## **foo.h:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #ifndef foo\_h\_\_  #define foo\_h\_\_    **extern** **void** foo(**void**);    #endif // foo\_h\_\_ |

##foo.c:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #include <stdio.h>  **void** foo(**void**)  {  **puts**("Hello, I am a shared library");  } |

##main.c:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include <stdio.h>  #include "foo.h"    **int** main(**void**)  {  **puts**("This is a shared library test...");  foo();  **return** 0;  } |

foo.h defines the interface to our library, a single function, foo(). foo.c contains the implementation of that function, and main.c is a driver program that uses our library.

For the purposes of this example, everything will happen in /home/username/foo

## **Step 1: Compiling with Position Independent Code**

We need to compile our library source code into position-independent code (PIC):[1](https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html#fn:pic)

$ gcc -c -Wall -Werror -fpic foo.c

## **Step 2: Creating a shared library from an object file**

Now we need to actually turn this object file into a shared library. We will call it libfoo.so:

gcc -shared -o libfoo.so foo.o

## **Step 3: Linking with a shared library**

As you can see, that was actually pretty easy. We have a shared library. Let us compile our main.c and link it with libfoo. We will call our final program test. Note that the -lfoo option is not looking for foo.o, but libfoo.so. GCC assumes that all libraries start with lib and end with .so or .a (.so is for shared object or shared libraries, and .a is for archive, or statically linked libraries).

$ gcc -Wall -o test main.c -lfoo

/usr/bin/ld: cannot find -lfoo

collect2: ld returned 1 exit status

### **Telling GCC where to find the shared library**

Uh-oh! The linker does not know where to find libfoo. GCC has a list of places it looks by default, but our directory is not in that list.[2](https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html#fn:gcclist) We need to tell GCC where to find libfoo.so. We will do that with the -L option. In this example, we will use the current directory, /home/username/foo:

$ gcc -L/home/username/foo -Wall -o test main.c -lfoo

## **Step 4: Making the library available at runtime**

Good, no errors. Now let us run our program:

$ ./test

./test: error while loading shared libraries: libfoo.so: cannot open shared object file: No such file or directory

Oh no! The loader cannot find the shared library.[3](https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html#fn:loadorder) We did not install it in a standard location, so we need to give the loader a little help. We have a couple of options: we can use the environment variable LD\_LIBRARY\_PATH for this, or rpath. Let us take a look first at LD\_LIBRARY\_PATH:

### **Using LD\_LIBRARY\_PATH**

$ echo $LD\_LIBRARY\_PATH

There is nothing in there. Let us fix that by prepending our working directory to the existing LD\_LIBRARY\_PATH:

$ LD\_LIBRARY\_PATH=/home/username/foo:$LD\_LIBRARY\_PATH

$ ./test

./test: error while loading shared libraries: libfoo.so: cannot open shared object file: No such file or directory

What happened? Our directory is in LD\_LIBRARY\_PATH, but we did not export it. In Linux, if you do not export the changes to an environment variable, they will not be inherited by the child processes. The loader and our test program did not inherit the changes we made. Thankfully, the fix is easy:

$ export LD\_LIBRARY\_PATH=/home/username/foo:$LD\_LIBRARY\_PATH

$ ./test

This is a shared library test...

Hello, I am a shared library

Good, it worked! LD\_LIBRARY\_PATH is great for quick tests and for systems on which you do not have admin privileges. As a downside, however, exporting the LD\_LIBRARY\_PATH variable means it may cause problems with other programs you run that also rely on LD\_LIBRARY\_PATH if you do not reset it to its previous state when you are done.