Create Static and Dynamic Link Libraries in C using gcc on Linux

Contents

Static and Dynamic Linking of Libraries

Static and dynamic linking are two processes of collecting and combining multiple object files in order to create a single executable. Linking can be performed at both compile time, when the source code is translated into machine code and load time, when the program is loaded into memory and executed by the loader, and even at run time, by application programs. And, it is performed by programs called *linkers*. Linkers are also called link editors. Linking is performed as the last step in compiling a program. In this tutorial static and dynamic linking with C modules will be discussed.

What is Linker?

Linker is system software which plays crucial role in software development because it enables separate compilation. Instead of organizing a large application as one monolithic source file, you can decompose it into smaller, more manageable chunks that can be modified and compiled separately. When you change one of the modules, you simply recompile it and re-link the application, without recompiling the other source files.

During static linking the linker copies all library routines used in the program into the executable image. This of course takes more space on the disk and in memory than dynamic linking. But static linking is faster and more portable because it does not require the presence of the library on the system where it runs.

At the other hand, in dynamic linking shareable library name is placed in the executable image, while actual linking takes place at run time when both the executable and the library are placed in memory. Dynamic linking serves the advantage of sharing a single shareable library among multiple programs.

Linker as a system program takes relocatable object files and command line arguments in order to generate an executable object file. To produce an executable file the Linker has to perform the symbol resolution, and Relocation.

Note: Object files come in three flavors viz *Relocatable*, *Executable*, and *Shared*.
Relocatable object files contain code and data in a form which can be combined with other object files of its kind at compile time to create an executable object file. They consist of various code and data sections. Instructions are in one section, initialized global variables in another section, and uninitialized variables are yet in another section.
Executable object files contain binary code and data in a form which can directly be copied into memory and executed.
Shared object files are files those can be loaded into memory and linked dynamically, at either load or run time by a linker.
Through this article, *static* and *dynamic* linking will be explained. While linking, the linker complains about missing function definitions, if there is any. During compilation, if compiler does not find a function definition for a particular module, it just assumes that the function is defined in another file, and treats it as an external reference. The compiler does not look at more than one file at a time. Whereas, linker may look at multiple files and seeks references for the modules that were not mentioned. The separate compilation and linking processes reduce the complexity of program and gives the ease to break code into smaller pieces which are better manageable.

What is Static Linking?

Static linking is the process of copying all library modules used in the program into the final executable image. This is performed by the linker and it is done as the last step of the compilation process. The linker combines library routines with the program code in order to resolve external references, and to generate an executable image suitable for loading into memory.

Let's see static linking by example. Here, we will take a very simple example of adding two integer quantities to demonstrate the static linking process. We will develop an add module and place in a separate add.c file. Prototype of add module will be placed in a separate file called add.h. Code file addDemo.c will be created to demonstrate the linking process.
To begin with, create a header file add.h and insert the add function signature into that as follows:

```
int add(int, int);
```

Now, create another source code file viz addDemo.c, and insert the following code into that.

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

int main()
{
 int x= 10, y = 20;
 printf("\n%d + %d = %d", x, y, add(x, y));
 return 0;
}
```

Create one more file named add.c that contains the code of add module. Insert the following code into add.c

```
int add(int quant1, int quant2)
```

```
{
  return(quant1 + quant2);
}
```

After having created above files, you can start building the executable as follows:

```
[root@host ~]# gcc -I . -c addDemo.c
```

The `-I` option tells GCC to search for header files in the directory which is specified after it. Here, GCC is asked to look for header files in the current directory along with the include directory. (in Unix like systems, dot(.) is interpreted as current directory).
Note: Some applications use header files installed in /usr/local/include and while compiling them, they usually tell GCC to look for these header files there.

The `-c` option tells GCC to compile to an object file. The object file will have name as *.o. Where * is the name of file without extension. It will stop after that and won't perform the linking to create the executable.
As similar to the above command, compile add.c to create the object file. It is done by following command.

```
[root@host ~]# gcc -c add.c
```

This time the `-I` option has been omitted because `add.c` requires no header files to include. The above command will produce a new file viz `add.o`. Now the final step is to generate the executable by linking `add.o`, and `addDemo.o` together. Execute the following command to generate executable object file.

```
[root@host ~]# gcc -o addDemo add.o addDemo.o
```

Although, the linker could directly be invoked for this purpose by using ld command, but we preferred to do it through the compiler because there are other object files or paths or options, which must be linked in order to get the final executable.

Now that we know how to create an executable object file from more than one binary object files. It's time to know about libraries. In the following section we will see what libraries are in software development process? How are they created? What is their significance in software development? How are they used, and many things which have made the use of libraries far obvious in software development?

How to Create Static Libraries?

Static and shared libraries are simply collections of binary object files they help during linking. A library contains hundreds or thousands of object files. During the demonstration of `addDemo` we had two object files viz `add.o`, and `addDemo.o`. There might be chances that you would have ten or more object files which have to be linked together in order to get the final executable object file. In those situations you will find yourself enervate, and every time you will have to specify a lengthy list of object files in order to get the final executable object file. Moreover, fenceless object files will be difficult to manage. Libraries solve all these difficulties, and help you to keep the organization of object files simple and maintainable.

Static libraries are explained here, dynamic libraries will be explained along with dynamic linking. Static libraries are bundle of relocatable object files. Usually they have `.a` extension. To demonstrate the use of static libraries we will extend the`addDemo` example further. So far we have `add.o` which contains the binary object code of `add` function which we used in `addDemo`. For more explanatory demonstration of use of libraries we would create a new header file `heymath.h` and will add signatures of two functions `add`, `sub` to that.

```
int add(int, int); //adds two integers
int sub(int, int); //subtracts second integer from first
```

Next, create a file `sub.c`, and add the following code to it. We have `add.c` already created.

```
int sub(int quant1, int quant2)
{
  return (quant1 - quant2);
}
```

Now compile sub.c as follows in order to get the binary object file.

```
[root@host ~]# gcc -c sub.c
```

Above command will produce binary object file sub.o.

Now, we have two binary object files viz add.o, and sub.o. We have add.o file in working directory as we have created it for previous example. If you have not done this so far then create the add.o from add.c in similar fashion as sub.o has been created. We will now create a static library by collecting both files together. It will make our final executable object file creation job easier and next time we will have not to specify two object files along with addDemo in order to generate the final executable object file. Create the static library libheymath by executing the following command:

```
[root@host ~]# ar rs libheymath.a add.o sub.o
```

The above command produces a new file libheymath.a, which is a static library containing two object files and can be used further as and when we wish to useadd, or sub functions or both in our programs.

To use the sub function in addDemo we need to make a few changes in addDemo.cand will recompile it. Make the following changes in addDemo.c.

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

int main()
{
  int x = 10, y = 20;
  printf("\n%d + %d = %d", x, y, add(x, y));
  printf("\n%d + %d = %d", x, y, sub(x, y));
  return 0;
}
```

If you see, we have replaced the first statement `#include <add.h>` by `#include <heymath.h>`. Because `heymath.h` now contains the signatures of both `add` and `sub`functions and added one more `printf` statement which is calling the `sub` function to print the difference of variable `x`, and `y`.

Now remove all `.o` files from working directory (rm will help you to do that). Create `addDemo.o` as follows:

```
[root@host ~]# gcc -I . -c addDemo.c
```

And link it with heymath.a to generate final executable object file.

```
[root@host ~]# gcc -o addDemo addDemo.o libheymath.a
```

You can also use the following command as an alternate to link the `libheymath.a`with `addDemo.o` in order to generate the final executable file.

```
[root@host ~]# gcc -o addDemo -L . addDemo.o -lheymath
```

In above command `-lheymath` should be read as `-l heymath` which tells the linker to link the object files contained in lib<library>.a with `addDemo` to generate the executable object file. In our example this is `libheymath.a`.

The -L option tells the linker to search for libraries in the following argument (similar to how we did for `-I`). So, what we created as of now is a static library. But this is not the end; systems use a lot of dynamic libraries as well. It is the right time to discuss them.

What is Dynamic Linking?

Dynamic linking defers much of the linking process until a program starts running. It performs the linking process "on the fly" as programs are executed in the system. During dynamic linking the name of the shared library is placed in the final executable file while the actual linking takes place at run time when both executable file and library are placed in the memory. The main advantage to using dynamically linked libraries is that the size of executable programs is dramatically reduced because each program does not have to store redundant copies of the library functions that it uses. Also, when DLL functions are updated, programs that use them will automatically obtain their benefits.

How to Create and Use Shared Libraries?

A shared library (on Linux) or a dynamic link library (dll on Windows) is a collection of object files. In dynamic linking, object files are not combined with programs at compile time, also, they are not copied permanently into the final executable file; therefore, a shared library reduces the size of final executable.

Shared libraries or dynamic link libraries (dlls) serve a great advantage of sharing a single copy of library among multiple programs, hence they are called shared libraries, and the process of linking them with multiple programs is called dynamic linking.

Shared libraries are loaded into memory by programs when they start. When a shared library is loaded properly, all programs that start later automatically use the already loaded shared library. Following text will demonstrate how to create and use shared library on Linux.

Let's continue with the previous example of `add`, and `sub` modules. As you remember we had two object files `add.o`, and `sub.o` (compiled from `add.c` and`sub.c`) that contain code of `add` and `sub` methods respectively. But we will have to recompile both `add.c` and `sub.c` again with `-fpic` or `-fPIC` option. The `-fPIC` or `-fpic` option enable *"position independent code"* generation, a requirement for shared libraries. Use -fPIC or -fpic to generate code. Which option should be used, -fPIC or -fpic to generate code that is target-dependent. The -fPIC choice always works, but may produce larger code than -fpic. Using -fpic option usually generates smaller and faster code, but will have platform-dependent limitations, such as the number of globally visible symbols or the size of the code. The linker will tell you whether it fits when you create the shared library. When in doubt, I choose -fPIC, because it always works. So, while creating shared library you have to recompile both add.c, and sub.c with following options:

```
[root@host ~]# gcc -Wall -fPIC -c add.c
[root@host ~]# gcc -Wall -fPIC -c sub.c
```

Above commands will produce two fresh object files in current directory add.o, and sub.o. The warning option -Wall enables warnings for many common errors, and should always be used. It combines a large number of other, more specific, warning options which can also be selected individually. For details you can see man page for warnings specified.

Now build the library libheymath.so using the following command.

```
[root@host ~]# gcc -shared -o libheymath.so add.o sub.o
```

This newly created shared library libheymath.so can be used as a substitute oflibheymath.a. But to use a shared library is not as straightforward as static library was. Once you create a shared library you will have to install it. And, the simplest approach of installation is to copy the library into one of the standard directories (e.g., /usr/lib) and run ldconfig command.

Thereafter executing ldconfig command, the addDemo executable can be built as follows. I recompile addDemo.c also. You can omit it if addDemo.o is already there in your working directory.

```
[root@host ~]# gcc -c addDemo.c
[root@host ~]# gcc -o addDemo addDemo.o libheymath.so
```

or

```
[root@host ~]# gcc -o addDemo addDemo.o -lheymath
```

You can list the shared library dependencies which your executable is dependent upon. The `ldd <name-of-executable>` command does that for you.

```
[root@host ~]# ldd addition

libheymath.so => /usr/lib/libheymath.so (0x00002b19378fa000)

libc.so.6 => /lib64/libc.so.6 (0x00002b1937afb000)

/lib64/ld-linux-x86-64.so.2 (0x00002b19376dd000)
```

Exercise

Create a static and dynamic library which supports the following functions for a calculator application:
Add
Subtract
Multiply
Divide

Write a calculator application and demonstrate the calculator using both static and dynamic libraries.