v1.0.1

Building And Using Static And Shared "C" Libraries

Table Of Contents:

Building And Using Static And Shared "C" Libraries

One of the problems with developed programs, is that they tend to grow larger and larger, bringing up overall compilation and linking time to a large figure, and polluting out makefile, and the directory where we placed the source files. The first time a program we write reaches this state, is normally when we look for a different way to manage our projects.

It is this point where we start thinking about combining out source code into small units of related files, that can be managed with a separate makefile, possibly by a different programmer (for a multi-programmer project).

What Is A "C" Library? What Is It Good For?

One of the tools that compilers supply us with are libraries. A library is a file containing several object files, that can be used as a single entity in a linking phase of a program. Normally the library is indexed, so it is easy to find symbols (functions, variables and so on) in them. For this reason, linking a program whose object files are ordered in libraries is faster than linking a program whose object files are separate on the disk. Also, when using a library, we have fewer files to look for and open, which even further speeds up linking.

Unix systems (as well as most other modern systems) allow us to create and use two kinds of libraries - static libraries and shared (or dynamic) libraries.

Static libraries are just collections of object files that are linked into the program during the linking phase of compilation, and are not relevant during runtime. This last comment seems obvious, as we already know that object files are also used only during the linking phase, and are not required during runtime - only the program's executable file is needed in order to run the program.

Shared libraries (also called dynamic libraries) are linked into the program in two stages. First, during compile time, the linker verifies that all the symbols (again, functions, variables and the like) required by the program, are either linked into the program, or in one of its shared libraries. However, the object files from the dynamic library are not inserted into the executable file. Instead, when the program is started, a program in the system (called a dynamic loader) checks out which shared libraries were linked with the program, loads them to memory, and attaches them to the copy of the program in memory.

The complex phase of dynamic loading makes launching the program slightly slower, but this is a very insignificant drawback, that is out-weighted by a great advantage - if a second program linked with the same shared library is executed, it can use the same copy of the shared library, thus saving a lot of memory. For example, the standard "C" library is normally a shared library, and is used by all C programs. Yet, only one copy of the library is stored in memory at any given time. This means we can use far less memory to run our programs, and the executable files are much smaller, thus saving a lot of disk space as well.

However, there is one drawback to this arrangement. If we re-compile the dynamic library and try to run a second copy of our program with the new library, we'll soon get stuck - the dynamic loader will find that a copy of the library is already stored in memory, and thus will attach it to our program, and not load the new (modified) version from disk. There are ways around this too, as we'll see in the last section of our discussion.

Creating A Static "C" Library Using "ar" and "ranlib"

The basic tool used to create static libraries is a program called `'ar'`, for 'archiver'. This program can be used to create static libraries (which are actually archive files), modify object files in the static library, list the names of object files in the library, and so on. In order to create a static library, we can use a command like this:

```
ar rc libutil.a util_file.o util_net.o util_math.o
```

This command creates a static library named 'libutil.a' and puts copies of the object files "util_file.o", "util_net.o" and "util_math.o" in it. If the library file already exists, it has the object files added to it, or replaced, if they are newer than those inside the library. The `'c'` flag tells ar to create the library if it doesn't already exist. The `'r'` flag tells it to replace older object files in the library, with the new object files.

After an archive is created, or modified, there is a need to index it. This index is later used by the compiler to speed up symbol-lookup inside the library, and to make sure that the order of the symbols in the library won't matter during compilation (this will be better understood when we take a deeper look at the link process at the end of this tutorial). The command used to create or update the index is called `'ranlib'`, and is invoked as follows:

```
ranlib libutil.a
```

On some systems, the archiver (which is not always `ar`) already takes care of the index, so ranlib is not needed (for example, when Sun's C compiler creates an archive, it is already indexed). However, because `'ar'` and `'ranlib'` are used by many makefiles for many packages, such platforms tend to supply a ranlib command that does nothing. This helps using the same makefile on both types of platforms.

Note: when an archive file's index generation date (stored inside the archive file) is older than the file's last modification date (stored in the file system), a compiler trying to use this library will complain its index is out of date, and abort. There are two ways to overcome the problem:

1. Use `'ranlib'` to re-generate the index.
2. When copying the archive file to another location, use `'cp -p'`, instead of only `'cp'`. The `'-p'` flag tells `'cp'` to keep all attributes of the file, including its access permissions, owner (if "cp" is invoked by a superuser) and its last modification date. This will cause the compiler to think the index inside the file is still updated. This method is useful for makefiles that need to copy the library to another directory for some reason.

Using A "C" Library In A Program

After we created our archive, we want to use it in a program. This is done by adding the library's name to the list of object file names given to the linker, using a special flag, normally `'-l'`. Here is an example:

```
cc main.o -L. -lutil -o prog
```

This will create a program using object file "main.o", and any symbols it requires from the "util" static library. Note that we omitted the "lib" prefix and the ".a" suffix when mentioning the library on the link command. The linker attaches these parts back to the name of the library to create a name of a file to look for. Note also the usage of the `'-L'` flag - this flag tells the linker that libraries might be found in the given directory ('.', refering to the current directory), in addition to the standard locations where the compiler looks for system libraries.

For an example of program that uses a static library, try looking at our static library example directory.

Creating A Shared "C" Library Using "ld"

The creation of a shared library is rather similar to the creation of a static library. Compile a list of object files, then insert them all into a shared library file. However, there are two major differences:

1. Compile for "Position Independent Code" (PIC) - When the object files are generated, we have no idea where in memory they will be inserted in a program that will use them. Many different programs may use the same library, and each load it into a different memory in address. Thus, we need that all jump calls ("goto", in assembly speak) and subroutine calls will use relative addresses, and not absolute addresses. Thus, we need to use a compiler flag that will cause this type of code to be generated.
   In most compilers, this is done by specifying `'-fPIC'` or `'-fpic'` on the compilation command.
2. Library File Creation - unlike a static library, a shared library is not an archive file. It has a format that is specific to the architecture for which it is being created. Thus, we need to use the compiler (either the compiler's driver, or its linker) to generate the library, and tell it that it should create a shared library, not a final program file.
   This is done by using the `'-G'` flag with some compilers, or the `'-shared'` flag with other compilers.

Thus, the set of commands we will use to create a shared library, would be something like this:

```
cc -fPIC -c util_file.c
cc -fPIC -c util_net.c
cc -fPIC -c util_math.c
cc -shared libutil.so util_file.o util_net.o util_math.o
```

The first three commands compile the source files with the `PIC` option, so they will be suitable for use in a shared library (they may still be used in a program directly, even thought they were compiled with `PIC`). The last command asks the compiler to generate a shared library

---

Using A Shared "C" Library - Quirks And Solutions

Using a shared library is done in two steps:

1. Compile Time - here we need to tell the linker to scan the shared library while building the executable program, so it will be convinced that no symbols are missing. It will not really take the object files from the shared library and insert them into the program.
2. Run Time - when we run the program, we need to tell the system's dynamic loader (the process in charge of automatically loading and linking shared libraries into the running process) where to find our shared library.

The compilation part is easy. It is done almost the same as when linking with static libraries:

```
cc main.o -L. -lutil -o prog
```

The linker will look for the file 'libutil.so' (`-lutil`) in the current directory (`-L.`), and link it to the program, but will not place its object files inside the resulting executable file, 'prog'.

The run-time part is a little trickier. Normally, the system's dynamic loader looks for shared libraries in some system specified directories (such as /lib, /usr/lib, /usr/X11/lib and so on). When we build a new shared library that is not part of the system, we can use the `'LD_LIBRARY_PATH'` environment variable to tell the dynamic loader to look in other directories. The way to do that depends on the type of shell we use ('tcsh' and 'csh', versus 'sh', 'bash', 'ksh' and similar shells), as well as on whether or not `'LD_LIBRARY_PATH'` is already defined. To check if you have this variable defined, try:

```
echo $LD_LIBRARY_PATH
```

If you get a message such as `'LD_LIBRARY_PATH: Undefined variable.'`, then it is not defined.

Here is how to define this variable, in all four cases:

1. 'tcsh' or 'csh', `LD_LIBRARY_PATH` is not defined:

   ```
   setenv LD_LIBRARY_PATH /full/path/to/library/directory
   ```

2. 'tcsh' or 'csh', `LD_LIBRARY_PATH` already defined:

   ```
   setenv LD_LIBRARY_PATH /full/path/to/library/directory:${LD_LIBRARY_PATH}
   ```

3. 'sh', 'bash' and similar, `LD_LIBRARY_PATH` is not defined:

   ```
   LD_LIBRARY_PATH=/full/path/to/library/directory
   export LD_LIBRARY_PATH
   ```

4. 'sh', 'bash' and similar, `LD_LIBRARY_PATH` already defined:

   ```
   LD_LIBRARY_PATH=/full/path/to/library/directory:${LD_LIBRARY_PATH}
   export LD_LIBRARY_PATH
   ```

After you've defined `LD_LIBRARY_PATH`, you can check if the system locates the library properly for a given program linked with this library:

```
ldd prog
```

You will get a few lines, each listing a library name on the left, and a full path to the library on the right. If a library is not found in any of the system default directories, or the directories mentioned in `'LD_LIBRARY_PATH'`, you will get a 'library not found' message. In such a case, verify that you properly defined the path to the directory inside `'LD_LIBRARY_PATH'`, and fix it, if necessary. If all goes well, you can run your program now like running any other program, and see it role...

For an example of a program that uses a shared library, try looking at our shared library example directory.

Using A Shared "C" Library Dynamically - Programming Interface

One of the less-commonly used feature of shared libraries is the ability to link them to a process anytime during its life. The linking method we showed earlier makes the shared library automatically loaded by the dynamic loader of the system. Yet, it is possible to make a linking operation at any other time, using the 'dl' library. This library provides us with a means to load a shared library, reference any of its symbols, call any of its functions, and finally detach it from the process when no longer needed.

Here is a scenario where this might be appealing: suppose that we wrote an application that needs to be able to read files created by different word processors. Normally, our program might need to be able to read tens of different file formats, but in a single run, it is likely that only one or two such document formats will be needed. We could write one shared library for each such format, all having the same interface (readfile and writefile for example), and one piece of code that determines the file format. Thus, when our program is asked to open such a file, it will first determine its format, then load the relevant shared library that can read and translate that format, and call its readfile function to read the document. We might have tens of such libraries, but only one of them will be placed in memory at any given time, making our application use less system resources. It will also allow us to ship the application with a small set of supported file formats, and add new file formats without the need to replace the whole application, by simply sending the client an additional set of shared libraries.

Loading A Shared Library Using `dlopen()`

In order to open and load the shared library, one should use the `dlopen()` function. It is used this way:

```
#include <dlfcn.h>      /* defines dlopen(), etc.       */
.
.
void* lib_handle;       /* handle of the opened library */

lib_handle = dlopen("/full/path/to/library", RTLD_LAZY);
if (!lib_handle) {
    fprintf(stderr, "Error during dlopen(): %s\n", dlerror());
    exit(1);
}
```

The `dlopen()` function gets two parameters. One is the full path to the shared library. The other is a flag defining whether all symbols refered to by the library need to be checked immediatly, or only when used. In our case, we may use the lazy approach (`RTLD_LAZY`) of checking only when used. The function returns a pointer to the loaded library, that may later be used to reference symbols in the library. It will return `NULL` in case an error occured. In that case, we may use the `dlerror()` function to print out a human-readable error message, as we did here.

Calling Functions Dynamically Using `dlsym()`

After we have a handle to a loaded shared library, we can find symbols in it, of both functions and variables. We need to define their types properly, and we need to make sure we made no mistakes. The compiler won't be able to check those declarations, so we should be extra carefull when typing them. Here is how to find the address of a function named 'readfile' that gets one string parameter, and returns a pointer to a `'struct local_file'` structure:

```
/* first define a function pointer variable to hold the function's address */
struct local_file* (*readfile)(const char* file_path);
/* then define a pointer to a possible error string */
const char* error_msg;
/* finally, define a pointer to the returned file */
struct local_file* a_file;

/* now locate the 'readfile' function in the library */
readfile = dlsym(lib_handle, "readfile");

/* check that no error occured */
error_msg = dlerror();
if (error_msg) {
    fprintf(stderr, "Error locating 'readfile' - %s\n", error_msg);
    exit(1);
}

/* finally, call the function, with a given file path */
a_file = (*readfile)("hello.txt");
```

As you can see, errors might occur anywhere along the code, so we should be carefull to make extensive error checking. Surely, you'll also check that 'a_file' is not `NULL`, after you call your function.

Unloading A Shared Library Using `dlclose()`

The final step is to close down the library, to free the memory it occupies. This should only be done if we are not intending to use it soon. If we do - it is better to leave it open, since library loading takes time. To close down the library, we use something like this:

```
dlclose(lib_handle);
```

This will free down all resources taken by the library (in particular, the memory its executable code takes up).

Automatic Startup And Cleanup Functions

Finally, the dynamic loading library gives us the option of defining two special functions in each library, namely `_init` and `_fini`. The `_init` function, if found, is invoked automatically when the library is opened, and before `dlopen()` returns. It may be used to invoke some startup code needed to initialize data structures used by the library, read configuration files, and so on.

The `_fini` function is called when the library is closed using `dlclose()`. It may be used to make cleanup operations required by the library (freeing data structures, closing files, etc.).

For an example of a program that uses the 'dl' interface, try looking at our dynamic-shared library example directory.

Getting a Deeper Understanding - The Complete Linking Story

The Importance Of Linking Order

In order to fully understand the way linking is done, and be able to overcome linking problems, we should bare in mind that the order in which we present the object files and the libraries to the linker, is the order in which the linker links them into the resulting binary file.

The linker checks each file in turn. If it is an object file, it is being placed fully into the executable file. If it is a library, the linker checks to see if any symbols referenced (i.e. used) in the previous object files but not defined (i.e. contained) in them, are in the library. If such a symbol is found, the whole object file from the library that contains the symbol - is being added to the executable file. This process continues until all object files and libraries on the command line were processed.

This process means that if library 'A' uses symbols in library 'B', then library 'A' has to appear on the link command before library 'B'. Otherwise, symbols might be missing - the linker never turns back to libraries it has already processed. If library 'B' also uses symbols found in library 'A' - then the only way to assure successful linking is to mention library 'A' on the link command again after library 'B', like this:

```
$(LD) ....... -lA -lB -lA
```

This means that linking will be slower (library 'A' will be processed twice). This also hints that one should try not to have such mutual dependencies between two libraries. If you have such dependencies - then either re-design your libraries' contents, or combine the two libraries into one larger library.

Note that object files found on the command line are always fully included in the executable file, so the order of mentioning them does not really matter. Thus, a good rule is to always mention the libraries after all object files.

Static Linking Vs. Dynamic Linking

When we discussed static libraries we said that the linker will try to look for a file named 'libutil.a'. We lied. Before looking for such a file, it will look for a file named 'libutil.so' - as a shared library. Only if it cannot find a shared library, will it look for 'libutil.a' as a static library. Thus, if we have created two copies of the library, one static and one shared, the shared will be preferred. This can be overridden using some linker flags (`'-Wl,static'` with some linkers, `'-Bstatic'` with other types of linkers. refer to the compiler's or the linker's manual for info about these flags).