Program Library HOWTO

David A. Wheeler

version 0.87, 22 March 2001

This HOWTO for programmers discusses how to create and use program libraries on Linux. This includes static libraries, shared libraries, and dynamically loaded libraries.

Table of Contents

1. Introduction

This HOWTO for programmers discusses how to create and use program libraries on Linux using the GNU toolset. A ``program library'' is simply a file containing compiled code (and data) that is to be incorporated later into a program; program libraries allow programs to be more modular, faster to recompile, and easier to update. Program libraries can be divided into three types: static libraries, shared libraries, and dynamically loaded (DL) libraries.

This paper first discusses static libraries, which are installed into a program executable before the program is run. It then discusses shared libraries, which are loaded at program start−up and shared between programs. Finally, it discusses dynamically loaded (DL) libraries, which can be loaded and used at any time while a program is running. DL libraries aren't really a different kind of library format (both static and shared libraries can be used as DL libraries); instead, the difference is in how DL libraries are used by programmers. The HOWTO wraps up with a section with more examples and a section with references to other sources of information.

This HOWTO discusses only the Executable and Linking Format (ELF) format for executables and libraries, the format used by nearly all Linux distributions today. The GNU gcc toolset can actually handle library formats other than ELF; in particular, most Linux distributions can still use the obsolete a.out format. However, these formats are outside the scope of this paper.

It's worth noting that some people use the term dynamically *linked* libraries (DLLs) to refer to shared libraries, some use the term DLL to mean any library that is used as a DL library, and some use the term DLL to mean a library meeting either condition. No matter which meaning you pick, this HOWTO covers DLLs on Linux.

If you're building an application that should port to many systems, you might consider using GNU libtool to build and install libraries instead of using the Linux tools directly. GNU libtool is a generic library support script that hides the complexity of using shared libraries (e.g., creating and installing them) behind a consistent, portable interface. On Linux, GNU libtool is built on top of the tools and conventions described in this HOWTO. For a portable interface to dynamically loaded libraries, you can use various portability wrappers. GNU libtool includes such a wrapper, called ``libltdl''. Alternatively, you could use the glib library (not to be confused with glibc) with its portable support for Dynamic Loading of Modules. You can learn more about glib at http://developer.gnome.org/doc/API/glib/glib−dynamic−loading−of−modules.html. Again, on Linux this functionality is implemented using the constructs described in this HOWTO. If you're actually developing or debugging the code on Linux, you'll probably still want the information in this HOWTO.

This HOWTO's master location is http://www.dwheeler.com/program−library, and it has been contributed to the Linux Documentation Project (http://www.linuxdoc.org).

2. Static Libraries

Static libraries are simply a collection of ordinary object files; conventionally, static libraries end with the ``.a'' suffix. This collection is created using the ar (archiver) program. Static libraries aren't used as often as they once were, because of the advantages of shared libraries (described below). Still, they're sometimes created, they existed first historically, and they're simpler to explain.

Static libraries permit users to link to programs without having to recompile its code, saving recompilation time. Note that recompilation time is less important given today's faster compilers, so this reason is not as strong as it once was. Static libraries are often useful for developers if they wish to permit programmers to link to their library, but don't want to give the library source code (which is an advantage to the library vendor, but obviously not an advantage to the programmer trying to use the library). In theory, code in static ELF libraries that is linked into an executable should run slightly faster (by 1−5%) than a shared library or a dynamically loaded library, but in practice this rarely seems to be the case due to other confounding factors.

To create a static library, or to add additional object files to an existing static library, use a command like this:

```
ar rcs my_library.a file1.o file2.o
```

This sample command adds the object files file1.o and file2.o to the static library my_library.a, creating my_library.a if it doesn't already exist. For more information on creating static libraries, see ar(1).

Once you've created a static library, you'll want to use it. You can use a static library by invoking it as part of the compilation and linking process when creating a program executable. If you're using gcc(1) to generate your executable, you can use the −l option to specify the library; see info:gcc for more information. You can also use the linker ld(1) directly, using its −l and −L options; however, in most cases it's better to use gcc(1) since the interface of ld(1) is more likely to change.

3. Shared Libraries

Shared libraries are libraries that are loaded by programs when they start. When a shared library is installed properly, all programs that start afterwards automatically use the new shared library. It's actually much more flexible and sophisticated than this, because the approach used by Linux permits you to:

- update libraries and still support programs that want to use older, non−backward−compatible versions of those libraries;
- override specific libraries or even specific functions in a library when executing a particular program.
- do all this while programs are running using existing libraries.

3.1. Conventions

For shared libraries to support all of these desired properties, a number of conventions and guidelines must be followed. You need to understand the difference between a library's names, in particular its ``soname'' and ``real name'' (and how they interact). You also need to understand where they should be placed in the filesystem.

3.1.1. Shared Library Names

Every shared library has a special name called the ``soname''. The soname has the prefix ``lib'', the name of the library, the phrase ``.so'', followed by a period and a version number that is incremented whenever the interface changes (as a special exception, the lowest−level C libraries don't start with ``lib''). A fully−qualified soname includes as a prefix the directory it's in; on a working system a fully−qualified soname is simply a symbolic link to the shared library's ``real name''.

Every shared library also has a ``real name'', which is the filename containing the actual library code. The real name adds to the soname a period, a minor number, another period, and the release number. The last period and release number are optional. The minor number and release number support configuration control by letting you know exactly what version(s) of the library are installed. Note that these numbers might not be the same as the numbers used to describe the library in documentation, although that does make things easier.

In addition, there's the name that the compiler uses when requesting a library, (I'll call it the ``linker name''), which is simply the soname without any version number.

The key to managing shared libraries is the separation of these names. Programs, when they internally list the shared libraries they need, should only list the soname they need. Conversely, when you create a shared library, you only create the library with a specific filename (with more detailed version information). When you install a new version of a library, you install it in one of a few special directories and then run the program ldconfig(8). ldconfig examines the existing files and creates the sonames as symbolic links to the real names, as well as setting up the cache file /etc/ld.so.cache (described in a moment).

ldconfig doesn't set up the linker names; typically this is done during library installation, and the linker name is simply created as a symbolic link to the ``latest'' soname or the latest real name. I would recommend having the linker name be a symbolic link to the soname, since in most cases if you update the library you'd like to automatically use it when linking. I asked H. J. Lu why ldconfig doesn't automatically set up the linker names. His explanation was basically that you might want to run code using the latest version of a library, but might instead want *development* to link against an old (possibly incompatible) library. Therefore, ldconfig makes no assumptions about what you want programs to link to, so installers must specifically modify symbolic links to update what the linker will use for a library.

Thus, `/usr/lib/libreadline.so.3` is a fully−qualified soname, which ldconfig would set to be a symbolic link to some realname like `/usr/lib/libreadline.so.3.0`. There should also be a linker name, `/usr/lib/libreadline.so` which could be a symbolic link referring to `/usr/lib/libreadline.so.3`.

3.1.2. Filesystem Placement

Shared libraries must be placed somewhere in the filesystem. Most open source software tends to follow the GNU standards; for more information see the info file documentation at info:standards#Directory_Variables. The GNU standards recommend installing by default all libraries in /usr/local/lib when distributing source code (and all commands should go into /usr/local/bin). They also define the convention for overriding these defaults and for invoking the installation routines.

The Filesystem Hierarchy Standard (FHS) discusses what should go where in a distribution (see http://www.pathname.com/fhs). According to the FHS, most libraries should be installed in /usr/lib, but libraries required for startup should be in /lib and libraries that are not part of the system should be in /usr/local/lib.

There isn't really a conflict between these two documents; the GNU standards recommend the default for developers of source code, while the FHS recommends the default for distributors (who selectively override the source code defaults, usually via the system's package management system). In practice this works nicely: the ``latest'' (possibly buggy!) source code that you download automatically installs itself in the ``local'' directory (`/usr/local`), and once that code has matured the package managers can trivially override the default to place the code in the standard place for distributions. Note that if your library calls programs that can only be called via libraries, you should place those programs in /usr/local/libexec (which becomes /usr/libexec in a distribution). One complication is that Red Hat−derived systems don't include /usr/local/lib by default in their search for libraries; see the discussion below about /etc/ld.so.conf. Other standard library locations include /usr/X11R6/lib for X−windows. Note that /lib/security is used for PAM modules, but those are usually loaded as DL libraries (also discussed below).

3.2. How Libraries are Used

On GNU glibc−based systems, including all Linux systems, starting up an ELF binary executable automatically causes the program loader to be loaded and run. On Linux systems, this loader is named /lib/ld−linux.so.X (where X is a version number). This loader, in turn, finds and loads all other shared libraries used by the program.

The list of directories to be searched is stored in the file /etc/ld.so.conf. Many Red Hat−derived distributions don't normally include /usr/local/lib in the file /etc/ld.so.conf. I consider this a bug, and adding /usr/local/lib to /etc/ld.so.conf is a common ``fix'' required to run many programs on Red Hat−derived systems.

If you want to just override a few functions in a library, but keep the rest of the library, you can enter the names of overriding libraries (.o files) in /etc/ld.so.preload; these ``preloading'' libraries will take precedence over the standard set. This preloading file is typically used for emergency patches; a distribution usually won't include such a file when delivered.

Searching all of these directories at program start−up would be grossly inefficient, so a caching arrangement is actually used. The program ldconfig(8) by default reads in the file /etc/ld.so.conf, sets up the appropriate symbolic links in the dynamic link directories (so they'll follow the standard conventions), and then writes a cache to /etc/ld.so.cache that's then used by other programs. This greatly speeds up access to libraries. The implication is that ldconfig must be run whenever a DLL is added, when a DLL is removed, or when the set of DLL directories changes; running ldconfig is often one of the steps performed by package managers when installing a library. On start−up, then, the dynamic loader actually uses the file /etc/ld.so.cache and then loads the libraries it needs.

By the way, FreeBSD uses slightly different filenames for this cache. In FreeBSD, the ELF cache is /var/run/ld−elf.so.hints and the a.out cache is /var/run/ld.so.hints. These are still updated by ldconfig(8), so this difference in location should only matter in a few exotic situations.

3.3. Environment Variables

Various environment variables can control this process, and in fact there are environment variables that permit you to override this process. For example, you can temporarily substitute a different library for this particular execution. In Linux, the environment variable LD_LIBRARY_PATH is a colon−separated set of directories where libraries should be searched for first, before the standard set of directories; this is useful when debugging a new library or using a nonstandard library for special purposes. The environment variable LD_PRELOAD lists shared libraries with functions that override the standard set, just as /etc/ld.so.preload does. These are implemented by the loader /lib/ld−linux.so. I should note that, while LD_LIBRARY_PATH works on many Unix−like systems, it doesn't work on all; for example, this functionality is available on HP−UX but as the environment variable SHLIB_PATH, and on AIX this functionality is through the variable LIBPATH (with the same syntax, a colon−separated list).

LD_LIBRARY_PATH is handy for development and testing, but shouldn't be modified by an installation process for normal use by normal users; see ``Why LD_LIBRARY_PATH is Bad'' at http://www.visi.com/~barr/ldpath.html for an explanation of why. But it's still useful for development or testing, and for working around problems that can't be worked around otherwise.

There are actually a number of other environment variables that control the loading process; their names begin with LD_ or RTLD_. Most of the others are for low−level debugging of the loader process or for implementing specialized capabilities. Most of them aren't well−documented; if you need to know about them, the best way to learn about them is to read the source code of the loader (part of gcc).

Permitting user control over dynamically linked libraries would be disastrous for setuid/setgid programs if special measures weren't taken. Therefore, in the GNU loader (which loads the rest of the program on program start−up), if the program is setuid or setgid these variables (and other similar variables) are ignored or greatly limited in what they can do. The loader determines if a program is setuid or setgid by checking the program's credentials; if the uid and euid differ, or the gid and the egid differ, the loader presumes the program is setuid/setgid (or descended from one) and therefore greatly limits its abilities to control linking. If you read the GNU glibc library source code, you can see this; see especially the files elf/rtld.c and sysdeps/generic/dl−sysdep.c. This means that if you cause the uid and gid to equal the euid and egid, and then call a program, these variables will have full effect. Other Unix−like systems handle the situation differently but for the same reason: a setuid/setgid program should not be unduly affected by the environment variables set.

3.4. Creating a Shared Library

Creating a shared library is easy. First, create the object files that will go into the shared library using the gcc −fPIC flag (this enables the ``position independent code'' generation, a requirement for shared libraries). Then create the shared library using this format:

```
gcc −shared −Wl,−soname,your_soname \
    −o library_name file_list library_list
```

Here's an example, which creates two object files (a.o and b.o) and then creates a shared library that contains both of them. Note that this compilation includes debugging information (−g) and will generate warnings (−Wall), which aren't required for shared libraries but are recommended. The compilation generates object files (using −c), and obviously includes the required −fPIC option:

```
gcc −fPIC −g −c −Wall a.c
gcc −fPIC −g −c −Wall b.c
gcc −shared −Wl,−soname,libmystuff.so.1 \
    −o libmystuff.so.1.0.1 a.o b.o −lc
```

Here are a few points worth noting:

- Don't strip the resulting library, and don't use the compiler option −fomit−frame−pointer unless you really have to. The resulting library will work, but these actions make debuggers mostly useless.
- Use −fPIC to generate code, not −fpic (the latter may not work, because if branches need large displacements −fpic may not generate fully position−independent code).
- In some cases, the call to gcc to create the object file will also need to include the option ``−Wl,−export−dynamic''. Normally, the dynamic symbol table contains only symbols which are used by a dynamic object. This option (when creating an ELF file) adds all symbols to the dynamic symbol table (see ld(1) for more information). You use this option when there are 'reverse dependencies', i.e., a DL library has unresolved symbols that by convention must be defined in the programs that intend to load these libraries. For ``reverse dependencies'' to work, the master program must make its symbols dynamically available. Note that you could say ``−rdynamic'' instead of ``−Wl,export−dynamic'' if you only work with Linux systems, but according to the ELF documentation the ``−rdynamic'' flag doesn't always work for gcc on non−Linux systems.

3.5. Installing and Using a Shared Library

Once you've created a shared library, you'll want to install it. The simple approach is simply to copy the library into one of the standard directories (e.g., /usr/lib) and run ldconfig(8).

If you can't do that (e.g., you don't have the right to modify /usr/lib), then you can use environment variables to control things. First, you'll need to create the shared libraries somewhere. Then, you'll need to set up the necesary symbolic links, in particular a link from a soname to the real name (as well as from a versionless soname, that is, a soname that ends in ``.so'' for users who don't specify a version at all). The simplest approach is to run:

```
 ldconfig −n directory_with_shared_libraries
```

Then you can set LD_LIBRARY_PATH, which is a colon−separated list of directories in which to search for shared libraries before the usual places. If you're using bash, you could invoke my_program this way using:

```
LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH  my_program
```

If you want to override just a few selected functions, you can do this by creating an overriding object file and setting LD_PRELOAD; the functions in this object file will override just those functions.

Usually you can update libraries without concern; if there was an API change, the library creator will change the soname. However, if a program breaks on an update to a library that kept the same soname, you can force it to use the older library version by copying the old library back somewhere, renaming the program (say to the old name plus ``.orig''), and then create a small ``wrapper'' script that resets the library to use and calls the real (renamed) program. You could place the old library in its own special area, if you like, though the numbering conventions do permit multiple versions to live in the same directory. The wrapper script could look something like this:

```
  #!/bin/sh
  export LD_LIBRARY_PATH=/usr/local/my_lib:$LD_LIBRARY_PATH
  exec /usr/bin/my_program.orig $*
```

You can see the list of the shared libraries used by a program using ldd(1). So, for example, you can see the shared libraries used by ls by typing:

```
  ldd /bin/ls
```

Generally you'll see a list of the sonames being depended on, along with the directory that those names resolve to. In practically all cases you'll have at least two dependencies:

- /lib/ld−linux.so.N (where N is 1 or more, usually at least 2). This is the library that loads all other libraries.
- libc.so.N (where N is 6 or more). This is the C library. Even other languages tend to use the C library (at least to implement their own libraries), so most programs at least include this one.

Beware: do *not* run ldd on a program you don't trust. As is clearly stated in the ldd(1) manual, ldd works by (in certain cases) by setting a special environment variable (for ELF objects, LD_TRACE_LOADED_OBJECTS) and then executing the program. It may be possible for an untrusted program to force the ldd user to run arbitrary code (instead of simply showing the ldd information). So, for safety's sake, don't use ldd on programs you don't trust to execute.

3.6. Incompatible Libraries

When a new version of a library is binary−incompatible with the old one the soname needs to change. In C, there are four basic reasons that a library would cease to be binary compatible:

1. The behavior of a function changes so that it no longer meets its original specification,
2. Exported data items change (exception: adding optional items to the ends of structures is okay, as long as those structures are only allocated within the library).
3. An exported function is removed.
4. The interface of an exported function changes.

If you can avoid these reasons, you can keep your libraries binary−compatible. Said another way, you can keep your Application Binary Interface (ABI) compatible if you avoid such changes. For example, you might want to add new functions but not delete the old ones. You can add items to structures but only if you can make sure that old programs won't be sensitive to such changes by adding items only to the end of the structure, only allowing the library (and not the application) to allocate the structure, making the extra items optional (or having the library fill them in), and so on. Watch out − you probably can't expand structures if users are using them in arrays.

For C++ (and other languages supporting compiled−in templates and/or compiled dispatched methods), the situation is trickier. All of the above issues apply, plus many more issues. The reason is that some information is implemented ``under the covers'' in the compiled code, resulting in dependencies that may not be obvious if you don't know how C++ is typically implemented. Strictly speaking, they aren't ``new'' issues, it's just that compiled C++ code invokes them in ways that may be surprising to you. The following is a (probably incomplete) list of things that you cannot do in C++ and retain binary compatibility, as reported by Troll Tech's Technical FAQ:

1. add reimplementations of virtual functions (unless it it safe for older binaries to call the original implementation), because the compiler evaluates SuperClass::virtualFunction() calls at compile−time (not link−time).
2. add or remove virtual member functions, because this would change the size and layout of the vtbl of every subclass.
3. change the type of any data members or move any data members that can be accessed via inline member functions.
4. change the class hierarchy, except to add new leaves.
5. add or remove private data members, because this would change the size and layout of every subclass.
6. remove public or protected member functions unless they are inline.
7. make a public or protected member function inline.
8. change what an inline function does, unless the old version continues working.
9. change the access rights (i.e. public, protected or private) of a member function in a portable program, because some compilers mangle the access rights into the function name.

Given this lengthy list, developers of C++ libraries in particular must plan for more than occasional updates that break binary compatibility. Fortunately, on Unix−like systems (including Linux) you can have multiple versions of a library loaded at the same time, so while there is some disk space loss, users can still run ``old'' programs needing old libraries.

4. Dynamically Loaded (DL) Libraries

Dynamically loaded (DL) libraries are libraries that are loaded at times other than during the startup of a program. They're particularly useful for implementing plugins or modules, because they permit waiting to load the plugin until it's needed. For example, the Pluggable Authentication Modules (PAM) system uses DL libraries to permit administrators to configure and reconfigure authentication. They're also useful for implementing interpreters that wish to occasionally compile their code into machine code and use the compiled version for efficiency purposes, all without stopping. This approach can be useful in implementing a just−in−time compiler or multi−user dungeon (MUD).

In Linux, DL libraries aren't actually special from the point−of−view of their format; they are built as standard object files or standard shared libraries as discussed above. The main difference is that the libraries aren't automatically loaded at program link time or start−up; instead, there is an API for opening a library, looking up symbols, handling errors, and closing the library. C users will need to include the header file <dlfcn.h> to use this API.

The interface used by Linux is essentially the same as that used in Solaris, which I'll call the ``dlopen()'' API. However, this same interface is not supported by all platforms; HP−UX uses the different shl_load() mechanism, and Windows platforms use DLLs with a completely different interface. If your goal is wide portability, you probably ought to consider using some wrapping library that hides differences between platforms. One approach is the glib library with its support for Dynamic Loading of Modules; it uses the underlying dynamic loading routines of the platform to implement a portable interface to these functions. You can learn more about glib at http://developer.gnome.org/doc/API/glib/glib−dynamic−loading−of−modules.html. Since the glib interface is well−explained in its documentation, I won't discuss it further here. Another approach is to use libltdl, which is part of GNU libtool. If you want much more functionality this, you might want to look into a CORBA Object Request Broker (ORB). If you're still interested in directly using the interface supported by Linux and Solaris, read on.

4.1. dlopen()

The dlopen(3) function opens a library and prepares it for use. In C its prototype is:

```
void * dlopen(const char *filename, int flag);
```

If filename begins with ``/'' (i.e., it's an absolute path), dlopen() will just try to use it (it won't search for a library). Otherwise, dlopen() will search for the library in the following order:

1. A colon−separated list of directories in the user's LD_LIBRARY path environment variable.
2. The list of libraries specified in /etc/ld.so.cache (which is generated from /etc/ld.so.conf).
3. /lib, followed by /usr/lib. Note the order here; this is the reverse of the order used by the old a.out loader. The old a.out loader, when loading a program, first searched /usr/lib, then /lib (see the man page ld.so(8)). This shouldn't normally matter, since a library should only be in one or the other directory (never both), and different libraries with the same name are a disaster waiting to happen.

In dlopen(), the value of `flag` must be either RTLD_LAZY, meaning ``resolve undefined symbols as code from the dynamic library is executed'', or RTLD_NOW, meaning ``resolve all undefined symbols before dlopen() returns and fail if this cannot be done''. RTLD_GLOBAL may be optionally or'ed with either value in `flag`, meaning that the external symbols defined in the library will be made available to subsequently loaded libraries. While you're debugging, you'll probably want to use RTLD_NOW; using RTLD_LAZY can create inscrutible errors if there are unresolved references. Using RTLD_NOW makes opening the library take slightly longer (but it speeds up lookups later); if this causes a user interface problem you can switch to RTLD_LAZY later.

If the libraries depend on each other (e.g., X depends on Y), then you need to load the dependees first (in this example, load Y first, and then X).

The return value of dlopen() is a ``handle'' that should be considered an opaque value to be used by the other DL library routines. dlopen() will return NULL if the attempt to load does not succeed, and you need to check for this. If the same library is loaded more than once with dlopen(), the same file handle is returned.

If the library exports a routine named _init, then that code is executed before dlopen() returns. You can use this fact in your own libraries to implement initialization routines. See Section 5.2 for more information.

4.2. dlerror()

Errors can be reported by calling dlerror(), which returns a string describing the error from the last call to dlopen(), dlsym(), or dlclose(). One oddity is that after calling dlerror(), future calls to dlerror() will return NULL until another error has been encountered.

4.3. dlsym()

There's no point in loading a DL library if you can't use it. The main routine for using a DL library is dlsym(3), which looks up the value of a symbol in a given (opened) library. This function is defined as:

```
 void * dlsym(void *handle, char *symbol);
```

the handle is the value returned from dlopen, and symbol is a NIL−terminated string. If you can avoid it, don't store the result of dlsym() into a void* pointer, because then you'll have to cast it each time you use it (and you'll give less information to other people trying to maintain the program).

dlsym() will return a NULL result if the symbol wasn't found. If you know that the symbol could never have the value of NULL or zero, that may be fine, but there's a potential ambiguity otherwise: if you got a NULL, does that mean there is no such symbol, or that NULL is the value of the symbol? The standard solution is to call dlerror() first (to clear any error condition that may have existed), then call dlsym() to request a symbol, then call dlerror() again to see if an error occurred. A code snippet would look like this:

```
dlerror(); /* clear error code */
s = (actual_type) dlsym(handle, symbol_being_searched_for);
if ((err = dlerror()) != NULL) {
 /* handle error, the symbol wasn't found */
} else {
 /* symbol found, its value is in s */
}
```

4.4. dlclose()

The converse of dlopen() is dlclose(), which closes a DL library. The dl library maintains link counts for dynamic file handles, so a dynamic library is not actually deallocated until dlclose has been called on it as many times as dlopen has succeeded on it. Thus, it's not a problem for the same program to load the same library multiple times. If a library is deallocated, its function _fini is called (if it exists); see Section 5.2 for more information. Note: dlclose() returns 0 on success, and non−zero on error; some Linux manual pages don't mention this.

4.5. DL Library Example

Here's an example from the man page of dlopen(3). This example loads the math library and prints the cosine of 2.0, and it checks for errors at every step (recommended):

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

int main(int argc, char **argv) {
    void *handle;
    double (*cosine)(double);
    char *error;

    handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
    if (!handle) {
        fputs (dlerror(), stderr);
        exit(1);
    }

    cosine = dlsym(handle, "cos");
    if ((error = dlerror()) != NULL)  {
        fputs(error, stderr);
        exit(1);
    }

    printf ("%f\n", (*cosine)(2.0));
    dlclose(handle);
}
```

If this program were in a file named "foo.c", you would build the program with the following command:

```
gcc −o foo foo.c −ldl
```

5. Miscellaneous

5.1. nm command

The nm(1) command can report the list of symbols in a given library. It works on both static and shared libraries. For a given library nm(1) can list the symbol names defined, each symbol's value, and the symbol's type. It can also identify where the symbol was defined in the source code (by filename and line number), if that information is available in the library (see the −l option).

The symbol type requires a little more explanation. The type is displayed as a letter; lowercase means that the symbol is local, while uppercase means that the symbol is global (external). Typical symbol types include T (a normal definition in the code section), D (initialized data section), B (uninitialized data section), U (undefined; the symbol is used by the library but not defined by the library), and W (weak; if another library also defines this symbol, that definition overrides this one).

If you know the name of a function, but you truly can't remember what library it was defined in, you can use nm's ``−o'' option (which prefixes the filename in each line) along with grep to find the library name. From a Bourne shell, you can search all the libraries in /lib, /usr/lib, direct subdirectories of /usr/lib, and /usr/local/lib for ``cos'' as follows:

```
nm −o /lib/* /usr/lib/* /usr/lib/*/* \
      /usr/local/lib/* 2> /dev/null | grep 'cos$'
```

Much more information about nm can be found in the nm ``info'' documentation locally installed at info:binutils#nm.

5.2. Special functions _init and _fini

Two special functions aid in initializing and finalizing a module: _init and _fini. If a function ``_init'' is exported in a library, then it is called when the library is first opened (via dlopen() or simply as a shared library). In a C program, this just means that you defined some function named _init. There is a corresponding function called _fini, which is called whenever a client finishes using the library (via a call dlclose() that brings its reference count to zero, or on normal exit of the program). The C prototypes for these functions are:

```
  void _init(void);
  void _fini(void);
```

When compiling the file into a ``.o'' file in gcc, be sure to add the gcc option ``−nostartfiles''. This keeps the C compiler from linking the system startup libraries against the .so file. Otherwise, you'll get a ``multiple−definition'' error. My thanks to Jim Mischel and Tim Gentry for their suggestion to add this discussion of _init and _fini, as well as help in creating it.

5.3. Shared Libraries Can Be Scripts

It's worth noting that the GNU loader permits shared libraries to be text files using a specialized scripting language instead of the usual library format. This is useful for indirectly combining other libraries. For example, here's the listing of `/usr/lib/libc.so` on one of my systems:

```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
GROUP ( /lib/libc.so.6 /usr/lib/libc_nonshared.a )
```

For more information about this, see the texinfo documentation on ld linker scripts (ld command language). General information is at info:ld#Options and info:ld#Commands, with likely commands discussed in info:ld#Option Commands.

5.4. GNU libtool

If you're building an application that should port to many systems, you might consider using GNU libtool to build and install libraries. GNU libtool is a generic library support script. Libtool hides the complexity of using shared libraries behind a consistent, portable interface. Libtool provides portable interfaces to create object files, link libraries (static and shared), link executables, debug executables, install libraries, install executables. It also includes libltdl, a portability wrapper for dynamically loading programs. For more information, see its documentation at http://www.gnu.org/software/libtool/manual.html

5.5. Removing symbols for space

All the symbols included in generated files are useful for debugging, but take up space. If you need space, you can eliminate some of it.

The best approach is to first generate the object files normally, and do all your debugging and testing first (debugging and testing is much easier with them). Afterwards, once you've tested the program thoroughly, use strip(1) to remove the symbols. The strip(1) command gives you a good deal of control over what symbols to eliminate; see its documentation for details.

Another approach is to use the GNU ld options ``−S'' and ``−s''; ``−S'' omits debugger symbol information (but not all symbols) from the output file, while ``−s'' omits all symbol information from the output file. You can invoke these options through gcc as ``−Wl,−S'' and ``−Wl,−s''. If you always strip the symbols and these options are sufficient, feel free, but this is a less flexible approach.

5.6. Extremely small executables

You might find the paper Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux useful. It describes how to make a truly tiny program executable. Frankly, you shouldn't use most of these tricks under normal circumstances, but they're quite instructive in showing how ELF really works.

6. More Examples

The following are more examples of all three approaches (static, shared, and dynamically loaded libraries). File libhello.c is a trivial library, with libhello.h as its header. File demo_use.c is a trivial caller of the library. This is followed by commented scripts (script_static and script_dynamic) showing how to use the library as a static and shared library. This is followed by demo_dynamic.c and script_dynamic, which show how to use the shared library as a dynamically loaded library.

6.1. File libhello.c

```
/* libhello.c - demonstrate library use. */

#include <stdio.h>

void hello(void) {
  printf("Hello, library world.\n");
}
```

6.2. File libhello.h

```
/* libhello.h - demonstrate library use. */


void hello(void);
```

6.3. File demo_use.c

```
/* demo_use.c -- demonstrate direct use of the "hello" routine */

#include "libhello.h"

int main(void) {
 hello();
 return 0;
}
```

6.4. File script_static

```
#!/bin/sh
# Static library demo

# Create static library's object file, libhello-static.o.
# I'm using the name libhello-static to clearly
# differentiate the static library from the
# dynamic library examples, but you don't need to use
# "-static" in the names of your
# object files or static libraries.
```

```
gcc -Wall -g -c -o libhello-static.o libhello.c

# Create static library.

ar rcs libhello-static.a libhello-static.o

# At this point we could just copy libhello-static.a
# somewhere else to use it.
# For demo purposes, we'll just keep the library
# in the current directory.

# Compile demo_use program file.

gcc -Wall -g -c demo_use.c -o demo_use.o

# Create demo_use program; -L. causes "." to be searched during
# creation of the program.  Note that this command causes
# the relevant object file in libhello-static.a to be
# incorporated into file demo_use_static.

gcc -g -o demo_use_static demo_use.o -L. -lhello-static

# Execute the program.

./demo_use_static
```

6.5. File script_shared

```
#!/bin/sh
# Shared library demo

# Create shared library's object file, libhello.o.

gcc -fPIC -Wall -g -c libhello.c

# Create shared library.
# Use -lc to link it against C library, since libhello
# depends on the C library.

gcc -g -shared -Wl,-soname,libhello.so.0 \
    -o libhello.so.0.0 libhello.o -lc

# At this point we could just copy libhello.so.0.0 into
# some directory, say /usr/local/lib.

# Now we need to call ldconfig to fix up the symbolic links.

# Set up the soname.  We could just execute:
# ln -sf libhello.so.0.0 libhello.so.0
# but let's let ldconfig figure it out.

/sbin/ldconfig -n .

# Set up the linker name.
# In a more sophisticated setting, we'd need to make
# sure that if there was an existing linker name,
# and if so, check if it should stay or not.

ln -sf libhello.so.0 libhello.so
```

```
# Compile demo_use program file.

gcc -Wall -g -c demo_use.c -o demo_use.o

# Create program demo_use.
# The -L. causes "." to be searched during creation
# of the program; note that this does NOT mean that "."
# will be searched when the program is executed.

gcc -g -o demo_use demo_use.o -L. -lhello

# Execute the program.  Note that we need to tell the program
# where the shared library is, using LD_LIBRARY_PATH.

LD_LIBRARY_PATH="." ./demo_use
```

6.6. File demo_dynamic.c

```
/* demo_dynamic.c -- demonstrate dynamic loading and
   use of the "hello" routine */


/* Need dlfcn.h for the routines to
   dynamically load libraries */
#include <dlfcn.h>

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

/* Note that we don't have to include "libhello.h".
   However, we do need to specify something related;
   we need to specify a type that will hold the value
   we're going to get from dlsym(). */

/* The type "simple_demo_function" describes a function that
   takes no arguments, and returns no value: */

typedef void (*simple_demo_function)(void);


int main(void) {
 const char *error;
 void *module;
 simple_demo_function demo_function;

 /* Load dynamically loaded library */
 module = dlopen("libhello.so", RTLD_LAZY);
 if (!module) {
   fprintf(stderr, "Couldn't open libhello.so: %s\n",
           dlerror());
   exit(1);
 }

 /* Get symbol */
 dlerror();
 demo_function = dlsym(module, "hello");
 if ((error = dlerror())) {
   fprintf(stderr, "Couldn't find hello: %s\n", error);
```

```
  exit(1);
 }

 /* Now call the function in the DL library */
 (*demo_function)();

 /* All done, close things cleanly */
 dlclose(module);
 return 0;
}
```

6.7. File script_dynamic

```
#!/bin/sh
# Dynamically loaded library demo

# Presume that libhello.so and friends have
# been created (see dynamic example).

# Compile demo_dynamic program file into an object file.

gcc -Wall -g -c demo_dynamic.c

# Create program demo_use.
# Note that we don't have to tell it where to search for DL libraries,
# since the only special library this program uses won't be
# loaded until after the program starts up.
# However, we DO need the option -ldl to include the library
# that loads the DL libraries.

gcc -g -o demo_dynamic demo_dynamic.o -ldl

# Execute the program.  Note that we need to tell the
# program where get the dynamically loaded library,
# using LD_LIBRARY_PATH.

LD_LIBRARY_PATH="." ./demo_dynamic
```

7. Other Information Sources

Particularly useful sources of information about libraries include the following:

* ``The GCC HOWTO'' by Daniel Barlow. In particular, this HOWTO discusses compiler options for creating libraries and how to query libraries. It covers information not covered here, and vice versa. This HOWTO is available through the Linux Documentation Project at http://www.linuxdoc.org.
* ``Executable and Linkable Format (ELF)'' by the Tool Interface Standards (TIS) committee (this is actually one chapter of the Portable Formats Specification Version 1.1 by the same committee). This provides information about the ELF format (it isn't specific to Linux or GNU gcc), and provides a great deal of detail on the ELF format. See ftp://tsx−11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz If you get the file from MIT, note that the format is unusual; after gunzipping and untarring, you'll get an ``hps'' file; just strip off the top and bottom lines, rename it to a ``ps'' file, and you'll get a printable Postscript file with the usual filename.
* ``ELF: From the Programmer's Perspective'' by Hongjui Lu. This gives Linux and GNU gcc−specific information on ELF, and is available at ftp://tsx−11.mit.edu/pub/linux/packages/GCC/elf.ps.gz.

---

8. Copyright and License

This document is Copyright (C) 2000 David A. Wheeler. It is covered by the GNU General Public License (GPL). You may redistribute it without cost. Interpret the document's source text as the ``program'' and adhere to the following terms:

> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA

These terms do permit mirroring by other web sites, but please:

- make sure your mirrors automatically get upgrades from the master site,
- clearly show the location of the master site, http://www.dwheeler.com/program−library, with a hypertext link to the master site, and
- give me (David A. Wheeler) credit as the author.

The first two points primarily protect me from repeatedly hearing about obsolete bugs. I do not want to hear about bugs I fixed a year ago, just because you are not properly mirroring the document. By linking to the master site, users can check and see if your mirror is up−to−date. I'm sensitive to the problems of sites which have very strong security requirements and therefore cannot risk normal connections to the Internet; if that describes your situation, at least try to meet the other points and try to occasionally sneakernet updates into your environment.

By this license, you may modify the document, but you can't claim that what you didn't write is yours (i.e., plagiarism) nor can you pretend that a modified version is identical to the original work. Modifying the work does not transfer copyright of the entire work to you; this is not a ``public domain'' work in terms of copyright law. See the license for details, in particular noting that ``You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.'' If you have questions about what the license allows, please contact me. In most cases, it's better if you send your changes to the master integrator (currently David A. Wheeler), so that your changes will be integrated with everyone else's changes into the master copy.