Lecture 08

Libraries: principles, building and use.

Advanced Programming - SISSA, UniTS, 2024-2025

Pasquale Claudio Africa

12 Nov 2024

Outline

1. What is a library?
2. Types of library
   - Header-only
   - Static
   - Shared (dynamic)
   - How to link against libraries
3. Static libraries
   - How to build
4. Shared libraries
   - The linking phase
   - The loading phase
   - How to build
   - Dynamic loading
5. How to compile (and use) third-party libraries?

What is a library?

What is a library?

A library is a collection of pre-written code or routines that can be reused by computer programs. These libraries typically contain functions, variables, classes, and procedures that perform common tasks, allowing developers to save time and effort by leveraging existing code rather than writing everything from scratch.

Libraries can be specific to a programming language or more general-purpose, applicable across different languages. Examples include the Standard Template Library (STL) in C++, the Java Class Library, and the Python Standard Library. Additionally, there are third-party libraries created by developers and organizations to extend the capabilities of programming languages or frameworks.

Why libraries are useful?

- **Code reusability**: Libraries provide a set of functionalities that can be used across multiple projects, reducing the need to write the same code over and over again.
- **Modularity**: Libraries promote modular programming by encapsulating specific functionalities into separate modules or components.
- **Abstraction**: Libraries abstract the underlying implementation details, allowing developers to use high-level interfaces without needing to understand the inner workings of the functions provided by the library.
- **Collaboration**: Communities of developers can share and collaborate on libraries, accelerating the development process. Many programming languages have centralized repositories or package managers to facilitate the distribution and installation of libraries.
- **Efficiency**: Libraries are often optimized and well-tested, providing efficient and reliable solutions for common programming tasks.

Components of a C++ library (1/2)

A library provides utilities that may be used to produce executable code. In C or C++, it is usually formed by:

- A set of **header files** that provide the *public interface* of the library, necessary for those who develop software using the library.
- One or more **library files** that contain, in the form of machine code, the *implementation* of the library. They may be *static* and *shared* (also called *dynamic*).

As an exception, there are libraries whose implementation is only contained in header files (thanks to *inline* functions and templates).

These are called *header-only* libraries, and are the easiest to use.

An example of such is Eigen , a powerful library for linear algebra.

Components of a C++ library (2/2)

Header files are only used in the development phase. In production, only **library files** are needed.

Precompiled executables that just use **shared libraries** do not need header files to work. This is why certain software packages are divided into standard and *development* versions; only the latter contains the full set of header files.

For example:

```
sudo apt install python3
```

will install the `python3` executable and the shared libraries it requires to be run, whereas

```
sudo apt install libpython3-dev
```

will download header files, libraries and tools required for building applications **based on** the source code of Python3 (called `CPython`, written in C).

Curated lists of awesome C++ and Python frameworks, libraries, resources, and shiny things.

- Popular GitHub repositories using C++ $(\geq 5k)$!
- awesome-cpp
- awesome-python
- awesome-scientific-python
- awesome-scientific-computing

Types of library

The build process

Header-only libraries

A library formed only by class templates and function templates contains only header files. One example is Eigen , but many others are available.

Using a header-only library is very simple: you have to store the header files in a directory later searched by the preprocessor.

So either you store them in a system include directory, like `/usr/include` or `/usr/local/include` (you must have administrator privileges),
or in a directory of your choice that you will then indicate using the `-I` option of the compiler (actually, of the **preprocessor**).

```
g++ -I/path/to/library/include/ ...
```

Header-only libraries: example

```
# Download Eigen 3.4.0.
wget https://gitlab.com/libeigen/eigen/-/archive/3.4.0/eigen-3.4.0.tar.gz

# Extract the archive to your Desktop.
tar xzvf eigen-3.4.0.tar.gz -C ${HOME}/Desktop

# Compile and run 'example/eigen.cpp'.
g++ -I${HOME}/Desktop/eigen-3.4.0 eigen.cpp -o main_eigen && ./main_eigen
```

As simple as that.

From now on, however, we will deal with libraries that contain machine code, not header-only libraries.

Static vs. shared libraries

- **Static library**: A static library, often denoted by a `.lib` (on Windows) or `.a` (on Unix-like systems) file extension, contains compiled code that is linked directly into an executable at compile time. When you build a program using a static library, a copy of the library's code is included in the final executable. This means that the resulting executable is independent of the original library file; it contains all the necessary code to run without relying on external library files.
- **Shared library** (Dynamic Link Library `.dll` on Windows, Shared Object `.so` on Unix-like systems, Dynamic Library `.dylib` on macOS): A shared library contains code that is *loaded at run-time* when the program starts or during execution. Instead of being included in the executable, the program *references* the shared library, and the operating system *loads the library into memory* when needed. Multiple programs can use the same shared library, which can result in more efficient use of system resources.

A guided example

`main.cpp` (developed by me)

```cpp
#include "mylib.hpp"
...
myfun();
...
```

`mylib` (developed by somebody else)

```cpp
// mylib.hpp
void myfun();

// mylib.cpp
#include "mylib.hpp"

void myfun() {}
```

The build process: preprocessing + compilation

The preprocessing and compilation steps

```
g++ -Imylib/ -c main.cpp
```

produce the object file `main.o`. What does it contain?

```
$ nm -C main.o
0000000000000000 T main
                 U myfun()
```

The `T` in the second column indicates that the function `main()` is actually defined (resolved) by the library. While `myfun()` is referenced but undefined. So, to produce a working executable, you have to specify to the linker another library or object file where it is defined.

Indeed, the linking phase fails...

```
$ g++ main.o -o main
/usr/bin/ld: main.o: in function `main':
main.cpp:(.text+0x9): undefined reference to `myfun()'
collect2: error: ld returned 1 exit status
```

Case 1: I have access to the implementation of `myfun()`

Step 1: compile the object file implementing `myfun()`

```
g++ -c mylib.cpp
```

Step 2: link my application against that object file

```
g++ main.o mylib/mylib.o -o main
```

Now both `main` and `myfun` are resolved:

```
$ nm -C main
00000000000011a9 T main
00000000000011bd T myfun()
...
```

Case 2: the reality

Real-case scenarios are typically much more complex because:

1. **Compilation takes time**!
2. One may need to use symbols defined in **multiple object files**, and compiling all of them and/or carrying out the whole list of object file names can be tedious.
3. If a change is made in `mylib` or it is updated, one has to *recompile* `mylib` and *relink* all their applications using `mylib`.
4. Developers of `mylib` may not be so nice: they want to **hide the actual implementation**. They are ok with providing users with `mylib.hpp` and the corresponding *machine code* (which is not human-readable), but not `mylib.cpp`.
5. **Dealing with multiple dependencies** makes the complexity increase.

This is why, typically, developers of a library provide users with *header files* and a *library* file.

The build process: linking against an external library

Option 1: indicate its full path during linking:

```
g++ main.o /path/to/mylib/libmylib.a -o main
```

Option 2: use the `-L<dir> -l<libname>` options.

```
g++ main.o -L/path/to/mylib -lmylib -o main
```

`-L<dir>` is not needed if the library is stored in a standard directory (typically `/usr/lib` or `/usr/local/lib`).

⚠️ Note that `libxx.a` becomes `-lxx`.

⚠️ If the linker finds a shared library with the same name available in the system and/or in the specified directories, it is given the precedence. If you want to override this behavior, use the `-static` flag.

⚠️ Order matters

When linking, the order matters. Libraries should be listed in reverse order of dependency. Libraries that depend on symbols from other libraries should come first in the list.

So, for example, if `myprogram` depends on `mylibrary1` which on turn depends on `mylibrary2`, then `mylibrary2` should come first:

```
g++ myprogram.o -lmylibrary1 -lmylibrary2 -o myprogram
```

Other permutations are wrong:

```
g++ myprogram.o -lmylibrary2 -lmylibrary1 -o myprogram
g++ -lmylibrary1 -lmylibrary2 myprogram.o -o myprogram
```

Undefined symbols in `main.o` are not searched in the given libraries.

Inspecting the content of a library

The command `nm` works not only with object files and executables, but also with libraries:

```
$ nm -C libmylib.a
...
0000000000000000 T myfun()
...
```

Besides `T` and `U`, the command may use other letters. The most important ones are:

- `D` or `G`: The symbol refers to initialized data.
- `V` or `W`: The symbol is a weak symbol. It basically means that the (ODR) One Definition Rule with not be applied by the linker on those symbols.

A note: If a function declared `inline` has been actually inlined, the corresponding symbol is not present, since inline in this case really means inline. The same happens for a `constexpr` function. If the compiler instead decides to treat them as normal functions, the symbol is marked `W`.

Static libraries: how to use and how to build

Static libraries

Static libraries are the *oldest* and most basic way of integrating third-party code. They are basically a collection of object files stored in a single archive.

At the linking stage of the compilation processes, the symbols (which identify objects used in the code) that are still unresolved (i.e., they have not been defined in that translation unit) are searched into the other object files indicated to the linker and in the indicated libraries, and eventually the corresponding code is inserted in the executable.

How to build a static library?

In practice, libraries result themselves from preprocessing and compiling their corresponding source codes. In our example:

```
g++ -c mylib.cpp
ar rs libmylib.a mylib.o
```

More in general, a **static library** is just an archive collecting object files:

```
g++ -c a.cpp b.cpp c.cpp d.cpp // Create object files.
ar rs libxx.a a.o b.o c.o
ar rs libxx.a d.o // You can add one more.
```

Option `r` adds/replaces an object in the library. Option `s` adds an index to the archive, making it a searchable library.

The command `ar -t libxx.a` lists all object files contained in the archive.

Pros and cons of static libraries

Pros

- The resulting executable is self-contained, i.e., it contains all the instructions required for its execution.

Cons

- If an external library receives an update (such as improvements or bugfixes), the user has to relink its code against the new version.
- We cannot load symbols dynamically, on the base of decisions taken at run-time (it's an advanced stuff, we will deal with it later).
- The executable might become large.

Shared libraries

Shared libraries

With shared libraries, the mechanism by which code from the library is integrated into your own is very different than the static case.

- The **linker** ensures that symbols that are still unresolved are provided by the library.
- However, the corresponding code is not inserted, and the symbols remain unresolved.
- Instead, a reference to the library is stored in the executable for later use by the **loader** (or dynamic loader). This special program looks for the libraries and loads the code corresponding to the symbols that are still unresolved *at run-time*.

⚠️ **The linker and the loader are two different programs.**

Shared libraries: the linking phase

Versioning and naming schemes

Version vs. release

The *version* is an identifier typically represented by a sequence of numbers, indicating instances of a library with a common public interface and functionality. I recommend you to stick with the Semantic Versioning convention.

Naming scheme

- **Link name:** Used in the linking stage with the `-lmylib` option, of the form `libmylib.so`.
- **`soname` (Shared Object Name):** Looked after by the loader, typically formed by the link name followed by the major version number, e.g., `libmylib.so.3`.
- **Real name:** The actual file storing the library with the full version number, e.g., `libmylib.so.3.2.4`.

Library versions in action

The `ldd` command lists all shared libraries used by an executable (or another shared library):

```
ldd /usr/bin/octave-cli | grep fftw3.so
libfftw3.so.3 => /lib/x86_64-linux-gnu/libfftw3.so.3 (...)
```

The loader searches for the library in special directories and finds `/lib/x86_64-linux-gnu/libfftw3.so.3`. This library is used when launching Octave.

If there's a new release, placing the corresponding file in the `/lib/x86_64-linux-gnu` directory, and resetting symbolic links, will make Octave use the new release without recompiling (and this is what happens when, for example, you upgrade a package via `apt` or similar).

Dependency management

```
$ ls -l /lib/x86_64-linux-gnu/libfftw3.so
... /lib/x86_64-linux-gnu/libfftw3.so -> libfftw3.so.3.5.8
```

This means that `libfftw3.so.3` is a symbolic link to `libfftw3.so.3.5.8`. Hence, we are actually using version 3.5.8 of `libfftw3`.

Another nice thing about shared libraries is that they may depend on another shared library. This information can be encoded when creating the library. For instance:

```
ldd /usr/x86_64-linux-gnu/libumfpack.so
...
libblas.so.3 => /usr/lib/libblas.so.3
```

The UMFPACK library is linked against version 3 of the BLAS library. This helps to avoid using an incorrect version of dependent libraries.

Shared libraries: the linking phase (1/2)

You then proceed as usual:

```
g++ -I/path/to/mylib -c main.cpp
g++ main.o -L/path/to/mylib -lmylib -o main
```

The linker looks for `libmylib.so` in system and/or in the specified directories, controls the symbols it provides, and verifies if the library contains a `soname`. If it doesn't, the link name `libmylib.so` is assumed to be also the `soname`.

For example, `libumfpack.so` provides a `soname` (of course, this has been taken care of by the library developers). If you wish, you can check it:

```
$ objdump -p /lib/x86_64-linux-gnu/libumfpack.so | grep SONAME
  SONAME               libumfpack.so.5
```

Shared libraries: the linking phase (2/2)

Being `libmylib.so` a shared library, the linker does not integrate the code of the resolved symbols into the executable. Instead, it just controls that the library provides the symbols and inserts the information about the soname of the library in the executable:

```
ldd main
libmylib.so.2 => /path/to/libmylib.so.2 (...)
```

In conclusion, linking a shared library is not more complicated than linking a static one. However, knowing what happens "under the hood" may be useful to tackle unexpected situations.

⚠️ **Even if the linker has found the library, it does not mean that the loader will find it as well!**

Shared libraries: the loading phase

Where does the loader search for shared libraries?

The loader has a different search strategy with respect to the linker. It looks in `/lib`, `/usr/lib`, and in all the directories contained in `/etc/ld.conf` or in files with the extension `conf` contained in the `/etc/ld.conf.d/` directory.

If you want to permanently add a directory in the search path of the loader, you need to add it to `/etc/ld.conf` or add a conf file in the `/etc/ld.conf.d/` directory with the name of the directory and then launch `ldconfig`. This command rebuilds the database of the shared libraries and should be called every time one adds a new library (for example, `apt` does it for you, and moreover, `ldconfig` is launched at every boot of the computer).

Launching the command `sudo ldconfig -n directory` has the same effect, but in this case modifications will remain valid until the next restart of the computer.

⚠️ All these operations require you to act as an administrator, for instance using the `sudo` command. Safer alternatives are in the next slide.

Alternative ways of directing the loader

1. **Setting the environment variable `LD_LIBRARY_PATH`:** It contains a colon-separated list of directory names where the loader will first look for libraries.

   ```
   # Permanently, for the current terminal session:
   export LD_LIBRARY_PATH+=:dir1:dir2
   ./main

   # Or, temporarily valid for a single command:
   LD_LIBRARY_PATH+=:dir1 ./main
   ```

2. **With the special linker option, `-Wl,-rpath,directory`:** During the compilation (linking stage) of the executable, for instance

   ```
   g++ main.cpp -Wl,-rpath,/path/to/mylib -L/path/to/mylib -lmylib
   ```

   The loader will look in `/path/to/mylib` before the standard directories. You can use also relative paths.

Shared libraries: how to build

How to build a shared library

1. **Compile the source files:**

   ```
   g++ -fPIC -c mylib.cpp
   ```

   `PIC` stands for Position-Independent Code.

2. **Create the library:**

   ```
   g++ -shared mylib.o -Wl,-soname,libmylib.so.1 -o libmylib.so.1.0
   ```

   Note: The library's real name is `libmylib.so.1.0`.

3. **Create symbolic links for version control:**

   ```
   ln -s libmylib.so.1.0 libmylib.so.1
   ln -s libmylib.so.1 libmylib.so
   ```

Linking the executable against the shared library

Compile the executable, linking the library:

```
g++ -I/path/to/mylib -c main.cpp
g++ main.o -L/path/to/mylib -lmylib -o main
```

However, running the executable may result in an error:

```
./main error while loading shared libraries:
    libmylib.so.1: cannot open shared object file: No such file or directory
```

To fix this, direct the loader as explained in the previous section, for instance by modifying `LD_LIBRARY_PATH` or changing the `rpath`:

```
g++ main.o -Wl,-rpath,/path/to/mylib -L/path/to/mylib -lmylib -o main
```

Now, the executable works as expected!

Releasing a new version

Assuming a new release (e.g., version 1.1), compile and link the new library without recompiling the executable:

```
g++ -c -fPIC mylib.cpp # mylib.cpp has some new features!
g++ -shared mylib.o -Wl,-soname,libmylib.so.1 -o libmylib.so.1.1
ln -s libmylib.so.1.1 libmylib.so.1
ln -s libmylib.so.1 libmylib.so
```

Now, running the executable uses the updated library without recompilation or relinking.

Note

For smaller projects without versioning, you can use the same name for link name, `soname`, and real name (e.g., `libmylib.so`). In this case, the `-Wl,soname` option can be omitted and the symbolic links are not needed.

Summary

- Object files should be compiled with the `-fPIC` option.
- The link name is used by the linker for matching symbols.
- The `soname` is used by the loader and is specified during library creation.
- Symbolic links can direct the loader to the desired library (useful for versioning).
- Use `-Wl,-rpath` during linking or set `LD_LIBRARY_PATH` for directory search during development.

Shared libraries: dynamic loading

Dynamic loading and plugins

Shared libraries offer two intriguing features:

1. Dynamic loading of the library.
2. Dynamic loading of symbols from the library.

These features form the foundation for implementing *plugins* (and are also employed in Python modules).

Dynamic loading is a fundamental aspect of a plugin architecture, allowing an application to load parts of its implementation dynamically based on user requests.

⚠️ **This is a very advanced topic. For more information, have a look at this interesting post (source code here).**

Pros and cons of shared libraries

Pros

1. Updating a library has an immediate effect on all codes linking against it. No recompilation or relinking is needed.
2. Executable is smaller since the code in the library is not duplicated.
3. We can load libraries and symbols runtime (*plugins*).

Cons

1. Executables depend on the library. If you delete the library, all codes using it won't run anymore.
2. Both the linking phase and **the loading phase** need careful management, especially when dealing with different library versions installed.

How to compile (and use) third-party libraries?

A (very) general guide

1. **Obtain the library**.
2. **Read the documentation**.
3. **Compile the library**.
4. **Install the library**, i.e., store header files and the generated (static or shared) library files into a convenient folder.
5. **Integrate it in your project**, i.e., include the folder containing header files and add proper link flags. In the case of shared libraries, don't forget to redirect the loader.

Looks easy, doesn't it? 😎

Actually, the crux lies in **step 3**:

- The library may have dependencies on other libraries. 😈
- Fortunately, some libraries use automatic build systems, simplifying the compilation process ... but forcing us to learn how to use these tools! 😅

➡️ Introduction to Makefile and CMake.