Advanced Methods for Scientific Computing (AMSC)
Lecture title: Libraries

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2024/2025

What is a library?

A library provides utilities that may be used to produce an 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 to those who develop software using the library.
- one or more library files that contain, in the form of machine language, the implementation of the library. They may be static and shared (also called dynamic).

As an exception, template-only libraries, like the Eigen, provide only header files.

Precompiled executables which just use shared libraries do not need header files to work (that's why certain software packages are divided into standard and development version, only the latter contains the full set of header files)

How to use header-only libraries

A library formed only by class templates and function templates contains only by header files. One example is `Eigen`, but there are quite a few other ones.
In the case of a header-only library life 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<myncludedir> ....
```

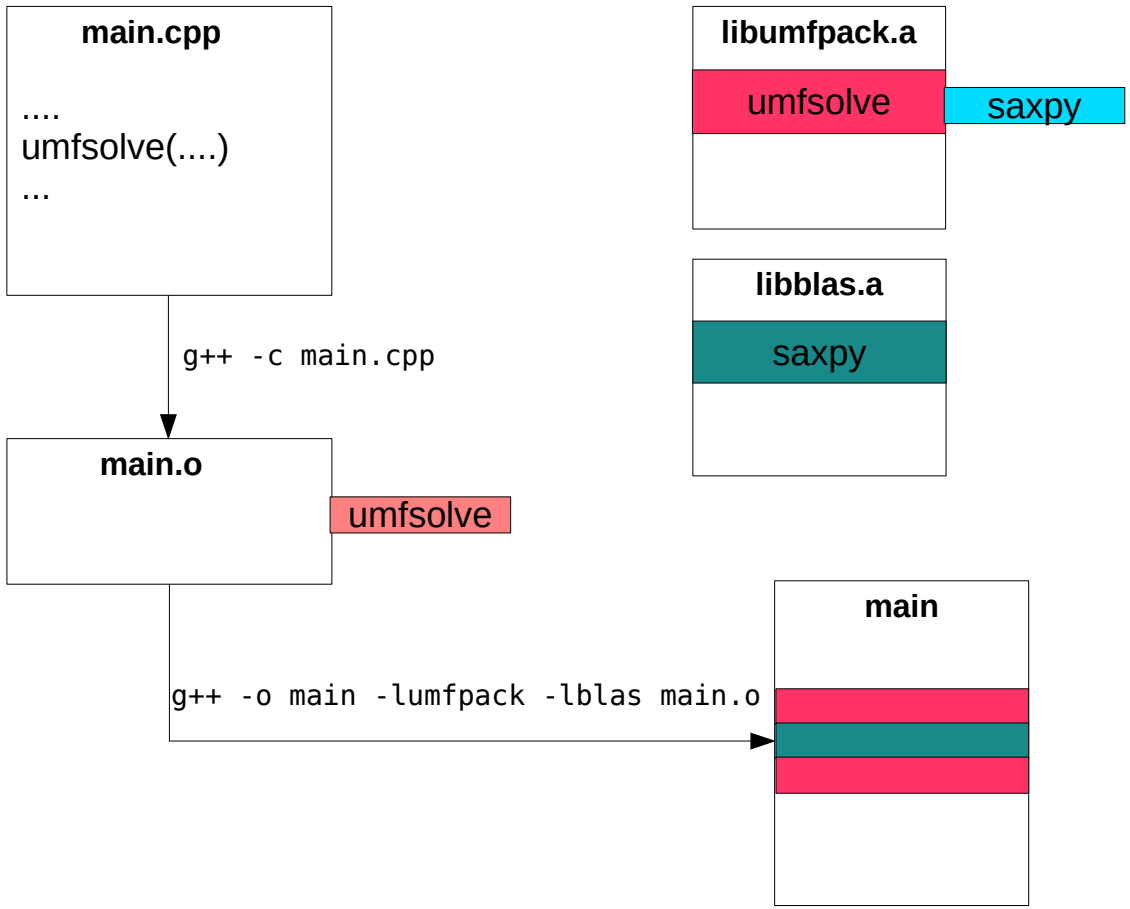
As simple as that. From now on, however, we will deal with library that contains machine code, not header-only libraries.

Static libraries

Static libraries are the oldest and most basic way of integrating "third party" code. They are basically a collection of object file 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.

The handling of static libraries



Advantages and disadvantages of static libraries

PROS

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

CONS

- To take advantage of an update of an external library we need to recompile the code (at least to replicate the linking stage), so we need the availability of the source (or at least of the object files);
- We cannot load symbols dynamically, on the base of decisions taken run-time (it's an advanced stuff, we will deal with it in another lecture);
- The executable may become large.

How to use a static library

If you want to link with a static library either you indicate its full path during linking

```
g++ main.o /opt/lib/libxx.a
```

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

```
g++ main.o –L/home/lib –lxx
```

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

Warning: If the linker finds that in the standard directory or in the directory indicated by `-L<dir>` there is a shared library (in the example `libxx.so`), it does not perform the static link. If you want to override this behavior use the `-static` option.

Order matters!

Beware that the order in which you list static libraries during the link process is important.

Suppose that you want to link your main with the static version of the umfpack library `libumfpack.a`, which itself uses some of the tools provided by the BLAS (`libblas.a`).

```
g++ main.o -lumfpack -lblas -o main
```

works, while

```
g++ main.o -lblas -lumfpack -o main
```

doesn't!. Order matters when linking static libraries.

And this is wrong:

```
g++ -lblas -lumfpack main.o -o main
```

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

How to create a static library

It is very simple. The library is just a collection of object files.

```
g++ -c a.cpp b.cpp c.cpp e.cpp //create object files
ar rs libxx.a a.o b.o c.o
ar rs libxx.a e.o //you can add one at a time
```

Option `r` adds/replaces an object in the library. Option `s` adds an index to the archive, making it a searcheable library.
During the development phase of a library it is simpler to use static libraries. When everything is ok you may want to create the shared version.

The command

```
ar -t libxx.a
```

lists all object files contained in the library.

What is contained in a library (it applies also to shared libraries)

To see the symbols contained in a library (or in an object file, or in an executable) you may use the command nm --demangle (possibly piped with `grep`).

```
>nm --demangle libsmall.a| grep foo
0000ac0 T nms::foo(double)
0001dc0 U nsm::foo2(int)
```

The T in the second column indicates that the function `nms::foo(double)` is actually defined (resolved) by the library. While `nsm::foo2(int)` 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.

Why demangling?

Remember that functions (free functions or methods) in C++ are identified by their fully qualified name (`nms::foo` for instance), the type of the arguments and, in the case of methods, the possible presence of the const qualifier. This is the basis of function overloading.

At the level of machine code, different members of a set of overloaded functions (like `foo(double)` and `foo(int)`) produce different symbols (a process termed name mangling): they are in fact treated as different functions!.

Demangling allows to recover their identifier, in a human readable form.

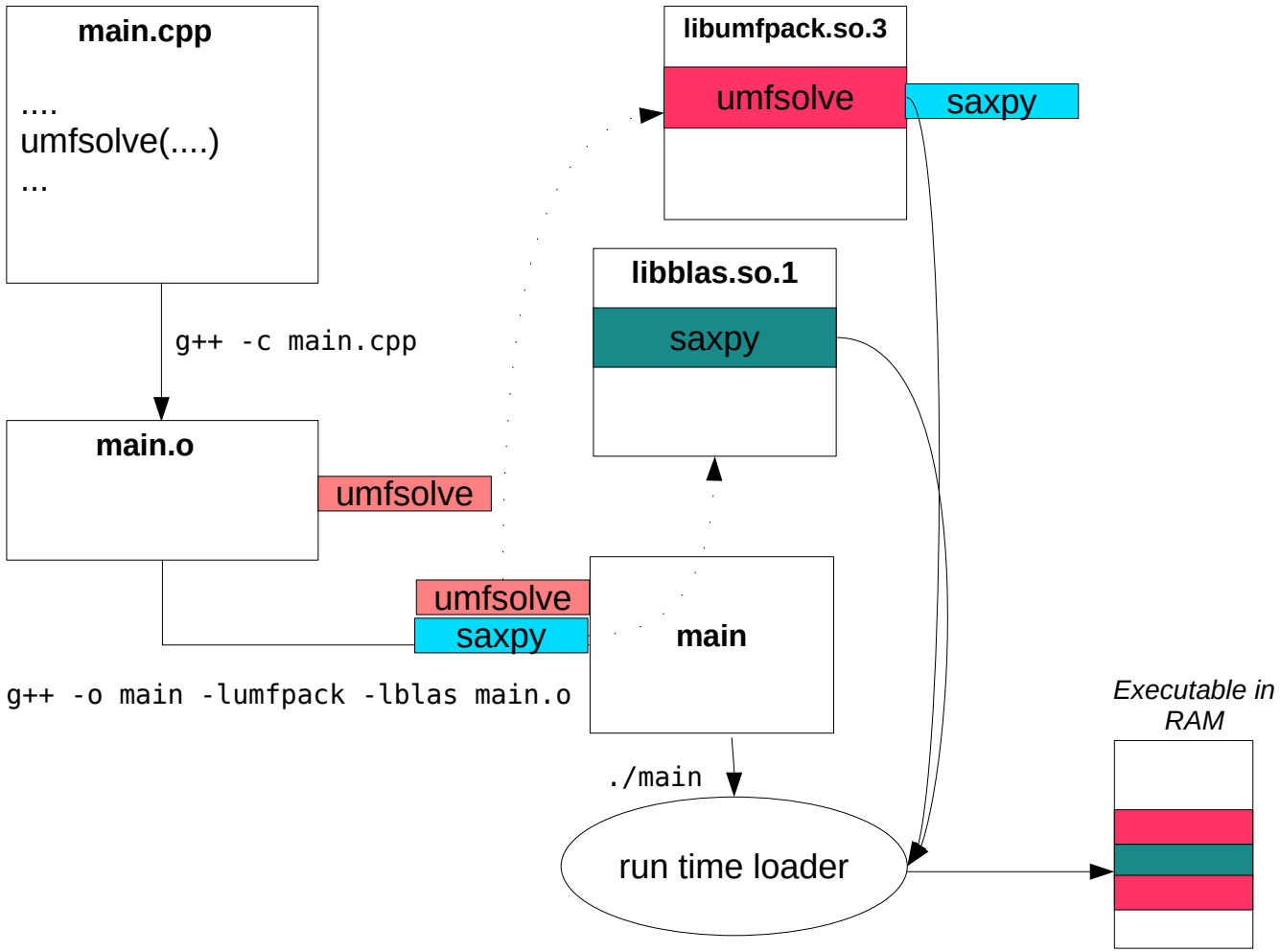
Dynamic (shared) libraries

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

The linker just makes sure that the symbols that are still unresolved are indeed provided by the library, with no ambiguities. But the corresponding code is not inserted, and the symbols remains unresolved. Instead, a reference to the library is stored in the executable for later use by the loader.

When the executable is launched, a special program, called loader (a.k.a dynamic linker) is called, which searches the libraries and loads the code corresponding to the symbols still unresolved.

The handling of shared libraries



Advantages and disadvantages of shared libraries

PROS

- Updating a library has immediate effect on all codes linking the library. No recompilation is needed.
- Executable is smaller since the code in the library is not replicated;
- We can load libraries and symbols run time (plugins).

CONS

- Executables depend on the library. If you delete the library all codes using it won't run anymore!
- You may have different versions of a library. Making sure that your code links to the correct version needs care (and sometimes it causes headaches).

How to link with a shared library

The procedure is identical than with static libraries. For instance,

```
g++ -c main.cpp
g++ -o main main.o -L/usr/lib -lfftw3
```

links `main.o` with the library `libfftw3.so` in `/usr/lib` (you can also give the full path of the library). The specification of `/usr/lib` is not really necessary: it is a directory normally searched by the linker.

This is what you do at compilation time. But it is useful to have an idea on how shared libraries are handled in a Unix system like Linux.

Shared libraries in Linux/Unix

In discussing shared libraries we need first to distinguish between the linking phase of the compilation process and the loading of the executable.

The linking aspects are not so different than in the static case. To fully understand the process of loading shared libraries (and the selection of the correct version) in Linux (and generally in POSIX-Unix systems) we need to discuss the difference between *version* and *release* and the corresponding naming scheme.

Using shared libraries

We also distinguish the bare *use* of a shared library and how to develop a shared library. The latter aspect will be covered in a later lecture.

We will see that in general the use of a shared library is rather simple, but it is useful to know some technical details on how "professional" libraries are handled in a Unix-system (also to understand the cause of possible errors)

Versions and releases

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

Within a version, we may have several releases, typically indicated by one or more numbers (mayor and minor or bug-fix). A new release is issued to fix bugs or improve of a library without changing 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.3.2.4` is version 3, release 2.4 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. It's the name used in the linking stage when you use the `-lmylib` option. It is of the form `libmylib.so`. The normal search rules apply. Remember that it is also possible to give the full path of the library instead of the `-l` option.
- soname (shared object name). It's the name looked after by the *loader*. Normally it is formed by the link name followed by the version. For instance, `libfftw3.so.3`. It is *fully qualified* if it contains the full path of the library: e.g. `/usr/lib/libfftw3.so.3`.
- real name. It's the name of the actual file that stores the library. For instance, `libfftw3.so.3.2.4`

How does it works?

The command `ldd` lists the shared libraries used by an executable.
On my computer:

```
> ldd /usr/bin/octave | grep fftw3.so
libfftw3.so.3 => /usr/lib/libfftw3.so.3 (...)
```

It means that my version of octave has been linked (by its developers) against version 3 of the `libfftw3` library, as indicated by the `soname` (the one in red).
The loader searches the occurrence of this library in special directories (we will discuss about it later) and has indeed found `/usr/lib/libfftw3.so.3` (full qualified name). This is the library used if I launch `octave` in my computer.

Which release? Well, lets take a closer look at the file

```
> ls -l /usr/lib/libfftw3.so.3
/usr/lib/libfftw3.so.3 -> libfftw3.so.3.2.4
```

I am in fact using release `2.4` of version 3.

Got it?

The executable (`octave`) contains the information of 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, lets say 2.5, I just need to place the corresponding file in the `/usr/lib` directory, reset the symbolic links and automagically `octave` will use the new release (this is what `apt-get` does when installing a new library in a Debian/Ubuntu system).

No need to recompile anything!

A little parenthesis...

What does

```
/usr/lib/libfftw3.so.3 -> libfftw3.so.3.2.4
```

mean?
It means that `/usr/lib/libfftw3.so.3` is a symbolic link to `/usr/lib/libfftw3.so.3.2.4`. A symbolic link is a sort-of reference to a file that can be used exactly as the file it is bound to (open, modify etc.). You can create a symbolic link with the command

```
ln -s <TargedFile> <LinkName>
```

You may think a symbolic link as the equivalent in the word of the filesystem of a C++ reference.

A nice things 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 in a later lecture). For instance,

```
>ldd /usr/lib/libumfpack.so
...
libblas.so.3gf => /usr/lib/libblas.so.3gf
```

The `UMFPACK` library is linked against version `3gf` of the `BLAS` library (a particular implementation of the BLAS).

This fact helps to avoid using an incorrect version of dependent libraries. Not only: when linking an executable that needs `UMFPACK` you have to indicate `-lumfpack` only!

This is not true for static libraries: you have to list all dependencies.

The final touch

You then proceed as usual

```
g++ -c main.cpp
g++ -o main main.o -lfftw3
```

The linker finds `libfftw3.so` in `/usr/lib`, controls the symbols it provides and verifies if the library contains a `soname`. If it doesn't, the link name `libfftw3.so` is assumed to be also the soname. Indeed `libfftw3.so` provides a `soname` (of course this has been taken care by the library developers). If you wish you can check it:

```
> objdump libfftw3.so -p | grep SONAME
SONAME libfftw3.so.3
```

Being `libfftw3.so` a shared library the linker does not integrate the code of the resolved symbols into the executable. Instead, it inserts the information about the `soname` of the library in the executable:

```
> ldd main
libfftw3.so.3 => /usr/lib/libfftw3.so.3 (...)
```

The loader can then do its job now!.

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 takle unexpected situations.

Remember: If the linker finds both 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.

Linker and loader are two different things

What you should keep in mind is that linker and loader are two different programs.
If the linker has found the library it does not mean that the loader will find it as well!

Let's have a closer look on where the loader searches for libraries.

Where does the loader search for shared libraries?

It looks in `/lib`, `/usr/lib` and in all the directories contained in `/etc/ld.conf` or in files with extension `conf` contained in the `/etc/ld.conf.d/` directory (so the search strategy is different than that of the linker!)

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`).

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-get` does it for you, and moreover `ldconfig` is launched at every boot of the computer).

All these operations require you act as administrator, for instance using the `sudo` command, better use the alternatives in the next slide

Alternative ways of directing the loader

- Setting the environment variable `LD_LIBRARY_PATH`. If it contains a column-separated list of directory names the loader will first look for libraries on those directories.

  `export LD_LIBRARY_PATH+=:dir1:dir2`

- With the special linker option, `-Wl,-rpath,directory` during the compilation (linking stage) of the executable, for instance

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

  The loader will look in `/opt/lib` before the standard directories. You can use also relative paths.

- Launching the command `sudo ldconfig -n directory` which adds `directory` to the loader search path (you need to have superuser privileges). This addition remains valid until the next restart of the computer. NOTE: prefer the first two alternatives!

Compiling objects for a shared library

The first step, as usual, is to compile the source files to produce object (.o) files. To be part of a shared library source code must be compiled using the `-fPIC` or `-fpic` options (pic=position independent code). The difference between the options is:

- `-fPIC` is more general (it always works) but it may generate larger code;
- `-fpic` may be more efficient, but it may not work in certain platforms.

Here is the command:

```
g++ -std=c++17 -Wall -fPIC -c a.cpp b.cpp
```

Suggestion: use `-fPIC`.
NO main() in libraries!

Creating the library

At this point we can create the library from the object file(s). We should select a `soname`, being this the first version of the library we set it to be `libsmall.so.1`.

To build a shared library we launck the linker using the command g++ (or `clang++`) with the option `-shared`.

```
g++ -shared -o libsmall.so a.o b.o
```

The library real name is `libsmall.so`.

Note: You can add option to link other shared libraries. Remember that the library must be put in place visible to the loader.
We have omitted here the question about versioning, to simplify things. So in this case soname, link name and file name coincide.

A note

A compiled python module is a shared library with a special name.
In the pybind11 folder you find several example of creation of python modules from C++ code, using pybind11.
Some examples uses cmake other uses standard Makefile, so you can see hot iw works.