Programming in Modern C++

Tutorial T04: How to build a C/C++ program?: Part 4: Static and Dynamic Library

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Tutorial Recap

- Understood the build process and pipeline for C/C++ projects
- Learnt `make` for build automation

Tutorial Objective

- To understand the role of libraries in C/C++ projects
- To learn about Static and Shared Libraries - how to build and use them

Tutorial Outline

1. Tutorial Recap
2. What is a Library?
   - Static vs Shared
3. Our Library Project
4. Static Library Project
   - Build Steps
   - Execution Trace
5. Shared / Dynamic Library Project
   - Build Steps
   - Execution Trace
   - Set Library Path
   - Late Binding and Dynamic Loading
6. Dynamic Link Library (DLL) Project: Windows
7. Tutorial Summary

What is a Library?

What is a Library?

Source: All Accessed 23-Sep-21
Static library, Wikipedia
Dynamic-link library, Wikipedia
Library (computing), Wikipedia
C standard library, Wikipedia
C++ Standard Library, Wikipedia

What is a Library?

- A library is a package of code that is meant to be reused by many programs. Typically, a C / C++ library comes in two pieces:
  - A *header file* that defines the functionality the library is exposing (offering) to the programs using it. For example, `stdio.h`, `math.h`, etc. in C and `iostream`, `vector`, etc. in C++
  - A *pre-compiled binary* that contains the implementation of that functionality pre-compiled into machine language. For example, `glibc` is the GNU C (Standard) Library on Unix

  Some libraries may be split into multiple files and/or have multiple header files
- Libraries are pre-compiled because
  - As libraries rarely change, they do not need to be recompiled often. They can just be reused in binary
  - As pre-compiled objects are in machine language, it prevents people from accessing or changing the source code protecting IP

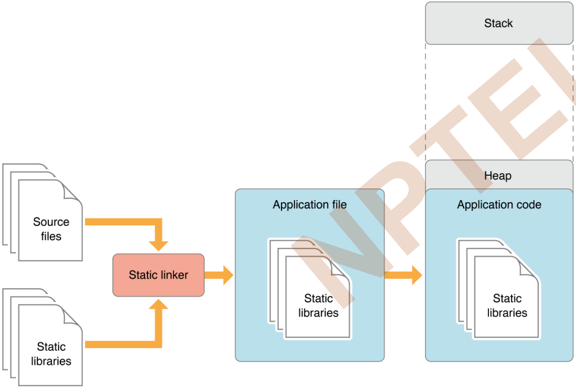
Source: A.1 — Static and dynamic libraries (Accessed 23-Sep-21)

Types of Library

- **Static Library**
  - Consists of routines that are compiled and linked into the program. A program compiled with a static library would have the functionality of the library as a part of the executable
  - Extensions:
    - ▷ *Unix*: `.a` (archive)
    - ▷ *Windows*: `.lib`
- **Dynamic / Shared Library**
  - Consists of routines that are loaded into the application at run time
  - Extensions:
    - ▷ *Unix*: `.so` (shared object)
    - ▷ *Windows*: `.dll` (dynamic link library)
- **Import Library**
  - An import library automates the process of loading and using a dynamic library
  - Extensions:
    - ▷ *Unix*: Shared object (`.so`) file doubles as both a dynamic and an import library
    - ▷ *Windows*: A small static library (`.lib`) of the same name as the dynamic library (`.dll`). The static library is linked into the program at compile time, and then the functionality of the dynamic library can effectively be used as if it were a static library

Static Library

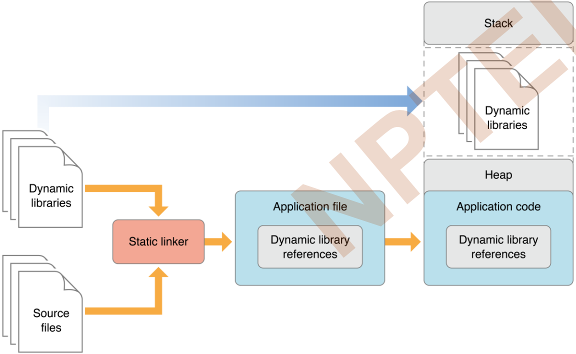
Static Library: Library Code is internal to Application



- Application *needs to recompile* - *difficult version management*
  - If library implementation changes - regular with version upgrade / bug fixes
  - And naturally, if library interface changes - infrequent
- *Large footprint* - especially *bad for mobile apps*
- *Multiple copies* of the same library may be loaded as part of different applications - *bad for mobile apps*
- *Fast in speed* as the library is already loaded and linked

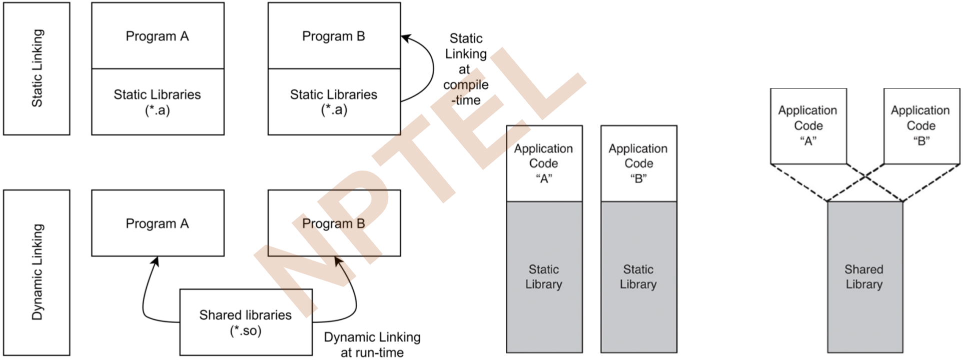
Shared / Dynamic Library

Shared / Dynamic Library: Only Library reference is internal to Application - Library Code is external



- Application *does not need to recompile* - *easy version management*
  - If library implementation changes - regular with version upgrade / bug fixes
  - However, it will need to recompile (like the static library), if library interface changes - infrequent
- *Small footprint* - especially *good for mobile apps*
- *Single copy* of the library will be loaded for different applications - *good for mobile apps*
- The functions in the library needs to be *re-entrant*. Care is needed with static variables
- *Slow in speed* as the library may need to be loaded and linked at run-time

Static vs Shared Library



Static vs Shared Library

Property	Static Library	Shared Library
Compilation	Recompilation is required for changes in external files	No need to recompile the executable
Linking time	Happens as the last step of the compilation process	Are added during linking when executable file and libraries are added to the memory
Import / Mechanism	Are resolved in a caller at compile-time and copied into a target application by the linker	Get imported at the time of execution of target program by the OS
Size	Are bigger in size, because external programs are built in the executable file	Are smaller, because there is only one copy of shared library that is kept in memory
External file changes	Executable file will have to be recompiled if any changes were applied to external files	No need to recompile the executable - only the shared library is replaced
Time / Performance	Takes longer to execute, as loading into memory happens every time while executing	Faster because shared library is already in the memory
Compatibility	Never has compatibility issue, since all code is in one executable module	Programs are dependent on having a compatible library

Source: Difference between Static and Shared libraries and Difference between Static and Shared libraries (Accessed 23-Sep-21)

Our Library Project

Sources: All Accessed 26-Sep-21
Building And Using Static And Shared "C" Libraries
Shared libraries with GCC on Linux
MinGW Static and Dynamic Libraries
Static and Dynamic Libraries in C Language

Our Library Project

- We present a tiny project to illustrate the ideas of static and shared / dynamic libraries
- We use the same set of header and source files to create and use
  - Static Library
  - Shared / Dynamic Library

  and compare them
- First the projects are created with `gcc`. These can work on Unix as well as Windows (with `minGW`: MinGW - Minimalist GNU for Windows)
- Then we show Microsoft-specific process on Windows

Project Files

```
/* lib_myMath.h: Header for my mathematical functions */ /* CPP guards omitted for brevity */
int myMax(int, int);
int myMin(int, int);

/* lib_myMath.c: Implementation for my mathematical functions */
#include "lib_myMath.h"
int myMax(int a, int b) { return a>b? a: b; }
int myMin(int a, int b) { return a<b? a: b; }

/* lib_myPrint.h: Header for my printing function */ /* CPP guards omitted for brevity */
void myPrint(const char*, int);

/* lib_myPrint.c: Implementation for my printing function */
#include <stdio.h>
#include "lib_myPrint.h"
void myPrint(const char *name, int a) { printf("%s: %d\n", name, a); }

/* myApp.c: My application */
#include <stdio.h>
#include "lib_myMath.h"
#include "lib_myPrint.h"
int main() {
    myPrint("Max(3, 5)", myMax(3, 5));
    myPrint("Min(3, 5)", myMin(3, 5));
}
```

Project: Folders and Code Organization

```
Home // Library demonstration project with library as well as application Home = Static or Shared
  |
  ---- app // Application files - will use library headers from ../inc and library from ../lib
  |     ---- myApp.c       // Application Source file
  |     ---- myApp.exe     // Application Executable
  |     ---- myApp.o       // Application Object file
  |     ---- myMakeApp.txt // Application Makefile
  |
  ---- inc // Headers to be included in application and library build
  |     ---- lib_myMath.h
  |     ---- lib_myPrint.h
  |
  ---- lib // Library files
      |
      ---- obj // Library object files
      |     ---- lib_myMath.o
      |     ---- lib_myPrint.o
      |
      ---- src // Library source files - will use library headers from ../../inc
      |     ---- lib_myMath.c
      |     ---- lib_myPrint.c
      |
      ---- lib_mylib.a       // Static Library binary file linked by the application
      ---- lib_mylib.so      // Shared Library binary file linked by the application
      ---- myMakeLibrary.txt // Library Makefile
```

Static Library Project

Static Library Project

Sources: All Accessed 26-Sep-21
A.1 — Static and dynamic libraries
A.2 — Using libraries with Visual Studio
A.3 — Using libraries with Code::Blocks

Static Library Project: Build Steps

- We can build this project by

```
$ gcc lib_myMath.c lib_myPrint.c myApp.c -o myApp
```

- Every time myApp.c is updated, we build lib_myMath.c and lib_myPrint.c even if there is no change. We can avoid the recompile by retaining the object files as:

```
$ gcc -c lib_myMath.c lib_myPrint.c
$ gcc lib_myMath.o lib_myPrint.o myApp.c -o myApp
```

- When we have many such files that rarely change, we would have a lot of such .o files to maintain. These can be bundled into an archive lib_mylib.a for ease of reference

```
$ ar rcs lib_mylib.a lib_myMath.o lib_myPrint.o
```

  - GNU ar utility creates, modifies, and extracts from *archives* (like ZIP) - holding a collection of multiple files in a structure that makes it possible to retrieve the individual files (called *members*)
  - Option rcs asks to create (c) an archive with replacement (r) of members and indexing (s)
  - For details check: ar(1) — Linux manual page

- Finally we use the .a file in place of the .o's to link to myApp.o

```
$ gcc -o myApp myApp.c lib_myLib.a -L.
```

  Alternately, we can place lib_myLib.a in default library path and link by -l_mylib

```
$ gcc -o myApp myApp.c -l_mylib -L.
```

Static Library Project: Makefiles

Static Library makefile

```
# Variables
CC=gcc
AR=ar
SDIR=src
IDIR=../inc
ODIR=obj
CFLAGS=-I$(IDIR)
LFLAGS=-L.
AFLAGS=rcs
# Macros
_DEPS = lib_myMath.h lib_myPrint.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))
_SRC = lib_myMath.c lib_myPrint.c
SRC = $(patsubst %,$(SDIR)/%,$(_SRC))
_OBJ = lib_myMath.o lib_myPrint.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))
# Rules
$(ODIR)/%.o: $(SDIR)/%.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS) -I.
%.o: $(SDIR)/%.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)
lib_mylib.a: $(OBJ)
    $(AR) $(AFLAGS) -o $@ $^
```

Application makefile

```
# Variables
CC=gcc
IDIR=inc
LDIR=lib
CFLAGS=-I../$(IDIR)
LFLAGS=-L../$(LDIR)
DEPS=









# Rules
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)
myApp: myApp.o
    $(CC) -o myApp myApp.o -l_mylib $(LFLAGS)
```

Static Library Project: Execution Trace

- Let us build and execute the project

```
// Build Library
D:\Library\Static\lib $ make -f myMakeLibrary.txt
gcc -c -o obj/lib_myMath.o src/lib_myMath.c -I../inc -I.
gcc -c -o obj/lib_myPrint.o src/lib_myPrint.c -I../inc -I.
ar rcs -o lib_mylib.a obj/lib_myMath.o obj/lib_myPrint.o

// Build Application
D:\Library\Static\app $ make -f myMakeApp.txt
gcc -c -o myApp.o myApp.c -I../inc
gcc -o myApp myApp.o -l_mylib -L../lib

// Run Application
D:\Library\Static\app $ myApp.exe
Max(3, 5): 5
Min(3, 5): 3
```

- So the static library is working as expected
- Next we check on the contents of various folders

Static Library Project: Directory Listing

```
Directory of D:\Library\Static
26-09-2021  14:58    <DIR>          app
26-09-2021  14:58    <DIR>          inc
26-09-2021  14:58    <DIR>          lib
Directory of D:\Library\Static\app
24-09-2021  15:53               179 myApp.c
26-09-2021  11:35            42,348 myApp.exe
26-09-2021  11:35               954 myApp.o
25-09-2021  13:23               215 myMakeApp.txt
Directory of D:\Library\Static\inc
24-09-2021  13:17                66 lib_myMath.h
24-09-2021  13:17                54 lib_myPrint.h
Directory of D:\Library\Static\lib
26-09-2021  11:33             1,722 lib_mylib.a
25-09-2021  13:22               524 myMakeLibrary.txt
26-09-2021  14:58    <DIR>          obj
26-09-2021  14:58    <DIR>          src
Directory of D:\Library\Static\lib\obj
26-09-2021  11:33               716 lib_myMath.o
26-09-2021  11:33               778 lib_myPrint.o
Directory of D:\Library\Static\lib\src
24-09-2021  13:16               143 lib_myMath.c
24-09-2021  13:18               144 lib_myPrint.c
```

Shared / Dynamic Library Project

Sources: All Accessed 26-Sep-21
A.1 — Static and dynamic libraries
A.2 — Using libraries with Visual Studio
A.3 — Using libraries with Code::Blocks

Shared Library Project: Build Steps

- As in the static case, first we compile `lib_myMath.c` and `lib_myPrint.c` to create the object (`.o`) files using the option `-fPIC`:

  ```
  $ gcc -fPIC -c lib_myMath.c lib_myPrint.c
  ```

  - `-fPIC` stands to mean: Compile for *Position Independent Code* (PIC)
    - ▷ For a shared library, the binary of the library and the application are separate and will be separately loaded at run time
    - ▷ So when the object files are generated, we need that all jump calls and subroutine calls to use relative addresses, and not absolute addresses
    - ▷ `-fPIC` flag tells `gcc` to generate this type of code

Shared Library Project: Build Steps

- Next step of building the library gets different now as we do not use ar. Rather we use gcc with the -shared option

```
$ gcc -shared -o lib_mylib.so lib_myMath.o lib_myPrint.o
```

- This creates a shared library lib_mylib.so where the extension .so stands for a *shared object*
- Finally we use the .so file to link to myApp.o

```
$ gcc -o myApp myApp.c lib_myLib.so -L.
```

- Alternately, we can place lib_myLib.so in default library path and link by -l_mylib

```
$ gcc -o myApp myApp.c -l_mylib
```

Shared Library Project: Makefiles

Shared Library makefile

```
# Variables
CC=gcc
SDIR=src
IDIR=../inc
ODIR=obj
CFLAGS=-I$(IDIR)
LFLAGS=-L.

# Macros
_DEPS = lib_myMath.h lib_myPrint.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))
_SRC = lib_myMath.c lib_myPrint.c
SRC = $(patsubst %,$(SDIR)/%,$(_SRC))
_OBJ = lib_myMath.o lib_myPrint.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))

# Rules
$(ODIR)/%.o: $(SDIR)/%.c $(DEPS)
    $(CC) -fPIC -c -o $@ $< $(CFLAGS) -I.
lib_mylib.so: $(OBJ)
    $(CC) -shared -o $@ $^
```

Application makefile

```
# Variables
CC=gcc
IDIR=inc
LDIR=lib
CFLAGS=-I../$(IDIR)
LFLAGS=-L../$(LDIR)
DEPS=

# Rules
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)
myApp: myApp.o
    $(CC) -o myApp myApp.o ../$(LDIR)/lib_mylib.so
```

Shared Library Project: Execution Trace

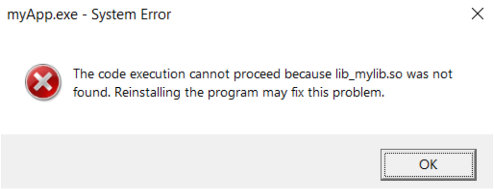
- Let us build and execute the project

```
// Build Library
D:\Library\Shared\lib $ make -f myMakeLibrary.txt
gcc -fPIC -c -o obj/lib_myMath.o src/lib_myMath.c -I../inc -I.
gcc -fPIC -c -o obj/lib_myPrint.o src/lib_myPrint.c -I../inc -I.
gcc -shared -o lib_mylib.so obj/lib_myMath.o obj/lib_myPrint.o

// Build Application
D:\Library\Shared\app $ make -f myMakeApp.txt
gcc -c -o myApp.o myApp.c -I../inc
gcc -o myApp myApp.o ../lib/lib_mylib.so

// Run Application
D:\Library\Shared\app $ myApp.exe
```

Oops! The shared library is not found! The system does not know that lib_mylib.so is in D:\Library\Shared\lib



myApp.exe - System Error

The code execution cannot proceed because lib_mylib.so was not found. Reinstalling the program may fix this problem.

Shared Library Project: Execution Trace

- If we copy `lib_mylib.so` to the application folder `D:\Library\Shared\app` (where `myApp.exe` resides), the problem goes away and the application runs successfully

```
// Run Application
D:\Library\Shared\app $ myApp.exe
Max(3, 5): 5
Min(3, 5): 3
```

- So the shared library is working as expected
  - However, copying the shared library to the application folder is not preferred as we would need to copy `lib_mylib.so` to every application that would use it.
  - We shall discuss a solution to this library path problem in the next section
- Next we check on the contents of various folders and compare the size of the libraries and applications in static and shared cases

Static Library		**Shared Library**		**Remarks**
`lib_mylib.a`	1,722	`lib_mylib.so`	27,749	File `.so` is larger than `.a` due to the overhead of exported references. With large number of functions in the library (as opposed to just 3) the relative overhead will go down
`myApp.exe`	42,348	`myApp.exe`	41,849	Shared `.exe` would be relatively much smaller with more functions in the library

Shared Library Project: Directory Listing

```
Directory of D:\Library\Shared
26-09-2021  14:58    <DIR>          app
26-09-2021  14:58    <DIR>          inc
26-09-2021  14:58    <DIR>          lib
Directory of D:\Library\Shared\app
24-09-2021  15:53               179 myApp.c
26-09-2021  11:45            41,849 myApp.exe
26-09-2021  11:45               954 myApp.o
26-09-2021  09:41               220 myMakeApp.txt
Directory of D:\Library\Shared\inc
24-09-2021  13:17                66 lib_myMath.h
24-09-2021  13:17                54 lib_myPrint.h
Directory of D:\Library\Shared\lib
26-09-2021  11:44            27,749 lib_mylib.so
26-09-2021  10:38               457 myMakeLibrary.txt
26-09-2021  14:58    <DIR>          obj
26-09-2021  14:58    <DIR>          src
Directory of D:\Library\Shared\lib\obj
26-09-2021  11:44               716 lib_myMath.o
26-09-2021  11:44               778 lib_myPrint.o
Directory of D:\Library\Shared\lib\src
24-09-2021  13:16               143 lib_myMath.c
24-09-2021  13:18               144 lib_myPrint.c
```

Shared Library Project: Set Library Path

- We managed to make the application work by copying the shared library to the application folder. However, this is not preferred as we would need to copy `lib_mylib.so` to every application that would use it
- So we need to tell the location for a shared library to the system. This can be done in one of three ways:
  - Place the shared library file in one of the *default library folders* like /usr/local/lib, /usr/local/lib64, /usr/lib and /usr/lib64 (on Unix) or like `C:\Windows\system32` and `C:\Windows` (on Windows). Refer to the system manual details on these folders
  - Add the folder of the library to the library search folders by *setting path / environment variables*:
    - ▷ Windows: Use `PATH`
    - ▷ Unix: Use `LD_LIBRARY_PATH`
  - We may also *Dynamically Load / Unload* a library at the run-time. This is known as *late binding*. This is achieved by using `dlopen()`, `dlsym()`, and `dlclose()` from `dlfcn.h` header

Shared Library Project: Set Library Path: Windows

- **Windows**: Set environment variables PATH to include the folder of the shared library

```
// Check PATH
D:\Library\Shared\app $ PATH
PATH=C:\Windows\system32;...

// Set PATH
D:\Library\Shared\app $ SET PATH=%PATH%;D:\Library\Shared\lib

// Check PATH
D:\Library\Shared\app $ PATH
PATH=C:\Windows\system32;...;D:\Library\Shared\app

// Run Application
D:\Library\Shared\app $ myApp.exe
Max(3, 5): 5
Min(3, 5): 3
```

  - PATH would be reset when we end the command session. Need to set it every time
  - We may set the folder in PATH Environment Variable - Use (preferred) or System to retain it across sessions. Refer: How to set the path and environment variables in Windows

Shared Library Project: Set Library Path: Unix

- **Unix**: Set environment variable LD_LIBRARY_PATH to include the directory of the shared library as follows:
  - For tcsh or csh:

    ```
    $ setenv LD_LIBRARY_PATH /full/path/to/library/directory:${LD_LIBRARY_PATH}
    ```
  - For sh, bash and similar:

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

  Note:
  - LD_LIBRARY_PATH is a list of directories in which to search for ELF libraries at execution time
  - The items in the list are separated by either colons or semicolons, and there is no support for escaping either separator
  - A zero-length directory name indicates the current working directory

Shared Library Project: Dynamic Loading

- We can link shared libraries to a process anytime during its life without automatically loading the shared library by the dynamic loader
- We can do this by using the ’dl’ library – 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
- This is useful when there may be multiple shared libraries for the same (similar) purpose and we get the know which one to link only at the run-time
- The steps invoved are:
    - Load the library from its path using `dlopen()` and get its *handle*
    - Use the *handle* to get access (*function pointers*) to the specific functions we intend to use by `dlsym()`
    - Use the *function pointers* to call the functions in the shared library
    - Finally, unload the library with `dlclose()`
- We first present a simple schematic example

Shared Library Project: Late Binding

- Known as *Late Binding* as the actual functions to be called are decided only at the run-time

Application

```
#include <dlfcn.h>

int main() {
    void* handle =
    dlopen("hello.so", RTLD_LAZY);
    // RTLD_LAZY: Relocations shall be performed at an
    // implementation-defined time, ranging from the
    // time of the dlopen() call until the first
    // reference to a given symbol occurs

    typedef void (*hello_t)();
    hello_t myHello = 0;

    myHello = (hello_t) dlsym(handle, "hello");

    myHello();

    dlclose(handle);
}
```

Library

```
#include <iostream>
using namespace std;

extern "C" void hello() {
   cout << "hello" << endl;
}
```

- Now we present our project in the context of late binding

Shared Library Project: Dynamic Loading

```
/* myDynamicApp.c */
#include <stdio.h>
#include <stdlib.h>
#include "lib_myMath.h"
#include "lib_myPrint.h"
#include <dlfcn.h> /* defines dlopen(), dlsym(), and dlclose() etc. */
int main() { void* lib_handle; /* handle of the opened library */
    const char* error_msg; /* Pointer to an error string */
    /* Load the shared library */
    lib_handle = dlopen("D:\Library\Shared\lib\lib_mylib.so", RTLD_LAZY); /* Library path - dynamic */
    if (!lib_handle) { error_msg = dlerror(); goto Error; }
    /* Define function pointers */
    int (*myMax)(int, int); int (*myMin)(int, int); void (*myPrint)(const char*, int);
    /* Locate the functions in the library. Pick by name and assign to the function pointer */
    myMax = dlsym(lib_handle, "myMax"); if (error_msg = dlerror()) goto Error;
    myMin = dlsym(lib_handle, "myMin"); if (error_msg = dlerror()) goto Error;
    myPrint = dlsym(lib_handle, "myPrint"); if (error_msg = dlerror()) goto Error;
    /* Call the functions */
    (*myPrint)("Max(3, 5)", (*myMax)(3, 5));
    (*myPrint)("Min(3, 5)", (*myMin)(3, 5));
    /* Unload the shared library */
    dlclose(lib_handle);
    return 0; /* Success */
Error: fprintf(stderr, "%s\n", error_msg); exit(1); /* Exit on error */
}
```

Dynamic Link Library (DLL) Project: Windows

Dynamic Link Library (DLL) Project: Windows

Sources: All Accessed 27-Sep-21
Microsoft Visual C++ Static and Dynamic Libraries
Walkthrough: Create and use a static library
Walkthrough: Create and use your own Dynamic Link Library (C++)
How to link DLLs to C++ Projects

Microsoft Visual C++ Static and Dynamic Libraries

- While static and shared library of Unix (specifically **GNU**) is available on Windows through `minGW`, Windows provides Microsoft specific support through MSVC
  - *Static Library*: To work with a static library:
    - ▷ We need to create a static library and an application project (that refers to the library project) using the MSVS IDE
      - — Check: Walkthrough: Create and use a static library
    - ▷ No change in the library or application codes is needed
  - *Dynamic Link Library (DLL)*: To work with a DLL:
    - ▷ We need to create a DLL and an application project (that refers to the DLL project) using the MSVS IDE
      - — Check: Walkthrough: Create and use your own Dynamic Link Library (C++)
    - ▷ We need to change the library and / or application codes with dllexport and dllimport - the Microsoft-specific storage-class attributes for C and C++
      - — Check: dllexport, dllimport
    - ▷ These can be specified by __declspec() to export and import functions, data, and objects to or from a DLL
    - ▷ We modify the library project file to illustrate

Library Project Files: Using dllexport

```
/* lib_myMath.h: Header for my mathematical functions */ /* File changed for export */
__declspec(dllexport) int myMax(int, int); // storage-class attribute __declspec(dllexport) used
__declspec(dllexport) int myMin(int, int); // to export function from DLL. This is MS-specific

/* lib_myMath.c: Implementation for my mathematical functions */ /* No change */
#include "lib_myMath.h"
int myMax(int a, int b) { return a>b? a: b; }
int myMin(int a, int b) { return a<b? a: b; }

/* lib_myPrint.h: Header for my printing function */ /* File changed for export */
__declspec(dllexport) void myPrint(const char*, int);

/* lib_myPrint.c: Implementation for my printing function */ /* No change */
#include <stdio.h>
#include "lib_myPrint.h"
void myPrint(const char *name, int a) { printf("%s: %d\n", name, a); }

/* myApp.c: My application */ /* No change */
#include <stdio.h>
#include "lib_myMath.h"
#include "lib_myPrint.h"
int main() {
    myPrint("Max(3, 5)", myMax(3, 5));
    myPrint("Min(3, 5)", myMin(3, 5));
}
```

Library Project Files: Using dllimport

- In the context of the DLL headers,

```
/* lib_myMath.h */
__declspec(dllexport) int myMax(int, int);
__declspec(dllexport) int myMin(int, int);

/* lib_myPrint.h */
__declspec(dllexport) void myPrint(const char*, int);
```

the application project may include the DLL headers or can just directly import the symbols by dllimport

Including DLL headers

```
/* myApp.c: My application */
#include <stdio.h>
#include "lib_myMath.h"

#include "lib_myPrint.h"



int main() {
    myPrint("Max(3, 5)", myMax(3, 5));
    myPrint("Min(3, 5)", myMin(3, 5));
}
```

Using dllexport

```
/* myApp.c: My application */
#include <stdio.h>
#include "lib_myMath.h"

// storage-class attribute __declspec(dllimport) used
// to import function from DLL. This is MS-specific
__declspec(dllimport) void myPrint(const char*, int);

int main() {
    myPrint("Max(3, 5)", myMax(3, 5));
    myPrint("Min(3, 5)", myMin(3, 5));
}
```

Tutorial Summary

- Understood the role of libraries in C/C++ projects in reuse
- Learn about Static and Shared Libraries in Unix and Windows - how to build and use them
- Learnt about DLLs