CS-118-02 Week 13

Varun Narravula

2022-04-25

Note: I'm doing all of these notes from my own knowledge based on what topics were touched on in class, because I tested positive for COVID. This may also include some information that I know myself but was never gone over in class (and won't be on quizzes). Take from it what you will.

Introduction To Libraries

When you use functions and classes in C/++, it's a way to create more manageable pieces of code that you can work with, test, and otherwise use. However, what if you want to reuse code from other projects? You can't always fetch the source and directly include it in your project (this is called dependency vendoring, and can happen, but only in specific scenarios). However, what you can do is you can bundle up your functions, classes, namespaces, macros, or whatever it may be into libraries that you can then share and link to as a dependency for your project.

What is a library? As said before, it is a way to contain all of your reusable code together in a single item. On Linux and most other *nix-like systems, these are referred to as `.so`, or **s**hared **o**bject files. On Windows, they're `.dll`, or **d**ynamic **l**ink **l**ibrary files. On macOS and kin (you rarely see these directly), they are called `.dylib`, or **dy**namic **lib**rary files. The Linux term may ring a bell, because you've already seen object files when you compile assembly files using `nasm` or C/++ files with `gcc` and the `-c` flag.

On most Linux and other *nix-like OSes, there is a structure called the FHS (or **F**ilesystem **H**ierarchy **S**tandard) that they comply to, and this standard directs that library/`.so` files should be located in `/lib` and/or `/lib64` (for libraries that core utilities depend on), `/usr/lib` (for user-installed software libraries), and a few other directories that I won't get into for brevity. If you look in here, you will find many other related files, but more often than not many will end in `.so` and a series of numbers that denote a version.

Creating A Static Library

Well, how do you create a library? Let's say you have a header called `add.h` that contains a single function declaration in it:

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

and its corresponding implementation in `add.c`:

```c
#include "add.h"

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

You want to create a library for this that others can reference from when writing code. Granted, this is overkill for such a trivial thing, but for example's sake, how do you do this? You can create a static archive library by compiling the files to an object file first and then using the `ar` command like this:

```
$ gcc -c -o add.o add.c
$ ar rcs add.a add.o
$ ls
add.a  add.c  add.h  add.o
```

The `ar` tool comes standard with the package that provides commands like `ld` and `objdump` (called GNU binutils), and is a trivial zipping system similar to the likes of `.zip` files, except it is only nowadays used for static libraries in this way rather than on normal files and does not compress contents; it has been largely replaced by another tool called `tar` for most purposes. Here, it is used to create a library of object code that other people can then link to in their binaries. However, they will not know the contents of this file without using the header.

You now give this static library archive and header to another person who would like to use some things in it. How do they use it? It's a little more complicated, but here's how an executable project can use it (assume this is a single file called `main.c`):

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

int main() {
  int result = add(1, 2);
  printf("%d\n", result);
  return 0;
}
```

To compile this into an executable binary, you need to let `gcc` know about your library. However, if you try to compile this using `gcc` like you normally do, then you'll run into an inexplicable linker error:

```
$ gcc -o main main.c
/usr/bin/ld: /tmp/ccRj9IzJ.o: in function `main':
main.c:(.text.startup+0xf): undefined reference to `add'
collect2: error: ld returned 1 exit status
```

Uh oh; how do you get around this? Shouldn't the compiler be able to find what `add` is; you already included it, right? Sadly, this is not the case. You need to let `gcc` know where your libraries are. This is done by using a couple of environment variables:

- `$LIBRARY_PATH` :: list of directories to discover static libraries in at compile time
- `$LD_LIBRARY_PATH` :: list of directories to discover dynamic libraries in at runtime

I'll go over the difference between static and dynamic libraries later and into what these variables mean in depth, but for now you only need `$LIBRARY_PATH`; this is a colon-separated list of directories that contain libraries that `gcc` will search in for libraries when compiling your files. To add this library `add.a` to your search path, you can do this temporarily in your compile time path command by using `gcc -o main main.c -L. -l:add.a`; notice the extra options appended after the main.c file. Here is what they do:

- `-L.`: adds `.` (or the current directory) to the `$LIBRARY_PATH` variable for that command
- `-l:add.a`: adds the `add.a` library to the list of compile targets; this is similar to arguments in the past like `-lpthread`, except here it uses a colon to specify the exact name of the library, which is less common but works

The odd thing about this command is that you have to specify dependents before dependencies. Notice how `main.c` is before the other flags, because it is a dependent of those libraries. If you try to mention these arguments before `main.c` as an input, then you will inexplicably run into the same error because `main.c` depends on the contents of the `add.a` static library. This is specifically a quirk of the GNU linker, and not of other linkers such as the LLVM linker `lld`, but it is good to know this to prevent pulling your hair out.

When distributing code using libraries for other people to write code against, you need to provide a way of exposing the code that can be used. The way to do this is by distributing header files along with the libraries. According to the FHS mentioned earlier, these would be stored under a directory called `/usr/include` if you are handling it using a package manager such as the `apt` package manager for Debian-based distributions like Ubuntu. Often the library itself and the header are split into different packages, where only developers that are using the library to compile things with install a package suffixed with `dev` or `devel` that contains the headers for the package, and everyone else that only needs the library to run software with would install the package that only contains the library and not the headers.

You may have noticed that when you included the `add.h` header here, you did so using quotes instead of the angled brackets like for `#include <stdio.h>` or some other builtin library. This is because these builtin header files are a part of the C/++ standard library and the preprocessor can only discover them in paths that are hardcoded at compile time (yes, the preprocessor is compiled too!). You can discover such paths by running `$(gcc -print-prog-name=cpp) -v`, but you would normally never put any files in them, and instead leave header discovery to some tool like `pkg-config`, which I won't go over. This is why you can't include the `add.h` header using the angled brackets, and instead use quotes; when you use quotes, it will search the directory relative to the file being compiled for the header first, and then search the hardcoded system header paths like the angled `#include` does. Make sure to keep this in important fact mind when using headers from other projects; if you know that your headers will be discovered automatically during compilation in these places, then use angled brackets; otherwise, you may need to use quotes.

Static vs. Dynamic Linking

Up until now, you have been compiling your executable using static linking. This means that you are taking the executable code from the library and copying it directly into your executable so you don't need access to the library later. However, this increases an executable's size, and this can get unmanageable, especially if you are trying to optimize for size; if everything on your system is statically compiled, then that means you might have duplicate code from the standard library in all of your binaries, or from other libraries that use the same resources and whatnot. This is what dynamic linking aims to solve, and it puts the meaning of "shared" into "shared object" files.

Dynamic linking is different from static linking in a few ways: most importantly, instead of taking the code from the library used and copying it directly into an executable, it only creates a reference to the function to call and where to call it from. That way, you don't copy in all of the executable code from a library, but only references to what you need. The downside to this is that you will need to have the library present wherever the binary is running. Otherwise, if it is not present, the program will crash at runtime with a message saying the library a function references is not available. This is, for rather plain reasons, not good, and is a downside to dynamic linking and a reason to statically link if you will not have access to that library later. However, your binary will be much smaller, and compiled code is not duplicated as a result of this.

You can see the dynamic libraries a binary is dependent on by using the `ldd` command. Almost all builtin executables on a system are dependent on the C standard library, and we can see this by running `ldd` on a common system command such as `sh` (the builtin shell), your output might vary slightly:

```
$ ldd /bin/sh
  linux-vdso.so.1 (0x00007ffce0baa000)
```

```
  libreadline.so.8 => /usr/lib/libreadline.so.8 (0x00007f8d5728e000)
  libhistory.so.8 => /usr/lib/libhistory.so.8 (0x00007f8d57280000)
  libncursesw.so.6 => /usr/lib/libncursesw.so.6 (0x00007f8d5720b000)
  libdl.so.2 => /usr/lib/libdl.so.2 (0x00007f8d57206000)
  libc.so.6 => /usr/lib/libc.so.6 (0x00007f8d57007000)
  /lib/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2 (0x00007f8d572e8000)
```

You can see that this links to multiple different shared objects that are present in `/usr/lib`, `/lib` or `/lib64`. Some are extremely important (such as) `libc.so.6`, which is the C standard library and `ld-linux-x86-64.so.2`. In fact, you've used the latter when linking an assembly file to use `printf` from the standard library. This file is called the ELF dynamic linker, and every dynamically linked file also dynamically links to this; this is responsible for looking up symbols that are referred to in executables that are dynamically linked, and if this loader cannot resolve a particular symbol, it will stop the program. There are a few other dependencies that this particular binary requires, such as `libncursesw.so.6` (which is a text user interface library), among others. These all have to be present for the binary to run properly.

If you look at the output of the executable that you compiled earlier that you linked earlier, then you will notice that it still links dynamically to the C standard library, because it uses `stdio.h`.

```
$ ldd main
  linux-vdso.so.1 (0x00007ffed0d9d000)
  libc.so.6 => /usr/lib/libc.so.6 (0x00007f59fa8ca000)
  /lib/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2 (0x00007f59faa8d000)
```

You can force statically linking the whole C standard library so that you don't need access to it at all, but this would dramatically increase the size of it. Check out the sizes of forcing static linking of the C library and not:

```
$ gcc -static -lc -o static main.c -L. -l:add.a
$ gcc -o dynamic main.c -L. -l:add.a
$ ls
add.a  add.c  add.h  add.o  dynamic  main.c  static
$ ldd ./static
  Not a dynamic executable
$ ldd ./dynamic
  linux-vdso.so.1 (0x00007ffed0d9d000)
  libc.so.6 => /usr/lib/libc.so.6 (0x00007f59fa8ca000)
  /lib/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2 (0x00007f59faa8d000)
$ du -h ./static
840K  ./static
$ du -h ./dynamic
36K   ./dynamic
```

There is a dramatic difference in size, 840 KB for the statically linked executable, and 36 KB for the dynamically linked executable; a huge difference.

Creating A Dynamic Library

How do you create a dynamic library? It's slightly different:

```
$ gcc -fPIC -c -o add.o add.c
$ gcc -shared -o libadd.so add.o
$ gcc -o main main.c -L. -ladd
$ ldd ./main
  linux-vdso.so.1 (0x00007ffe72162000)
  libadd.so => not found
  libc.so.6 => /usr/lib/libc.so.6 (0x00007f8c5f376000)
  /lib/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2 (0x00007f8c5f539000)
$ du -h ./main
36K   ./main
```

The extra flags do a few things. `-fPIC` means compile the code as "position-independent code", which means that code does not depend on specific address locations in order to work. This is required for shared libraries to function properly; you don't want addresses to have to be the same, because this would be impossible to for the dynamic loader `ld-linux` to ensure. The other flag to create the library, `-shared` is self-explanatory. Also notice how `-ladd` is used instead of `-l:libadd.so`; this is similar syntax to `-lpthread` and other flags you've seen before.

However, if you try running `main` like you normally do, it will fail with this error:

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

Uh oh; it says `libadd.so` is not found. What? It's in the same directory, why is it not found? This cycles back to the `$LD_LIBRARY_PATH` environment variable. This is the difference between `$LIBRARY_PATH` and `$LD_LIBRARY_PATH`; the former is only for libraries that need to be discovered at compile time, and the latter is for libraries that need to be discovered at runtime. You run into this because `libadd.so` is not in `$LD_LIBRARY_PATH`; how can you set it then? You can manually set it by setting the environment variable for your current session or command using `export LD_LIBRARY_PATH=./` or run it temporarily with that by running `LD_LIBRARY_PATH=. main`, but this is clunky. Try dropping `libadd.so` into `/usr/lib` and see what happens:

```
$ sudo mv ./libadd.so /usr/lib
Password:
$ ./main
3
$ sudo rm /usr/lib/libadd.so
$ ./main
./main: error while loading shared libraries: libadd.so: cannot open shared
object file: No such file or directory
```

It works when it's in here! This is because LD_LIBRARY_PATH is predefined with a few directories, including `/usr/lib`. The dynamic linker looks in these automatically, so you don't need to specify LD_LIBRARY_PATH anymore. In fact, packages normally drop their libraries into here or some other place that is predefined in `LD_LIBRARY_PATH` to avoid having to continually add paths to this environment variable for executables to work.

Inspecting Symbols with `nm`

You can inspect the symbols of a library to debug linker errors by using the `nm` command that comes with GNU binutils. To inspect the dynamic `libadd.so` library, run this command:

```
$ nm libadd.so
0000000000001100 T add
0000000000004008 b completed.0
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000001040 t deregister_tm_clones
00000000000010b0 t __do_global_dtors_aux
0000000000003df0 d __do_global_dtors_aux_fini_array_entry
0000000000004000 d __dso_handle
0000000000003df8 d _DYNAMIC
0000000000001104 t _fini
00000000000010f0 t frame_dummy
0000000000003de8 d __frame_dummy_init_array_entry
0000000000002094 r __FRAME_END__
0000000000003fc8 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000002000 r __GNU_EH_FRAME_HDR
0000000000001000 t _init
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000001070 t register_tm_clones
0000000000004008 d __TMC_END__
```

This output has three columns: one for a symbol value, symbol type, and its name. You can look at what a library contains using this; you see the `add` symbol, which has a type of T (global text symbol, which means it's in the .text section most likely), along with many other symbols that help the binary run.

However, what if you tried to compile a shared library with C++? You'll see something weird. Try compiling the `libadd.so` library using `g++` instead of `gcc`:

```
$ g++ -fPIC -c -o add.o add.cpp
$ g++ -shared -o libadd.so add.o
$ nm ./libadd.so
0000000000004008 b completed.0
                 w __cxa_finalize@@GLIBC_2.2.5
```

```
0000000000001040 t deregister_tm_clones
00000000000010b0 t __do_global_dtors_aux
0000000000003dc0 d __do_global_dtors_aux_fini_array_entry
0000000000004000 d __dso_handle
0000000000003dc8 d _DYNAMIC
0000000000001104 t _fini
00000000000010f0 t frame_dummy
0000000000003db8 d __frame_dummy_init_array_entry
0000000000002094 r __FRAME_END__
0000000000003fc8 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000002000 r __GNU_EH_FRAME_HDR
0000000000001000 t _init
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000001070 t register_tm_clones
0000000000004008 d __TMC_END__
0000000000001100 T _Z3addii
```

Now the `add` symbol has much more information appended to it. Do you remember name mangling? This is C++'s name mangling in action. It adds extra information to the name of the function in order to distinguish it from other functions in other namespaces, classes, and whatnot. Here, it specifies `ii` for the arguments (int, int), among other things. This only happens when you do not specify `extern "C"` and instead only specify `extern` for your function declaration in your header, and is the reason for `extern C` in order to be able to compile assembly files. Technically, you can work with mangled names, but it's extremely inconvenient, as evidenced from this output.

If you specified `extern "C"` in the header instead of `extern`, then you get this:

```
$ g++ -fPIC -c -o add.o add.cpp
$ g++ -shared -o libadd.so add.o
$ nm ./libadd.so
0000000000001100 T add
0000000000004008 b completed.0
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000001040 t deregister_tm_clones
00000000000010b0 t __do_global_dtors_aux
0000000000003dc0 d __do_global_dtors_aux_fini_array_entry
0000000000004000 d __dso_handle
0000000000003dc8 d _DYNAMIC
0000000000001104 t _fini
00000000000010f0 t frame_dummy
0000000000003db8 d __frame_dummy_init_array_entry
0000000000002094 r __FRAME_END__
0000000000003fc8 d _GLOBAL_OFFSET_TABLE_
```

```
                 w __gmon_start__
0000000000002000 r __GNU_EH_FRAME_HDR
0000000000001000 t _init
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000001070 t register_tm_clones
0000000000004008 d __TMC_END__
```

Here, you see a normal `add` symbol, which is much easier to link to. I'm not going to go too much into this, because you rarely need to inspect library symbols like this, but that's how you do it.