Lab 06

Plugins

Paolo Joseph Baioni

November 14, 2025

How to use a third-party library

Basic compile/link flags:

```
$ g++ -I${mkLibrarynameInc} -c main.cpp
$ g++ -L${mkLibrarynameLib} -llibraryname main.o -o main
```

Warning: by mistake, one can include headers and link against libraries related to different installations/versions of the same library! The compile, link and loading phase may succeed, but the executable may crash, resulting in a very subtle yet painful error to debug!

Using shared libraries

Shared libraries:

- User point of view
- Developer point of view

Users need to take care to

- **linking** phase during compilation
- **loading** phase at execute time

Developers usually contribute to a

- **development** phase
- **release** phase

Versions and releases

The *version* is a symbol (typically a number) by which we indicate a set of instances of a library with a common public interface and functionality.

Within a version, one may have several *releases*, typically indicated by one or more numbers (major and minor or bug-fix). A new release is issue to fix bugs or improve of a library without changes in its public interface. So a code linked against version 1, release 1 of a library should work (in principle) when you update the library to version 1, release 2.

Normally version and releases are separated by a dot in the library name: `libfftw3.so.3.3.9` is version 3, release 3.9 of the `fftw3` library (The Fastest Fourier Transform in the West).

Naming scheme of shared libraries (Linux/Unix)

We give some nomenclature used when describing a shared library

- **link name**: name used in the linking stage when you use the `-lmylib` option.
- **soname**: *shared object name* looked after by the *loader* stage.
- **full name**: name of the actual file that stores the library.

Example:

`fftw`: is the *link name*.

`libfftw3.so.3`: is the *soname*.

`libfftw3.so.3.3.9`: is the *real name* of the file.

We call a *fully qualified library* a soname that contains the full path to the library.

How does it work?

The command `ldd` lists the shared libraries used by an object file.

Example:

```
$ ldd ${mkOctavePrefix}/lib/octave/6.2.0/liboctave.so
...
libfftw3.so.3 -> /u/sw/toolchains/
   gcc-glibc/11.2.0/pkgs/fftw/3.3.9/lib/libfftw3.so.3
...
```

- `Octave` has been linked to `libfftw3`.

The **loader** searches the occurrence of this library, finding his full qualified name.

Which release?

```
$ ls -l ${mkFftwLib}/libfftw3.so.3
/full/path/to/libfftw3.so.3 -> libfftw3.so.3.3.9
```

I am in fact using release `3.9` of version 3.

Explanation

The executable (`octave`) contains the information on which shared library to load, including version information (its `soname`). This part has been taken care by the developers of `Octave`.

When I launch the program the loader looks in special directories, among which `/usr/lib` for a file that matches the `soname`. This file is typically a symbolic link to the real file containing the library.

If I have a new release of `fftw3` version 3, let's say 3.4.1, I just need to place the corresponding shared library file, reset the symbolic links and automagically `octave` will use the new release (this is what `apt` does when installing a new update in a Debian/Ubuntu system, for example).

No need to recompile anything!

Another nice thing about shared libraries

A shared library may depend on another shared library. This information may be encoded when creating the library (just as for an executable, we will see it later on).
For instance

```
$ ldd /usr/lib/x86_64-linux-gnu/libumfpack.so.5
...
libblas.so.3 => /lib/x86_64-linux-gnu/libblas.so.3
...
```

The UMFPACK library is linked against version 3 of the BLAS library.

This prevents using incorrect version of libraries. Moreover, when creating an executable that needs UMFPACK I have to indicate only `-lumfpack`!
Note: This is not true for static libraries: you have to list all dependencies.

How to link against a shared library

It is now sufficient to proceed as usual

```
g++ -I${mkFFtwInc} -c main.cpp
g++ -L${mkFFtwLib} -lfftw3 main.o -o main
```

The linker finds `libfftw3.so`, controls the symbols it provides and verifies if the library contains a `soname` (if not the link name is assumed to be also the soname).
Indeed `libfftw3.so` provides a `soname`. If we wish we can check it:

```
$ objdump libx.so.1.3 -p | grep SONAME
SONAME   libfftw3.so.3
```

(of course this has been taken care by the library developers).

Being `libfftw3.so` a shared library the linker does not resolve the symbols by integrating the corresponding code in the executable. Instead, it inserts the information about the `soname` of the library:

```
$ ldd main
libfftw3.so.3 => /full/path/to/libfftw3.so.3
```

The loader can then do its job now!

In conclusion, linking with a shared library is not more complicated than linking with a static one.

Remember: By default if the linker finds both the static and shared version of a library it gives precedence to the shared one. If you want to by sure to link with the static version you need to use the `-static` linker option.

Directories where the loader searches the shared libraries

- `/lib*` or `/usr/lib*` .
- Additional directories can specified by `/etc/ld.conf` and files inside the directory `/etc/ld.conf.d/`.

The command `ldconfig` rebuilds the data base of the shared libraries and should be called every time one adds a new library (of course `apt` does it for you, and moreover `ldconfig` is launched at every boot of the computer).

Note: all this operations require you act as superuser, for instance with the `sudo` command.

Alternative ways of directing the loader

- Setting the environment variable `LD_LIBRARY_PATH`. If it contains a comma-separated list of directory names the loader will first look for libraries on these directories (analogous to `PATH` for executables):

  ```
  export LD_LIBRARY_PATH+=:dir1:dir2
  ```

- With the special flag `-Wl,-rpath=directory` during the compilation of the executable, for instance

  ```
  g++ main.cpp -o main -Wl,-rpath=/opt/lib  -L. -lsmall
  ```

  Here the loader will look in `/opt/lib` before the standard directories. You can use also relative paths. See e.g. gcc directory options and gcc link options

- Launching the command `sudo ldconfig -n directory` which adds `directory` to the loader search path (superuser privileges are required). This addition remains valid until the next reboot of the computer. **Note**: prefer the other alternatives!

How to build a shared library?

We will dedicate another lecture to this issue, where we will also show how to handle shared libraries and symbols dynamically. For the moment we need to know only the following:

- When building a shared library we need to pass the option `-shared` to the linker
- Object code used in a shared library must be *position independent* (compiler option `-fPIC`)

Basic build of a shared library starting from an object file:

```
$ g++ -shared -Wl,-soname,libutility.so utility.o -o libutility.so
```

Exercise 1: quadrature rules with plugins

- Implement a code that enables the user to compute the integral of a scalar function by selecting the quadrature rule as the name of a dynamically loadable object;
- The dynamically loadable object should define a function with the following signature: `double integrate(std::function<double (double)>, double a, double b)`.
- Implement plugins for midpoint and trapezoidal rule.
- Implement a plugin for quadrature with adaptive refinement.