# Overview of Libraries

Shared/Dynamic Libraries

Static Libraries

# Shared/Dynamic Libraries

Shared/dynamic library make it easy to share functions and resources across multiple executable files. Multiple applications can also access the contents of a single copy of a shared/dynamic library in memory at the same time.

* **Uses fewer resources:** Don't get loaded into the RAM together with the main program, so don't occupy space unless required. When a lib is needed, it is loaded and run.
* **Promotes modular architecture:** Help develop large programsthat require multiple language versions or a program that requires modular architecture.
* **Aid easy deployment and installation:** When a function within a shared/dynamic library needs an update or fix, the deployment and installation of the lib does not require the program to be relinked with the lib. Additionally, if multiple programs use the same lib, then all of them get benefited from the update or the fix.

## Create and Use

### Windows

Folder structure:

**$** **create-dll**

Input files

│ build.bat

│ clean.bat

│

├───bin

Output files

│ addition\_lib.dll

│ test\_lib.exe

│

├───build

Temporary files

│ addition\_lib.o

│ test\_lib.o

│

└───src

Input files

addition\_lib.cpp

addition\_lib.h

test\_lib.cpp

**1. Prepare the source code**

In src/addition\_lib.h:

#pragma once

#ifdef \_\_cplusplus

    extern "C" {

#endif

double add(double a, double b);

#ifdef \_\_cplusplus

    }

#endif

In src/addition\_lib.cpp:

#include <stdio.h>

#include "addition\_lib.h"

double add(double a, double b) {

    return a + b;

}

In src/test\_lib.cpp:

#include <iostream>

#include "addition\_lib.h"

void test\_add() {

    double sum = add(1.2, 2.4);

    std::cout << "Sum is " << sum << std::endl;

}

int main() {

    test\_add();

    return 0;

}

**2. Prepare the build script**

In build.bat:

set libfile=addition\_lib

set testlibfile=test\_lib

set src=src

set build=build

set bin=bin

Here we decide to export all functions and variables to the DLL

:: Build dll

g++ -c %src%\%libfile%.cpp

move /Y \*.o %build%

g++ -shared -o %bin%\%libfile%.dll %build%\%libfile%.o -Wl,--export-all-symbols

:: Build exe

g++ -c %src%\%testlibfile%.cpp

move /Y \*.o %build%

g++ -o %bin%\%testlibfile%.exe %build%\%testlibfile%.o -L%bin% -l%libfile%

pause

If you want to rebuild the program, you can also need a clean.bat to clean output files quickly:

set build=build

set bin=bin

del -f %build%\\*.a %build%\\*.o   %bin%\\*.dll %bin%\\*.exe

pause

**3. Build the DLL**

To build the DLL, run:

# In create-dll dir

build.bat

**4. Run the DLL**

To run the DLL, run:

# In create-dll/bin dir

test\_lib.exe

Output:

Sum is 3.6

**NOTES:**

* Running this "g++ -o out-file.exe -L. llib-file.dll out-file.o" will cause error "*cannot file lib-file.dll*". The correct way is: "g++ -o out-file.exe -L. llib-file out-file.o". Briefly, **must remove the .dll extension from the lib-file name**.
* Running this "g++ -o out-file.exe -L. llib-dir\lib-file out-file.o" will cause error "*cannot file lib-dir\lib-file*". The correct way is: "g++ -o out-file.exe -Llib-dir llib-file out-file.o". Briefly, the **input lib only accepts file name, not file path**. That's why -L is used.

**TIP: Choose functions to export**

In the above example, we export all functions and variables of addition\_lib to the DLL. What if we only want to export some functions? We have to change things a bit as below:

In src/addition\_lib.h:

#pragma once

#ifdef \_\_cplusplus

    extern "C" {

#endif

#ifdef BUILD\_LIB

    #define DLL\_LIB \_\_declspec(dllexport)

Here we only export add() to the DLL

#else

    #define DLL\_LIB \_\_declspec(dllimport)

#endif

DLL\_LIB double add(double a, double b);

#ifdef \_\_cplusplus

    }

#endif

In build.bat, modify the following lines:

From:

:: Build dll

g++ -c %src%\%libfile%.cpp

move /Y \*.o %build%

g++ -shared -o %bin%\%libfile%.dll %build%\%libfile%.o -Wl,--export-all-symbols

To:

:: Build dll

g++ -c -DBUILD\_LIB %src%\%libfile%.cpp

move /Y \*.o %build%

g++ -shared -o %bin%\%libfile%.dll %build%\%libfile%.o

### Linux

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjFn6zQQkJ2e8vxCVxhl2yuD>

<https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html>

**Example:**

**1. Prepare the source code**

Similar to above code

**2. Build the SO**

Create a makefile, or simply run this command:

$ g++ -std=c++11 -Wall -g -fPIC src/addition\_lib.cpp -shared -o libmylib.so

**3. Build and run the executable**

1. Specify .so files at build time, so the linker knows where to get function definitions:

$ g++ -std=c++11 -Wall -g src/test\_lib.cpp -Wl,-rpath=. -L. -lmylib

2. Specify .so files at run time, so the executable knows where to find them:

(1) [Optional] Find where the lib is placed if you don't know it.

$ sudo find / -iname \*lib-name\*.so\*

(2) Check for the existence of the dynamic library path environment variable (LD\_LIBRARY\_PATH)

$ echo $LD\_LIBRARY\_PATH

If there is nothing to be displayed, add a default path value (or not if you wish to):

$ LD\_LIBRARY\_PATH=/usr/local/lib

(3) Make a script to run every time (E.g., *run.sh*):

$ LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/dir-path/to/so/files

$ export LD\_LIBRARY\_PATH

$ ./my\_app

**NOTES:**

* The dynamic library must be named as lib\* (e.g., libmylib) when created. And when calling it, must replace the lib with l (e.g., libmylib will become -lmylib)
* You must always specify the runtime library search path to shared libs during linking by using -Wl,-rpath linking flag (e.g., -Wl,-rpath=/usr/local/lib)

## LD\_DEBUG (Linux)

LD\_DEBUG is an environment variable in Linux that allows to **get detailed information of the loading and linking of shared libraries**. This output can help identify problems related to library loading, symbol resolution, and library dependencies.

### Options

LD\_DEBUG can take several values, each corresponding to a specific type of debugging information. Below are the most commonly used options:

1. libs: Shows detailed info about the loading of shared libraries, including their **paths** and any **issues** encountered during loading.
2. bindings: Shows info related to **symbol binding**, including which shared libraries are providing which symbols.
3. files: Outputs the **file names** being loaded and **their paths** to assist in tracing library locations.
4. reloc: Displays **relocation info**, detailing how **addresses** are adjusted during the loading of libraries.
5. symbols: **Lists symbols** that are being resolved during the linking process.
6. all: Enables all **debugging info** available, providing the most comprehensive output. This one can cover all other options.

**Tip**: You can combine several options separated by commas, such as libs,bindings.

### Examples

General syntax: LD\_DEBUG=<option> ./your-shared-lib

|  |  |  |
| --- | --- | --- |
| **Option** | **Example** | **Explanation** |
| libs | $ LD\_DEBUG=libs ./your-shared-lib  12345: /lib/x86\_64-linux-gnu/libc.so.6: loaded  12345: /lib/x86\_64-linux-gnu/libm.so.6: loaded  12345: /usr/local/lib/libfoo.so.1: not found | 12345 is the process ID (PID) of the program being executed.   1. libc.so.6 library was successfully loaded. 2. libm.so.6 library was successfully loaded. 3. libfoo.so.1 library was not found, which may lead to runtime errors if it is required. |
| bindings | $ LD\_DEBUG=bindings ./your-shared-lib  12345: binding file /lib/x86\_64-linux-gnu/libc.so.6  12345: resolving symbol 'printf' in /lib/x86\_64-linux-gnu/libc.so.6  12345: symbol 'printf' found in /lib/x86\_64-linux-gnu/libc.so.6  12345: resolving symbol 'malloc' in /lib/x86\_64-linux-gnu/libc.so.6  12345: symbol 'malloc' not found in any loaded library | 1. The linker is binding symbols from libc.so.6 library. 2. The linker is trying to find the printf symbol. 3. Confirms successful resolution of the printf symbol. 4. The linker is trying to find the malloc symbol. 5. The malloc symbol could not be resolved, which may indicate a missing library or incorrect linkage. |
| files | $ LD\_DEBUG=files ./your-shared-lib  12345: trying file '/lib/x86\_64-linux-gnu/libc.so.6'  12345: trying file '/usr/lib/libc.so.6'  12345: trying file '/lib/x86\_64-linux-gnu/libm.so.6'  12345: trying file '/usr/lib/libm.so.6' | 1. The linker is searching for libc.so.6 library in the specified path. 2. Each subsequent line indicates additional search attempts in other directories. This helps you understand how the linker is resolving library paths and whether it is searching in the correct locations. |
| reloc | $ LD\_DEBUG=reloc ./your-shared-lib  12345: relocating symbol 'printf' to address 0x7ffff7a33400  12345: relocating symbol 'malloc' to address 0x7ffff7a33000 | 1. The linker is adjusting the address of the printf symbol to the specified memory location. 2. Each relocation message shows how the linker adjusts addresses for the symbols as they are loaded into memory. |
| symbols | $ LD\_DEBUG=symbols ./your-shared-lib  12345: resolving symbol 'printf' in /lib/x86\_64-linux-gnu/libc.so.6  12345: resolving symbol 'malloc' in /lib/x86\_64-linux-gnu/libc.so.6 | 1. The linker is trying to find the printf symbol in the specified library. 2. This output is focused on symbol resolution, showing which symbols are being looked up and whether they are found. |
| all | $ LD\_DEBUG=all ./your-shared-lib  12345: loading /lib/x86\_64-linux-gnu/libc.so.6  12345: found libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6  12345: resolving symbol 'printf' in /lib/x86\_64-linux-gnu/libc.so.6  12345: symbol 'printf' found in /lib/x86\_64-linux-gnu/libc.so.6  12345: relocating symbol 'printf' to address 0x7ffff7a33400 |  |

# Static Libraries

## Windows

## Linux

<https://medium.com/@meghamohan/all-about-static-libraries-in-c-cea57990c495>