How-To Geek

How to Use Linux's ar Command to Create Static Libraries

DAVE MCKAY @thegurkha
JUL 8, 2019, 8:00 AM EDT | 9 MIN READ

Fatmawati Achmad Zaenuri/Shutterstock.com

Use Linux's `ar` command to create function libraries when you're developing software. This tutorial will show you how to create a static library, modify it, and use it in a program, complete with sample code.

The `ar` command is a real veteran—it has been around since 1971. The name `ar` references the original intended use for the tool, which was [to create archive files](). An archive file is a single file that acts as a container for other files. Sometimes for many other files. Files can be added to, removed from, or extracted from the archive. People looking for that type of functionality no longer turn to `ar`. That role has been taken over by other utilities such as `tar`.

The `ar` command is still used for a few specialist purposes, though. ar is used to create static libraries. These are used in software development. And ar is also be used to create package files such as the ".deb" files used in the Debian Linux distribution and its derivatives such as Ubuntu.

We're going to run through the steps required to create and modify a static library, and demonstrate how to use the library in a program. To do that we need a requirement for the static library to fulfill. The purpose of this library is to encode strings of text and to decode encoded text.

Please note, this is a quick and dirty hack for demonstration purposes. Don't use this encryption for anything that is of value. It is the world's simplest substitution cipher, where A becomes B, B becomes C, and so on.

RELATED: ***How to Compress and Extract Files Using the tar Command on Linux***

The cipher_encode() and cipher_decode() Functions

We're going to be working in a directory called "library," and later we'll create a subdirectory called "test."

We have two files in this directory. In a text file called cipher_encode.c we have the `cipher_encode()` function:

```
void cipher_encode(char *text)
{
 for (int i=0; text[i] != 0x0; i++) {
   text[i]++;
 }

} // end of cipher_encode
```

The corresponding `cipher_decode()` function is in a text file called cipher_decode.c:

```
void cipher_decode(char *text)
{
 for (int i=0; text[i] != 0x0; i++) {
   text[i]--;
 }

} // end of cipher_decode
```

Files which contain programming instructions are called source code files. We're going to make a library file called libcipher.a. It will contain the compiled versions of these two source code files. We'll also create a short text file called libcipher.h. This is a header file containing the definitions of the two functions in our new library.

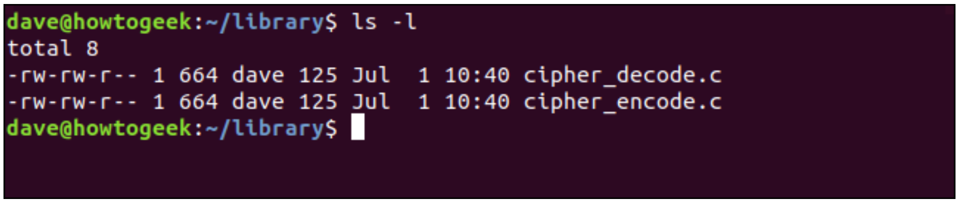
Anyone with the library and the header file will be able to use the two functions in their own programs. They do not need to re-invent the wheel and re-write the functions; they simply make use of the copies in our library.

Compiling the cipher_encode.c and cipher_decode.c Files

To compile the source code files, we will use `gcc`, the standard GNU compiler. The `-c` (compile, no link) option tells `gcc` to compile the files and then stop. It produces an intermediary file from each source code file called an object file. The `gcc` linker usually takes all the object files and links them together to make an executable program. We're skipping that step by using the `-c` option. We just need the object files.

Let's check we have the files we think we do.

```
ls -l
```



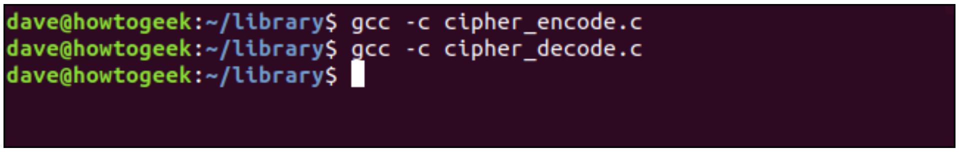
```
dave@howtogeek:~/library$ ls -l
total 8
-rw-rw-r-- 1 664 dave 125 Jul  1 10:40 cipher_decode.c
-rw-rw-r-- 1 664 dave 125 Jul  1 10:40 cipher_encode.c
dave@howtogeek:~/library$ 
```

The two source code files are present in this directory. Let's use `gcc` to compile them to object files.

```
gcc -c cipher_encode.c
```

```
gcc -c cipher_decode.c
```

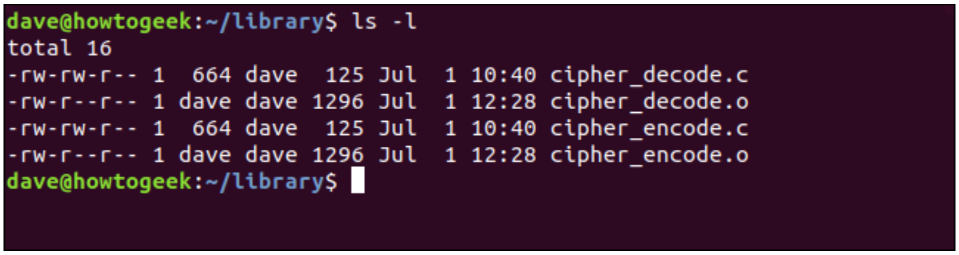
There should be no output from `gcc` if all goes well.



```
dave@howtogeek:~/library$ gcc -c cipher_encode.c
dave@howtogeek:~/library$ gcc -c cipher_decode.c
dave@howtogeek:~/library$ 
```

This generates two object files with the same name as the source code files, but with ".o" extensions. These are the files we need to add to the library file.

```
ls -l
```



```
dave@howtogeek:~/library$ ls -l
total 16
-rw-rw-r-- 1  664 dave  125 Jul  1 10:40 cipher_decode.c
-rw-r--r-- 1 dave dave 1296 Jul  1 12:28 cipher_decode.o
-rw-rw-r-- 1  664 dave  125 Jul  1 10:40 cipher_encode.c
-rw-r--r-- 1 dave dave 1296 Jul  1 12:28 cipher_encode.o
dave@howtogeek:~/library$
```

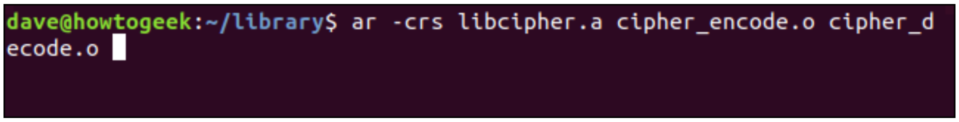
Creating the libcipher.a Library

To create the library file—which is actually an archive file—we will use `ar`.

We are using the `-c` (create) option to create the library file, the `-r` (add with replace) option to add the files to the library file, and the `-s` (index) option to create an index of the files inside the library file.

We are going to call the library file libcipher.a. We provide that name on the command line, together with the names of the object files we are going to add to the library.

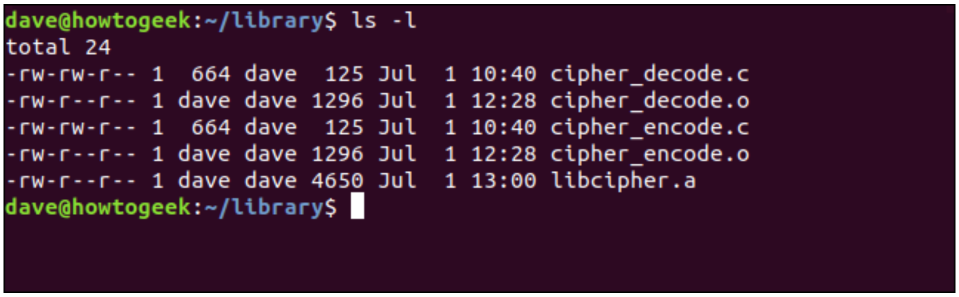
```
ar -crs libcipher.a cipher_encode.o cipher_decode.o
```



```
dave@howtogeek:~/library$ ar -crs libcipher.a cipher_encode.o cipher_d
ecode.o
```

If we list the files in the directory, we will see we now have a libcipher.a file.

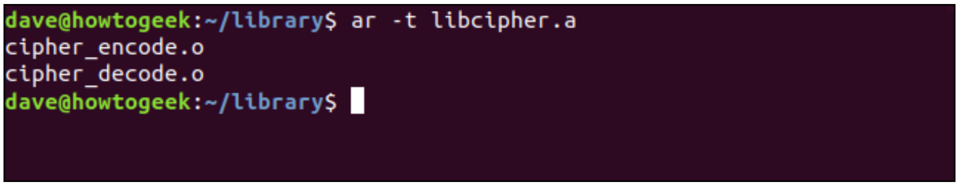
ls -l



```
dave@howtogeek:~/library$ ls -l
total 24
-rw-rw-r-- 1  664 dave  125 Jul  1 10:40 cipher_decode.c
-rw-r--r-- 1 dave dave 1296 Jul  1 12:28 cipher_decode.o
-rw-rw-r-- 1  664 dave  125 Jul  1 10:40 cipher_encode.c
-rw-r--r-- 1 dave dave 1296 Jul  1 12:28 cipher_encode.o
-rw-r--r-- 1 dave dave 4650 Jul  1 13:00 libcipher.a
dave@howtogeek:~/library$ 
```

If we use the `-t` (table) option with *ar* we can see the modules inside the library file.

```
ar -t libcipher.a
```



```
dave@howtogeek:~/library$ ar -t libcipher.a
cipher_encode.o
cipher_decode.o
dave@howtogeek:~/library$ 
```

Creating the libcipher.h header File

The libcipher.h file will be included in any program that uses the libcipher.a library. The libcipher.h file must contain the definition of the functions that are in the library.

To create the header file, we must type the function definitions into a text editor such as gedit. Name the file "libcipher.h" and save it in the same directory as the libcipher.a file.

```
void cipher_encode(char *text);
void cipher_decode(char *text);
```

Using the libcipher Library

The only sure way to test our new library is to write a little program to use it. First, we'll make a directory called test.

```
mkdir test
```

We'll copy the library and header files into the new directory.

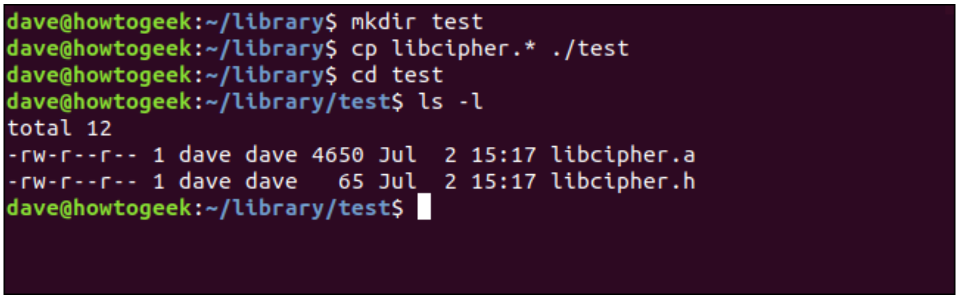
```
cp libcipher.* ./test
```

We'll change into the new directory.

```
cd test
```

Let's check that our two files are here.

```
ls -l
```



```
dave@howtogeek:~/library$ mkdir test
dave@howtogeek:~/library$ cp libcipher.* ./test
dave@howtogeek:~/library$ cd test
dave@howtogeek:~/library/test$ ls -l
total 12
-rw-r--r-- 1 dave dave 4650 Jul  2 15:17 libcipher.a
-rw-r--r-- 1 dave dave   65 Jul  2 15:17 libcipher.h
dave@howtogeek:~/library/test$
```

We need to create a small program that can use the library and prove that it functions as expected. Type the following lines of text into an editor. Save the contents of the editor to a file named "test.c" in the *test* directory.

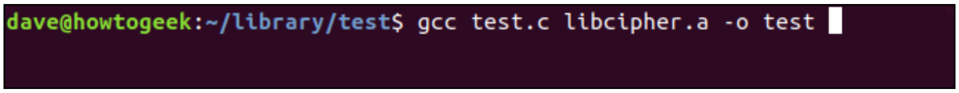
```
#include <stdio.h>
#include <stdlib.h>

#include "libcipher.h"

int main(int argc, char *argv[])
{
 char text[]="How-To Geek loves Linux";

 puts(text);
 cipher_encode(text);
 puts(text);

 cipher_decode(text);
 puts(text);

 exit (0);

} // end of main
```

The program flow is very simple:

- It includes the libcipher.h file so that it can see the library function definitions.
- It creates a string called "text" and stores the words "How-To Geek loves Linux" in it.
- It prints that string to the screen.
- it calls the `cipher_encode()` function to encode the string, and it prints the encoded string to the screen.
- It calls `cipher_decode()` to decode the string and prints the decoded string to the screen.

To generate the `test` program, we need to compile the test.c program and link in the library. The `-o` (output) option tells `gcc` what to call the executable program that it generates.

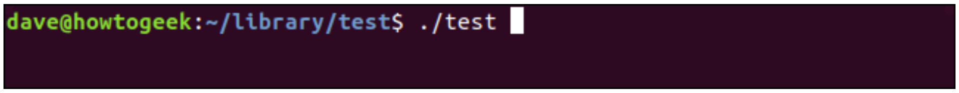
```
gcc test.c libcipher.a -o test
```



```
dave@howtogeek:~/library/test$ gcc test.c libcipher.a -o test
```

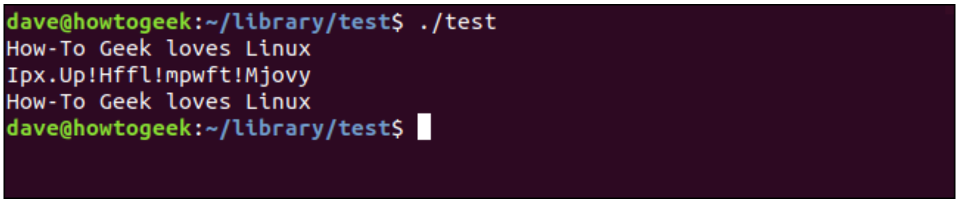
If `gcc` silently returns you to the command prompt, all is well. Now let's test our program. Moment of truth:

```
./test
```



```
dave@howtogeek:~/library/test$ ./test
```

And we see the expected output. The `test` program prints the plain text prints the encrypted text and then prints the decrypted text. It is using the functions within our new library. Our library is working.



```
dave@howtogeek:~/library/test$ ./test
How-To Geek loves Linux
Ipx.Up!Hffl!mpwft!Mjovy
How-To Geek loves Linux
dave@howtogeek:~/library/test$
```

Success. But why stop there?

Adding Another Module to the Library

Let's add another function to the library. We'll add a function that the programmer can use to display the version of the library that they are using. We'll need to create the new function, compile it, and add the new object file to the existing library file.

Type the following lines into an editor. Save the contents of the editor to a file named cipher_version.c, in the *library* directory.

```
#include <stdio.h>

void cipher_version(void)
{
 puts("How-To Geek :: VERY INSECURE Cipher Library");
 puts("Version 0.0.1 Alpha\n");

} // end of cipher_version
```

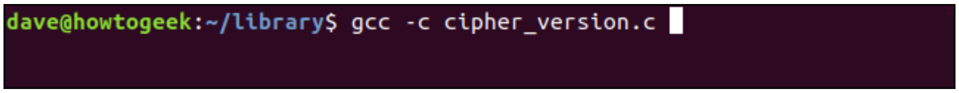
We need to add the definition of the new function to the libcipher.h header file. Add a new line to the bottom of that file, so that it looks like this:

```
void cipher_encode(char *text);
void cipher_decode(char *text);
void cipher_version(void);
```

Save the modified libcipher.h file.

We need to compile the cipher_version.c file so that we have a cipher_version.o object file.

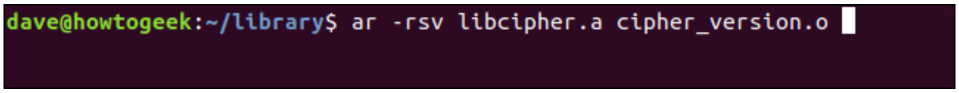
```
gcc -c cipher_version.c
```



```
dave@howtogeek:~/library$ gcc -c cipher_version.c
```

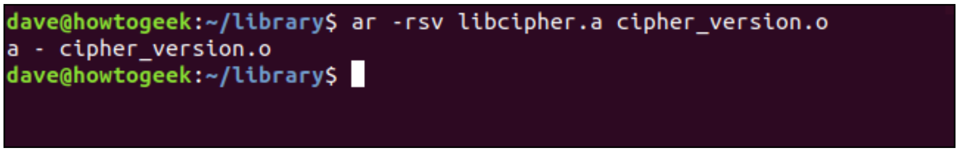
This creates a cipher_version.o file. We can add the new object file to the libcipher.a library with the following command. The `-v` (verbose) option makes the usually silent `ar` tell us what it has done.

```
ar -rsv libcipher.a cipher_version.o
```



```
dave@howtogeek:~/library$ ar -rsv libcipher.a cipher_version.o
```

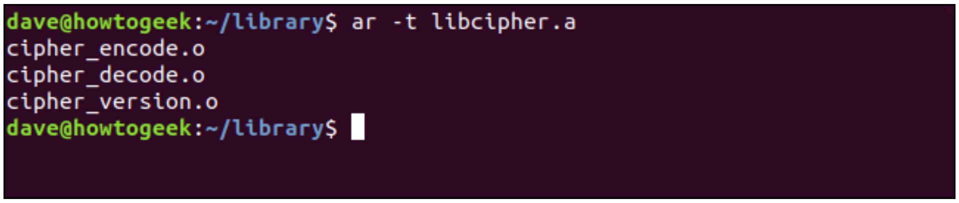
The new object file is added to the library file. `ar` prints out confirmation. The "a" means "added."



```
dave@howtogeek:~/library$ ar -rsv libcipher.a cipher_version.o
a - cipher_version.o
dave@howtogeek:~/library$
```

We can use the `-t` (table) option to see what modules are inside the library file.

```
ar -t libcipher.a
```



```
dave@howtogeek:~/library$ ar -t libcipher.a
cipher_encode.o
cipher_decode.o
cipher_version.o
dave@howtogeek:~/library$
```

There are now three modules inside our library file. Let's make use of the new function.

Using the cipher_version() Function.

Let's remove the old library and header file from the test directory, copy in the new files and then change back into the test directory.

We'll delete the old versions of the files.

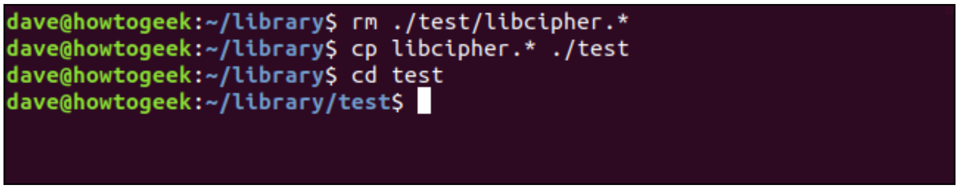
```
rm ./test/libcipher.*
```

We'll copy the new versions into the test directory.

```
cp libcipher.* ./test
```

We'll change into the test directory.

```
cd test
```



```
dave@howtogeek:~/library$ rm ./test/libcipher.*
dave@howtogeek:~/library$ cp libcipher.* ./test
dave@howtogeek:~/library$ cd test
dave@howtogeek:~/library/test$
```

And now we can modify the test.c program so that it uses the new library function.

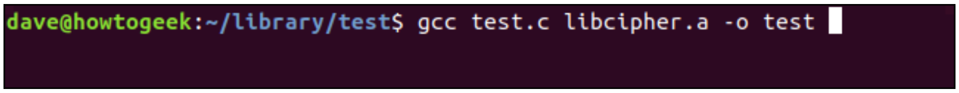
We need to add a new line to the test.c program that calls `cipher_version()` function. We'll place this before the first `puts(text);` line.

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

```
#include <stdlib.h>

#include "libcipher.h"

int main(int argc, char *argv[])
{
 char text[]="How-To Geek loves Linux";

 // new line added here
 cipher_version();

 puts(text);

 cipher_encode(text);
 puts(text);

 cipher_decode(text);
 puts(text);

 exit (0);

} // end of main
```

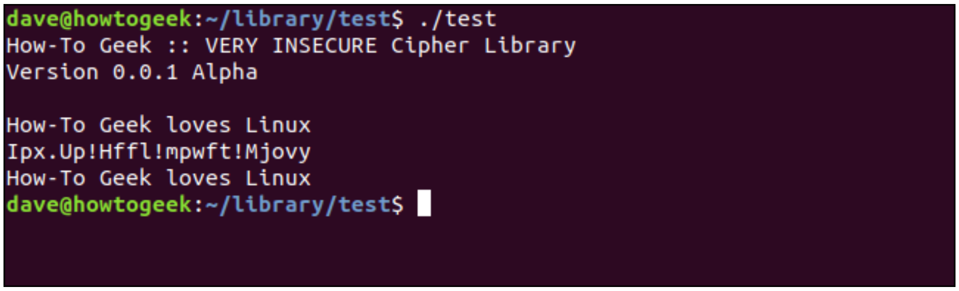
Save this as test.c. We can now compile it and test that the new function is operational.

```
gcc test.c libcipher.a -o test
```



```
dave@howtogeek:~/library/test$ gcc test.c libcipher.a -o test
```

Let's run the new version of `test`:



```
dave@howtogeek:~/library/test$ ./test
How-To Geek :: VERY INSECURE Cipher Library
Version 0.0.1 Alpha

How-To Geek loves Linux
Ipx.Up!Hffl!mpwft!Mjovy
How-To Geek loves Linux
dave@howtogeek:~/library/test$
```

The new function is working. We can see the version of the library at the start of the output from `test`.

But there may be a problem.

Replacing a Module In the Library

This isn't the first version of the library; it's the second. Our version number is incorrect. The first version had no `cipher_version()` function in it. This one does. So this should be version "0.0.2". We need to replace the `cipher_version()` function in the library with a corrected one.

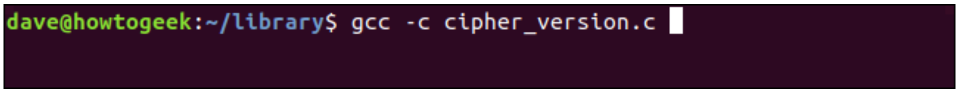
Thankfully, `ar` makes that very easy to do.

First, let's edit the cipher_version.c file in the *library* directory. Change the "Version 0.0.1 Alpha" text to "Version 0.0.2 Alpha". It should look like this:

```
#include <stdio.h>

void cipher_version(void)
{
 puts("How-To Geek :: VERY INSECURE Cipher Library");
 puts("Version 0.0.2 Alpha\n");

} // end of cipher_version
```

Save this file. We need to compile it again to create a new cipher_version.o object file.

```
gcc -c cipher_version.c
```

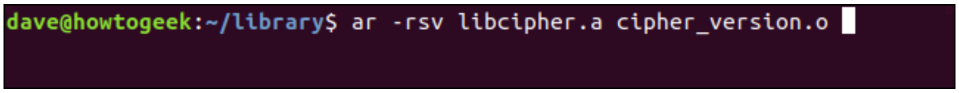


```
dave@howtogeek:~/library$ gcc -c cipher_version.c
```

Now we will *replace* the existing cipher_version.o object in the library with our newly compiled version.

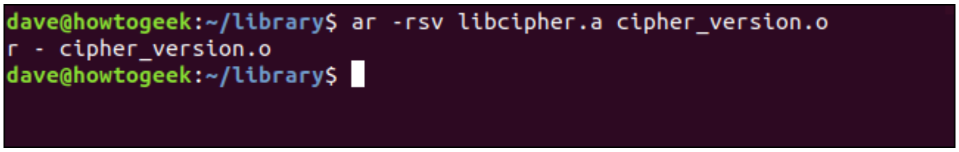
We've used the `-r` (add with replace) option before, to add new modules to the library. When we use it with a module that already exists in the library, `ar` will replace the old version with the new one. The `-s` (index) option will update the library index and the `-v` (verbose) option will make *ar* tell us what it has done.

```
ar -rsv libcipher.a cipher_version.o
```



```
dave@howtogeek:~/library$ ar -rsv libcipher.a cipher_version.o
```

This time *ar* reports that it has replaced the cipher_version.o module. The "r" means replaced.



```
dave@howtogeek:~/library$ ar -rsv libcipher.a cipher_version.o
r - cipher_version.o
dave@howtogeek:~/library$
```

Using the Updated cipher_version() Function

We should use our modified library and check that it works.

We will copy the library files to the test directory.

```
cp libcipher.* ./test
```

We'll change into the test directory.

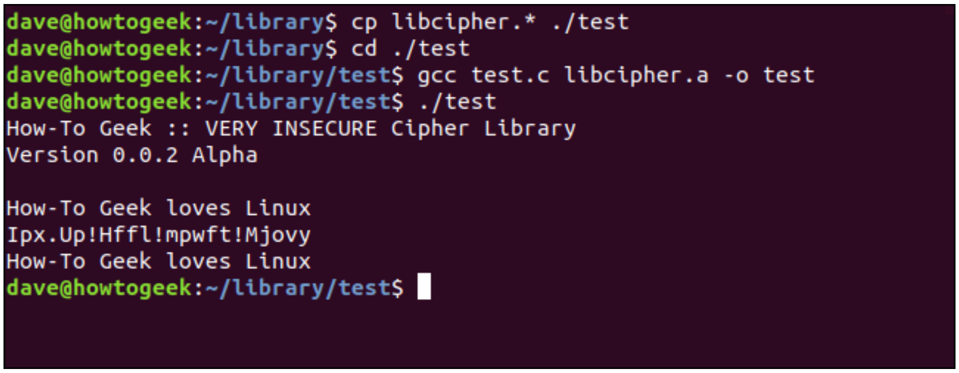
```
cd ./test
```

We need to compile our test program again with our new library.

```
gcc test.c libcipher.a -o test
```

And now we can test our program.

```
./test
```



```
dave@howtogeek:~/library$ cp libcipher.* ./test
dave@howtogeek:~/library$ cd ./test
dave@howtogeek:~/library/test$ gcc test.c libcipher.a -o test
dave@howtogeek:~/library/test$ ./test
How-To Geek :: VERY INSECURE Cipher Library
Version 0.0.2 Alpha

How-To Geek loves Linux
Ipx.Up!Hffl!mpwft!Mjovy
How-To Geek loves Linux
dave@howtogeek:~/library/test$
```

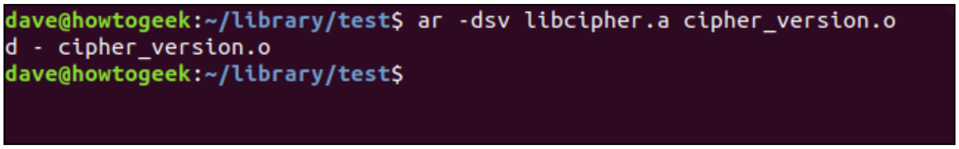
The output from the test program is what we’d expected. The correct version number is showing in the version string, and the encryption and decryption routines are working.

Deleting Modules from a Library

It seems a shame after all that, but let’s delete the cipher_version.o file from the library file.

To do this, we’ll use the `-d` (delete) option. We’ll also use the `-v` (verbose) option, so that *ar* tells us what it has done. We’ll also include the `-s` (index) option to update the index in the library file.

```
ar -dsv libcipher.a cipher_version.o
```

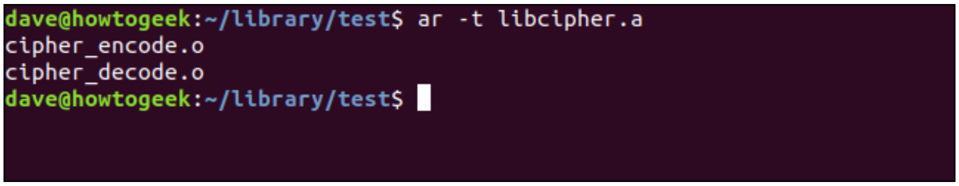


```
dave@howtogeek:~/library/test$ ar -dsv libcipher.a cipher_version.o
d - cipher_version.o
dave@howtogeek:~/library/test$
```

ar reports that it has removed the module. The “d” means “deleted.”

If we ask ar to list the modules inside the library file, we'll see that we are back to two modules.

```
ar -t libcipher.a
```



```
dave@howtogeek:~/library/test$ ar -t libcipher.a
cipher_encode.o
cipher_decode.o
dave@howtogeek:~/library/test$
```

If you are going to delete modules from your library, remember to remove their definition from the library header file.

Share Your Code

Libraries make code shareable in a practical but private way. Anyone that you give the library file and header file to can use your library, but your actual source code remains private.

DAVE MCKAY

Dave McKay first used computers when punched paper tape was in vogue, and he has been programming ever since. After over 30 years in the IT industry, he is now a full-time technology journalist. During his career, he has worked as a freelance programmer, manager of an international software development team, an IT services project manager, and, most recently, as a Data Protection Officer. His writing has been published by howtogeek.com, cloudsavvyit.com, itenterpriser.com, and opensource.com. Dave is a Linux evangelist and open source advocate. **READ FULL BIO »**

The above article may contain affiliate links, which help support How-To Geek.