memory before the application starts, and when several processes require the same library, it will be loaded only once on the system. This feature saves on memory usage by the application.

Another thing to note is that when a bug is fixed in a shared library, every application that references this library will profit from it. This also means that if the bug remains undetected, each referencing application will suffer from it (if the application uses the affected parts).

It can be very hard for beginners when an application requires a specific version of the library, but the linker only knows the location of an incompatible version. In this case, you must help the linker find the path to the correct version.

Although this is not an everyday issue, understanding dynamic linking will surely help you in fixing such problems. Fortunately, the mechanics for this are quite straightforward.

To detect which libraries are required for an application to start, you can use `ldd`, which will print out the shared libraries used by a given file:

```
$ ldd my_app
        linux-vdso.so.1 (0x00007ffd1299c000)
        libmy_shared.so => not found
        libc.so.6 => /lib64/libc.so.6 (0x00007f56b869b000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f56b8881000)
```

Note that the library `libmy_shared.so` is part of the repository but is not found. This is because the dynamic linker, which is responsible for loading all dependencies into memory before executing the application, cannot find this library in the standard locations it searches.

Errors associated with linkers finding incompatible versions of common libraries (like `bzip2`, for example) can be quite confusing for a new user. One way around this is to add the repository folder to the environment variable `LD_LIBRARY_PATH` to tell the linker where to look for the correct version. In this case, the right version is in this folder, so you can export it:

```
$ LD_LIBRARY_PATH=$(pwd):$LD_LIBRARY_PATH
$ export LD_LIBRARY_PATH
```

Now the dynamic linker knows where to find the library, and the application can be executed. You can rerun `ldd` to invoke the dynamic linker, which inspects the application's dependencies and loads them into memory. The memory address is shown after the object path:

```
$ ldd my_app
        linux-vdso.so.1 (0x00007ffd385f7000)
        libmy_shared.so => /home/stephan/library_sample/libmy_shared.so
(0x00007f3fad401000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f3fad21d000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f3fad408000)
```

To find out which linker is invoked, you can use `file`:

```
$ file my_app
my_app: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=26c677b771122b4c99f0fd9ee001e6c743550fa6, for GNU/Linux 3.2.0, not
stripped
```

The linker `/lib64/ld-linux-x86-64.so.2` is a symbolic link to `ld-2.30.so`, which is the default linker for my Linux distribution:

```
$ file /lib64/ld-linux-x86-64.so.2
/lib64/ld-linux-x86-64.so.2: symbolic link to ld-2.31.so
```

Looking back to the output of `ldd`, you can also see (next to `libmy_shared.so`) that each dependency ends with a number (e.g., `/lib64/libc.so.6`). The usual naming scheme of shared objects is:

```
**lib** XYZ.so **.<MAJOR>** . **<MINOR>**
```

On my system, `libc.so.6` is also a symbolic link to the shared object `libc-2.30.so` in the same folder:

```
$ file /lib64/libc.so.6
/lib64/libc.so.6: symbolic link to libc-2.31.so
```

If you are facing the issue that an application will not start because the loaded library has the wrong version, it is very likely that you can fix this issue by inspecting and rearranging the symbolic links or specifying the correct search path (see "The dynamic loader: ld.so" below).

For more information, look on the `ldd` man page.

Dynamic loading

Dynamic loading means that a library (e.g., a `.so` file) is loaded during a program's runtime. This is done using a certain programming scheme. Dynamic loading is applied when an application uses plugins that can be modified during runtime.

The dynamic loader: ld.so

On Linux, you mostly are dealing with shared objects, so there must be a mechanism that detects an application's dependencies and loads them into memory.

`ld.so` looks for shared objects in these places in the following order:

1. The relative or absolute path in the application (hardcoded with the `-rpath` compiler option on GCC)
2. In the environment variable `LD_LIBRARY_PATH`
3. In the file `/etc/ld.so.cache`

Keep in mind that adding a library to the systems library archive `/usr/lib64` requires administrator privileges. You could copy `libmy_shared.so` manually to the library archive and make the application work without setting `LD_LIBRARY_PATH`:

```
unset LD_LIBRARY_PATH
sudo cp libmy_shared.so /usr/lib64/
```

When you run `ldd`, you can see the path to the library archive shows up now:

```
$ ldd my_app
        linux-vdso.so.1 (0x00007ffe82fab000)
        libmy_shared.so => /lib64/libmy_shared.so (0x00007f0a963e0000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f0a96216000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f0a96401000)
```

Customize the shared library at compile time

If you want your application to use your shared libraries, you can specify an absolute or relative path during compile time.

Modify the makefile (line 10) and recompile the program by invoking `make -B`. Then, the output of `ldd` shows `libmy_shared.so` is listed with its absolute path.

Change this:

```
CFLAGS =-Wall -Werror -Wl,-rpath,$(shell pwd)
```

To this (be sure to edit the username):

```
CFLAGS =/home/stephan/library_sample/libmy_shared.so
```

Then recompile:

```
$ make
```

Confirm it is using the absolute path you set, which you can see on line 2 of the output:

```
$ ldd my_app
    linux-vdso.so.1 (0x00007ffe143ed000)
        libmy_shared.so => /lib64/libmy_shared.so (0x00007fe50926d000)
        /home/stephan/library_sample/libmy_shared.so (0x00007fe509268000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fe50909e000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe50928e000)
```

This is a good example, but how would this work if you were making a library for others to use? New library locations can be registered by writing them to `/etc/ld.so.conf` or creating a `<library-name>.conf` file containing the location under `/etc/ld.so.conf.d/`. Afterward, `ldconfig` must be executed to rewrite the `ld.so.cache` file. This step is sometimes necessary after you install a program that brings some special shared libraries with it.

How to handle multiple architectures

Usually, there are different libraries for the 32-bit and 64-bit versions of applications. The following list shows their standard locations for different Linux distributions:

Red Hat family

- 32 bit: `/usr/lib`
- 64 bit: `/usr/lib64`

Debian family

- 32 bit: `/usr/lib/i386-linux-gnu`
- 64 bit: `/usr/lib/x86_64-linux-gnu`

Arch Linux family

- 32 bit: `/usr/lib32`
- 64 bit: `/usr/lib64`

FreeBSD (technically not a Linux distribution)

- 32bit: `/usr/lib32`
- 64bit: `/usr/lib`

Knowing where to look for these key libraries can make broken library links a problem of the past.

While it may be confusing at first, understanding dependency management in Linux libraries is a way to feel in control of the operating system. Run through these steps with other applications to become familiar with common libraries, and continue to learn how to fix any library challenges that could come up along your way.

How static linking works on Linux

By Jayashree Huttanagoudar

Code for applications written using C usually has multiple source files, but ultimately you will need to compile them into a single executable.

You can do this in two ways: by creating a *static* library or a *dynamic* library (also called a *shared* library). These two types of libraries vary in terms of how they are created and linked. Your choice of which to use depends on your use case.

In this article, I explain how to create a statically linked executable.

Using a linker with static libraries

A linker is a command that combines several pieces of a program together and reorganizes the memory allocation for them.

The functions of a linker include:

- Integrating all the pieces of a program
- Figuring out a new memory organization so that all the pieces fit together
- Reviving addresses so that the program can run under the new memory organization
- Resolving symbolic references

As a result of all these linker functionalities, a runnable program called an *executable* is created.

Static libraries are created by copying all necessary library modules used in a program into the final executable image. The linker links static libraries as a last step in the compilation process. An executable is created by resolving external references, combining the library routines with program code.

Create the object files

Here's an example of a static library, along with the linking process. First, create the header file `mymath.h` with these function signatures:

```
int add(int a, int b);
int sub(int a, int b);
int mult(int a, int b);
int divi(int a, int b);
```

Create `add.c`, `sub.c` , `mult.c` and `divi.c` with these function definitions:

```
// add.c
int add(int a, int b){
return (a+b);
}

//sub.c
int sub(int a, int b){
return (a-b);
}

//mult.c
int mult(int a, int b){
return (a*b);
}

//divi.c
int divi(int a, int b){
return (a/b);
}
```

Now generate object files `add.o`, `sub.o`, `mult.o`, and `divi.o` using GCC:

```
$ gcc -c add.c sub.c mult.c divi.c
```

The `-c` option skips the linking step and creates only object files.

Create a static library called `libmymath.a`, then remove the object files, as they're no longer required. (Note that using a `trash` command is safer than `rm`.)

```
$ ar rs libmymath.a add.o sub.o mult.o divi.o
$ trash *.o
$ ls
add.c  divi.c  libmymath.a  mult.c  mymath.h  sub.c
```

You have now created a simple example math library called `libmymath`, which you can use in C code. There are, of course, very complex C libraries out there, and this is the process their developers use to generate the final product that you and I install for use in C code.

Next, use your math library in some custom code and then link it.

Create a statically linked application

Suppose you've written a command for mathematics. Create a file called `mathDemo.c` and paste this code into it:

```
#include <mymath.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int x, y;
  printf("Enter two numbers\n");
  scanf("%d%d",&x,&y);

  printf("\n%d + %d = %d", x, y, add(x, y));
  printf("\n%d - %d = %d", x, y, sub(x, y));
  printf("\n%d * %d = %d", x, y, mult(x, y));

  if(y==0){
    printf("\nDenominator is zero so can't perform division\n");
      exit(0);
  }else{
      printf("\n%d / %d = %d\n", x, y, divi(x, y));
      return 0;
  }
}
```

Notice that the first line is an `include` statement referencing, by name, your own `libmymath` library.

Create an object file called `mathDemo.o` for `mathDemo.c`:

```
$ gcc -I . -c mathDemo.c
```

The `-I` option tells GCC to search for header files listed after it. In this case, you're specifying the current directory, represented by a single dot (`.`).

Link `mathDemo.o` with `libmymath.a` to create the final executable. There are two ways to express this to GCC.

You can point to the files:

```
$ gcc -static -o mathDemo mathDemo.o libmymath.a
```

Alternately, you can specify the library path along with the library name:

```
$ gcc -static -o mathDemo -L . mathDemo.o -lmymath
```

In the latter example, the `-lmymath` option tells the linker to link the object files present in the `libmymath.a` with the object file `mathDemo.o` to create the final executable. The `-L` option directs the linker to look for libraries in the following argument (similar to what you would do with `-I`).

Analyzing the result

Confirm that it's statically linked using the `file` command:

```
$ file mathDemo
mathDemo: ELF 64-bit LSB executable, x86-64...
statically linked, with debug_info, not stripped
```

Using the `ldd` command, you can see that the executable is not dynamically linked:

```
$ ldd ./mathDemo
        not a dynamic executable
```

You can also check the size of the `mathDemo` executable:

```
$ du -h ./mathDemo
932K    ./mathDemo
```

By comparison, the dynamic executable of the same code took up just 24K.

Run the command to see it work:

```
$ ./mathDemo
Enter two numbers
10
5

10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
10 / 5 = 2
```

Looks good!

When to use static linking

Dynamically linked executables are generally preferred over statically linked executables because dynamic linking keeps an application's components modular. Should a library receive a critical security update, it can be easily patched because it exists outside of the applications that use it.

When you use static linking, a library's code gets "hidden" within the executable you create, meaning the only way to patch it is to re-compile and re-release a new executable every time a library gets an update—and you have better things to do with your time, trust me.

However, static linking is a reasonable option if the code of a library exists either in the same code base as the executable using it or in specialized embedded devices that are expected to receive no updates.

32-bit life support: Cross-compiling with GCC

By Seth Kenlon

If you're a developer creating binary packages, like an RPM, DEB, Flatpak, or Snap, you have to compile code for a variety of different target platforms. Typical targets include 32-bit and 64-bit x86 and ARM. You could do your builds on different physical or virtual machines, but that means maintaining several systems. Instead, you can use the GNU Compiler Collection (GCC) to cross-compile, producing binaries for several different architectures from a single build machine.

Assume you have a simple dice-rolling game that you want to cross-compile. Something written in C is relatively easy on most systems, so to add complexity for the sake of realism, I wrote this example in C++, so the program depends on something not present in C (`iostream`, specifically).

```
#include <iostream>
#include <cstdlib>

using namespace std;

void lose (int c);
void win (int c);
void draw ();

int main() {
  int i;
    do {
      cout << "Pick a number between 1 and 20: \n";
      cin >> i;
      int c = rand ( ) % 21;
      if (i > 20) lose (c);
      else if (i < c ) lose (c);
      else if (i > c ) win (c);
```

```
      else draw ();
      }
      while (1==1);
      }

void lose (int c )
  {
    cout << "You lose! Computer rolled " << c << "\n";
  }

void win (int c )
  {
    cout << "You win!! Computer rolled " << c << "\n";
   }

void draw ( )
   {
     cout << "What are the chances. You tied. Try again, I dare you! \n";
   }
```

Compile it on your system using the g++ command:

```
$ g++ dice.cpp -o dice
```

Then run it to confirm that it works:

```
$ ./dice
Pick a number between 1 and 20:
[...]
```

You can see what kind of binary you just produced with the `file` command:

```
$ file ./dice
dice: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 5.1.15, not stripped
```

And just as important, what libraries it links to with `ldd`:

```
$ ldd dice
linux-vdso.so.1 => (0x00007ffe0d1dc000)
libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6
(0x00007fce8410e000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007fce83d4f000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6
(0x00007fce83a52000)
```

```
/lib64/ld-linux-x86-64.so.2 (0x00007fce84449000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1
(0x00007fce8383c000)
```

You have confirmed two things from these tests: The binary you just ran is 64-bit, and it is linked to 64-bit libraries.

That means that, in order to cross-compile for 32-bit, you must tell g++ to:

1. Produce a 32-bit binary
2. Link to 32-bit libraries instead of the default 64-bit libraries

Setting up your dev environment

To compile to 32-bit, you need 32-bit libraries and headers installed on your system. If you run a pure 64-bit system, then you have no 32-bit libraries or headers and need to install a base set. At the very least, you need the C and C++ libraries (glibc and libstdc++) along with 32-bit version of GCC libraries (libgcc). The names of these packages may vary from distribution to distribution. On Slackware, a pure 64-bit distribution with 32-bit compatibility is available from the multilib packages provided by Alien BOB. On Fedora, CentOS, and RHEL:

```
$ yum install libstdc++-*.i686
$ yum install glibc-*.i686
$ yum install libgcc.i686
```

Regardless of the system you're using, you also must install any 32-bit libraries your project uses. For instance, if you include yaml-cpp in your project, then you must install the 32-bit version of yaml-cpp or, on many systems, the development package for yaml-cpp (for instance, yaml-cpp-devel on Fedora) before compiling it.

Once that's taken care of, the compilation is fairly simple:

```
$ g++ -m32 dice.cpp -o dice32 -L /usr/lib -march=i686
```

The `-m32` flag tells GCC to compile in 32-bit mode. The `-march=i686` option further defines what kind of optimizations to use (refer to `info gcc` for a list of options). The `-L` flag sets the path to the libraries you want GCC to link to. This is usually `/usr/lib` for 32-bit, although, depending on how your system is set up, it could be `/usr/lib32` or even /opt/usr/lib or any place you know you keep your 32-bit libraries.

After the code compiles, see proof of your build:

```
$ file ./dice32
dice: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs) [...]
```

And, of course, `ldd ./dice32` points to your 32-bit libraries.

Different architectures

Compiling 32-bit on 64-bit for the same processor family allows GCC to make many assumptions about how to compile the code. If you need to compile for an entirely different processor, you must install the appropriate cross-build GCC utilities. Which utility you install depends on what you are compiling. This process is a little more complex than compiling for the same CPU family.

When you're cross-compiling for the same family, you can expect to find the same set of 32-bit libraries as 64-bit libraries, because your Linux distribution is maintaining both. When compiling for an entirely different architecture, you may have to hunt down libraries required by your code. The versions you need may not be in your distribution's repositories because your distribution may not provide packages for your target system, or it may not mirror all packages in a convenient location. If the code you're compiling is yours, then you probably have a good idea of what its dependencies are and possibly where to find them. If the code is something you have downloaded and need to compile, then you probably aren't as familiar with its requirements. In that case, investigate what the code requires to build correctly (they're usually listed in the README or INSTALL files, and certainly in the source code itself), then go gather the components.

For example, if you need to compile C code for ARM, you must first install gcc-arm-linux-gnu (32-bit) or gcc-aarch64-linux-gnu (64-bit) on Fedora or RHEL, or arm-linux-gnueabi-gcc and binutils-arm-linux-gnueabi on Ubuntu. This provides the commands and libraries you need to build (at least) a simple C program. Additionally, you need whatever libraries your code uses. You can place header files in the usual location (`/usr/include` on most systems), or you can place them in a directory of your choice and point GCC to it with the `-I` option.

When compiling, don't use the standard gcc or g++ command. Instead, use the GCC utility you installed. For example:

```
$ arm-linux-gnu-g++ dice.cpp \
  -I/home/seth/src/crossbuild/arm/cpp \
  -o armdice.bin
```

Verify what you've built:

```
$ file armdice.bin
armdice.bin: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV) [...]
```

Libraries and deliverables

This was a simple example of how to use cross-compiling. In real life, your source code may produce more than just a single binary. While you can manage this manually, there's probably no good reason to do that.