Program Libraries

Program Libraries

What is a program library?

A library is a collection of implementations of behavior, written in terms of a language, that has a well-defined interface by which behavior is invoked

– Wikipedia

A "program library" is simple 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.

- The Linux Documentation Project

Program Libraries

Program libraries can be divided into three types: static, shared and dynamically loaded.

Static libraries are incorporated into a program executable before the program is run.

Shared libraries are loaded at program start-up and may be shared by different programs.

Dynamically loaded libraries are loaded while the executable is running.

Static Libraries

Static libraries are collections of object files. This type of library file usually has a ".a" suffix.

Static libraries let users link to object files without recompiling the code. It also allow a library to be distributed without releasing source code.

Static Libraries

Example:

my_library.hpp

```
#pragma once
namespace my_library {
   extern "C" void my_library_function();
}
```

my_library.cpp

```
#include <iostream>
#include "my_library.hpp"

namespace my_library {
   void my_library_function() {
      std::cout << "My Library Function." << std::endl;
   }
}
```

Static Libraries

Example:

main.cpp

```
#include <iostream>
#include "my_library.hpp"

int main() {
   std::cout << "Main Function:" << std::endl;
   my_library::my_library_function();
}
```

Static Libraries

Example – Compiling with object files.

```
$ g++ -c my_library.cpp
$ g++ -c main.cpp
$ g++ main.o my_library.o –o main
$ ./main
Main Function:
My Library Function.
```

Static Libraries

Example – Compiling as static library.

```
$ g++ -c my_library.cpp
$ ar rcs libmy_library.a my_library.o
$ g++ -c main.cpp
$ g++ main.o libmy_library.a –o main
$ ./main
Main Function:
My Library Function.

$ g++ main.o –l my_library –L ./ -o main
$ ./main
Main Function:
My Library Function.
```

Shared Libraries

Shared Libraries are loaded when a program is loaded. Programs can all share access to a shared library and will be upgraded if a newer version of the library is installed.

Multiple versions of the library can be installed to allow programs with specific requirements use particular versions of the library.

These libraries are usually .so on Linux and .dylib on MAC OS X.

Shared Libraries

To make all of this functionality work, the shared libraries follow a specific naming convention.

Every library has a "**soname**" which starts with the prefix "**lib**" followed by the **name** of the library, the extension "**.so**" then a **period** and a **version number**.

Shared Libraries

Each shared library also has a "real name" which is the name of the file with the actual library code. This "real name" is the "soname" with additional **period** and **minor version number** and optionally a **period** and **release number.**

These version and release numbers allow the exact version of the library to be determined.

Shared Libraries

For a single shared library, the system will often have a number of files linking together.

For example:

soname

`/usr/lib/libreadline.so.3`

linking to a realname like:

`/usr/lib/libreadline.so.3.0`

Should also be:

`/usr/lib/libreadline.so`

linking to:

`/usr/lib/libreadline.so.3`

Shared Libraries

Example – Compiling a shared library

```
$ g++ -fPIC –c my_library.cpp
(-fPIC position independent code, needed for library code)

$ g++ -shared –W1,-soname,libmy_library.so.1 \
   -o libmy_library.so.1.0.1 my_library.o –lc

$ ln –s libmy_library.so.1.0.1 libmy_library.so.1
$ ln –s libmy_library.so.1 libmy_library.so
```

Shared Libraries

Example – Using a shared library

```
$ g++ main.cpp –lmy_library –L ./ -o main
$ ./main
Main Function
My Library Function.
```

Shared Libraries

The problem with this example is that we have a fixed path for the library.

It has been told that the library is going to be in the same directory.

The location of shared libraries may differ depending on the system.

Shared Libraries

`$LD_LIBRARY_PATH` is one way of temporarily setting a search path for libraries.

When a program is launched that requires a shared library, the system will search the directories in `$LD_LIBRARY_PATH` for that library.

However, it is only intended for testing.

Shared Libraries

If you want your library to be installed into the system you can copy the .so files into one of the standard directories - `/usr/lib` and run `ldconfig`

Any program using your library can simply use `-lmy_library` and the system will find it in `/usr/lib`

Dynamically Loaded Libraries

Dynamically Loaded Libraries (not to be confused with Dynamic-Link Libraries) are loaded by the program itself inside the source code.

The libraries themselves are built as standard object or shared libraries, the only difference is that the libraries aren't loaded at compiler linking phase or start-up but at some point determined by the programmer.

Dynamically Loaded Libraries

The function:

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

Will open a library, prepare it for use and return a handle.

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

Searches the library for a specific symbol.

```
void dlclose();
```

Closes the library.

Dynamically Loaded Libraries

```
#include <iostream>
#include <dlfcn.h>

int main() {
   std::cout << "Main Function" << std::cout;
   void *handle = dlopen("libmy_library.so", RTLD_LAZY);
   if(!handle){/*error*/}

   void (*lib_fun)();
   lib_fun = (void (*)())dlsym(handle,"my_library_function");
   if(dlerror() != NULL){/*error*/}

   (*lib_fun)();

   dlclose(handle);
}
```

Libraries in Windows

Static libraries on Windows have the extension `.lib` instead of `.a` and can be created in the same way as Linux/OS X if using MinGW or created through Visual Studio.

Libraries in Windows

Shared libraries (`.dll`) require a few changes. The extra qualifiers:

```
__declspec(dllimport)
__declspec(dllexport)
```

Are used to define the interface of the DLL. These can be used to mark functions, data or objects as imported or exported from a DLL.

DLLs can also be created from Visual Studio.

Summary

Three types of libraries:

- Static
- Shared
- Dynamically Loaded

Correct use of libraries avoids:

- Unnecessary work
- Coupling between projects
- Slow compile times
- Large compiled executables

Sources:

The Linux Documentation Project (tldp.org)