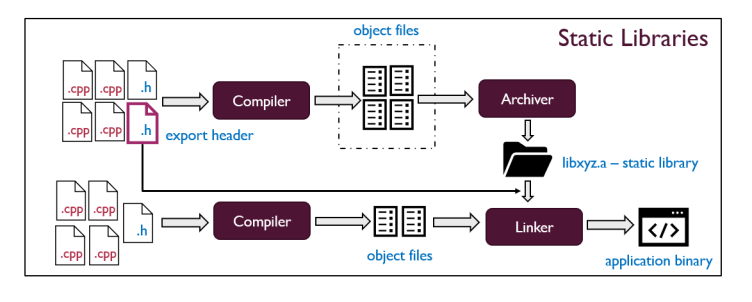
# Static Libraries in C/C++

## ****What are static libraries?****

A static library is collection of pre-compiled object files (in the form of a library archive) and export headers, which allows code reuse without exposing the implementation.



### ****A static library is the logical choice in following situations:****

* when software developers are looking for the simplest possible way to let other developers use their work, while still hiding the implementation details
* the intended functionality to be used from the library is not a large chunk of code
* C/C++ application developers want to prioritize simplicity and portability

### ****It is not the best fit when:****

* the application uses a lot of common or shared resources (which themselves are mostly managed with shared lib objects)
* the application binary uses a number of dynamic libraries - care must be taken to ensure the right flags are used while compiling the static library (e.g. -fPIC)

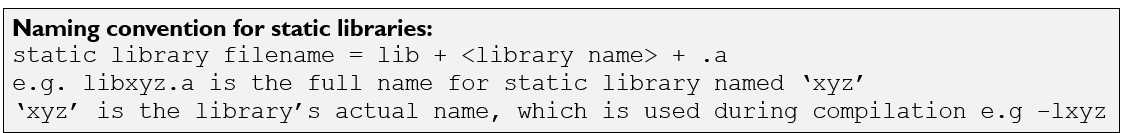
A static library is **'statically linked'** - meaning the compiler resolves all symbols, to create a fully linked and relocated application binary. Such an application binary has no external dependencies and can be loaded into memory and run.

However, this also means that if the library implementation changes - it requires rebuilding the application binary and linking it against the new version of the static library. Unlike shared libraries, static libraries cannot be updated on the fly. But using a static library implies that only the required sections make it to the final application binary, and no more.

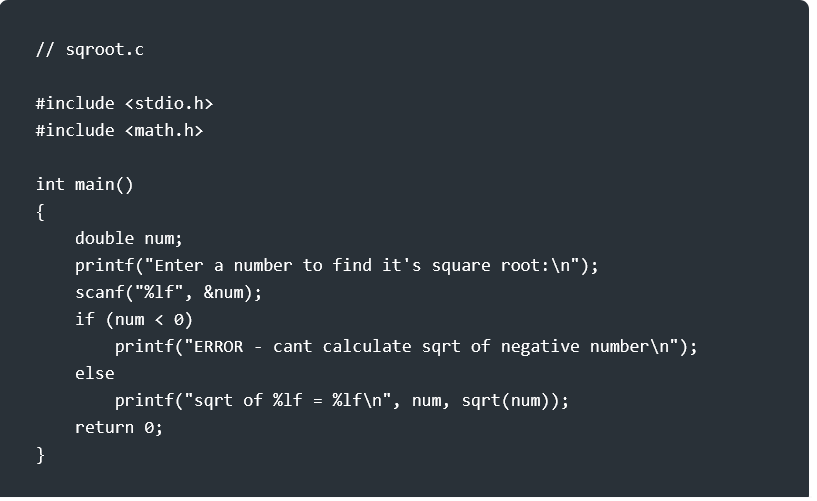
## ****Compiling by linking against a static library****

In general the compiler driver to use a static library looks something of this form:

**$ gcc -Wall <source files> -L <path to the lib files> -l<names of lib files>**



Consider this simple C program, which accepts a number and prints it's square root:

****

Let us try compiling this program the usual way:

$ cc -Wall sqroot.c -o sqroot

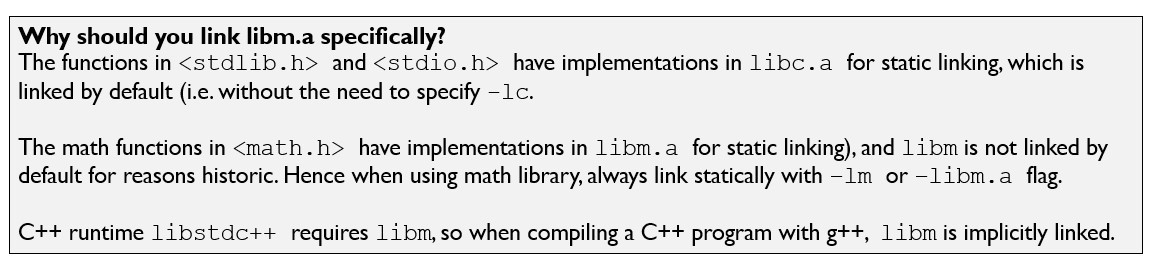
What happened? GCC complains about an undefined reference to 'sqrt' function:

usr/bin/ld: /tmp/ccNj3wiM.o: in function `main':

sqrt.c:(.text+0x66): undefined reference to `sqrt'

collect2: error: ld returned 1 exit status

The answer to this is found below:

****

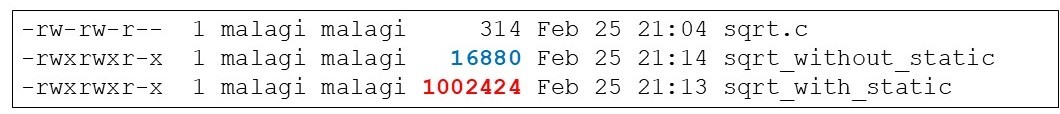
To fix this problem we link against 'libm.a'

$ cc -Wall sqroot.c -o sqroot -lm

However the executable itself is not 'fully static yet'. In order to achieve this we use the -static flag, which instructs the linker to create a fully linked executable:

$ cc -Wall -static sqroot.c -o sqroot -lm

Since the required sections from a static library object file need to be copied in entirety, this bloats the size of the application binary when generating a binary with the -static flag.The difference in size between an executable generated with and without -static flag can be seen below:

****

## ****Resolving issues with use of static libraries****

**“Linker Dependency Rule: if A references a symbol in B, always list A before B’**

The linker searches the object files and the library archives from left to right, in the same order as they appear on the command line. If a library that defines a symbol appears on the command line before the object file that references the symbol, then the reference will not be resolved, and the linker will fail. For example the following fails with a linker error:

$ cc -Wall -static -lm sqroot.c -o sqroot

**Repeat libraries on command line if necessary to satisfy circular dependencies**

Dependencies of static libraries work the same - the library that needs symbols must be first, then the library that resolves the symbol. When multiple static libraries are being used, there is a possibility of introducing a situation known as "circular dependency". That is, a function from libx.a references a symbol in liby.a; and liby.a references a symbol in libx.a. Such situations can be addressed by repeating the name of a static library on the command line if necessary.

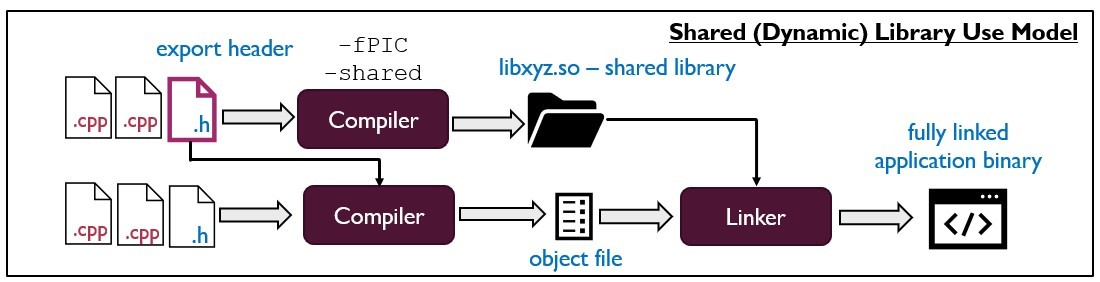
$ cc -Wall -static main.c -o main -lx -ly -lx

Reference: <https://www.linkedin.com/pulse/static-libraries-cc-santosh-s-malagi/>

**Shared (Dynamic) Libraries in C/C++**

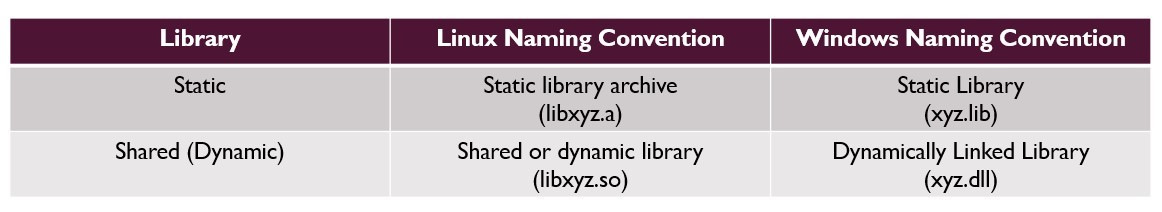
## What are shared (dynamic) libraries?

A shared or dynamic library is collection of pre-compiled object files (in the form of shared library objects) and export headers, which allows code reuse without exposing the implementation. Unlike the use of static libraries, which generates a fully linked and re-located application binary use of shared libraries results in a partially linked binary?



The actual linking process is completed in either of the two ways:

* **Load time linking:** the dynamic linker (ld on Linux based systems) loads the specified shared library objects at load time
* **Run time linking:** application binary requests the dynamic linker to load specified shared library objects at runtime.



## Shared libraries - strong points

* The biggest advantage offered by shared libraries - is that the library implementation itself can be updated on the fly (i.e. on the target system) without the need to recompile the application itself
* C/C++ application uses a lot of shared resources
* Preferred modern methodology to reuse code on multi-tasking OS

## Some caveats to using shared libraries

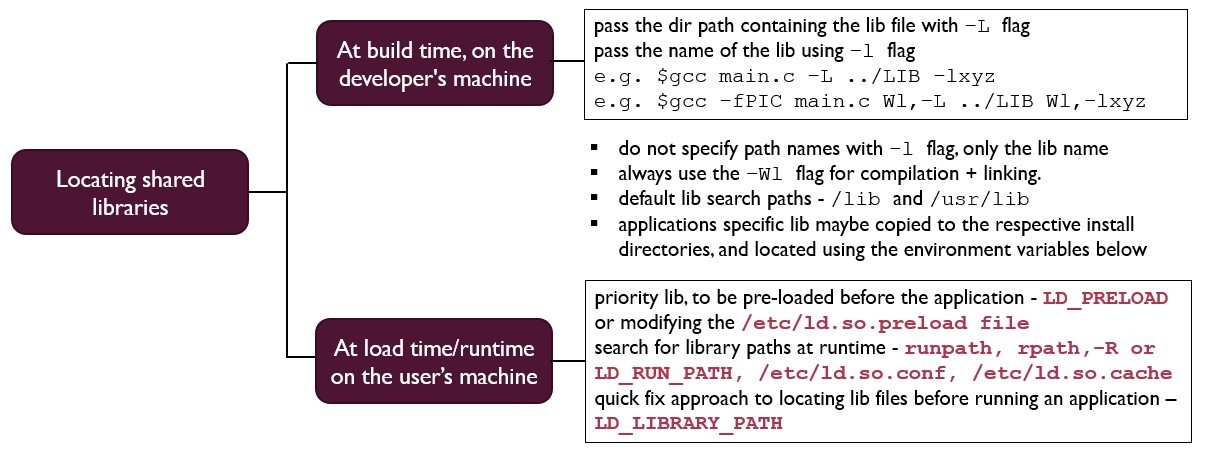
* Application portability is dependent on the availability of the shared library on the target system - applications built using shared library will not work if the shared lib objects cannot be found on the target system.
* the intended functionality to be used from the library is not a large chunk of code
* the entire process of compilation and application packaging is more involved - not done correctly means application will fail and crash!

## What is Position Independent Code (PIC)?

If each process (application binary) were to load its own copy of the library in memory - this would defeat the very purpose of shared libraries as it would result in wasteful memory. Instead shared library object files are generated as Position Independent Code (PIC) object files. Such object files need not be relocated again and hence can be loaded from anywhere in the memory. E.g. if process A has already loaded a copy of shared lib object in memory, the linker can map it to the memory layout of process B without having to reload it again.

PIC avoids having to reload multiple copies of the same shared lib object. This also further helps the OS kernel optimize the sharing of commonly used resources.

To generate PIC shared lib object files they are compiled with -fPIC flag.



## Compiling by linking against a shared library

When building the application on the developer's machine pass the dir path containing the path to the shared lib object files with -L flag. Pass the name of the libraries using -l flag.

The required export headers are included and the APIs are called in the code. The linker only cares at this stage that the required symbols can be found but however does not examine the contents. This is taken care of during load time/runtime when the dynamic linker completes the task of linking.

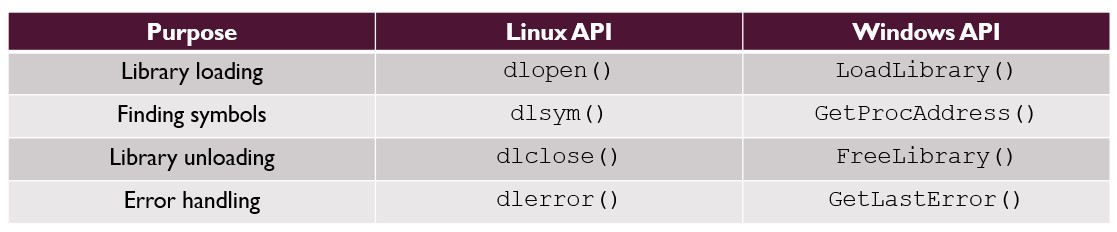
## Load time vs run time linking

### Load time linking

The loader transfers control to the Linux dynamic linker (ld-linux.so) which searches for the required shared library objects first in memory, if not already loaded then in pre-configured search paths and loads the library object. The dynamic linker then relocates the application binary + shared lib objects and completes the linking process. At this stage the application binary is ready for execution.

### Run time linking

Instead of the ld-linux.so automatically loading the required shared lib objects, the application binary explicitly instructs ld-linux.so to load the required shared library objects. This is achieved by using the following system APIs when developing the application:



Reference: <https://www.linkedin.com/pulse/shared-dynamic-libraries-c-santosh-s-malagi/>