Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: Shared libraries

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

Dynamic vs. static libraries

In scientific computing (but not only) often code is contained into libraries to which numerical applications link.
For instance, notable libraries are

- blas the basic linear algebra subroutines. It provides highly optimized basic operations on vectors and matrices. There are different implementations, a rather efficient one if the atlas;
- umfpack and MUMPS, two libraries of direct solvers for sparse matrices;
- Trilinos a huge set of libraries for high performance scientific computing;
- PETSC another library for high performance computing.
- LifeV a library of finite elements developed at MOX/EPFL/Emory.

What is a library?

A library in C or C++ is usually formed by

- a set of header files that provide the public interface of the library, and necessary to those who develop software using the library.
- a set of library files that contain, in the form of machine language, the implementation of the library. They may be static and dynamic.

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

On the other hand, precompiled programs which just use a dynamic library do not need the header to be installed (that's why certain software packages are divided into standard and development version, only the latter contains the full set of header files

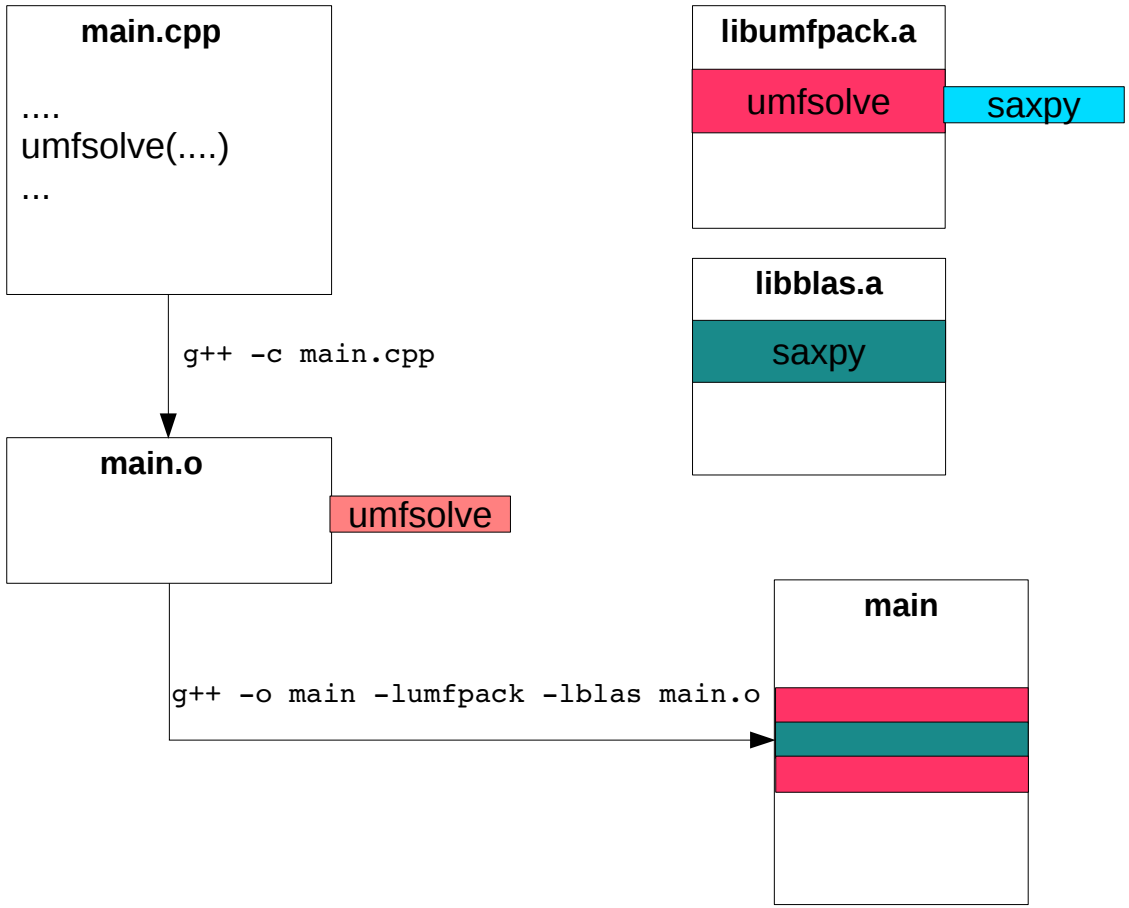
.

Static libraries

Static libraries are the oldest and most basic way of integrating "third party" code in a software code.

At the linking stage of the compilation processes the *symbols* (names of objects/functions etc.) that are still unresolved are searched into the given *libraries* (in the order indicated in the command) and 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. contains all the instructions necessary for its execution.

CONS

- The executable may be large;
- To take advantage of an update of an external library we need to recompile our code (at least to replicate the linking stage), so we need the availability of the source (or at least the object files).
- We cannot load dynamically symbols on the base of decisions taken run-time.

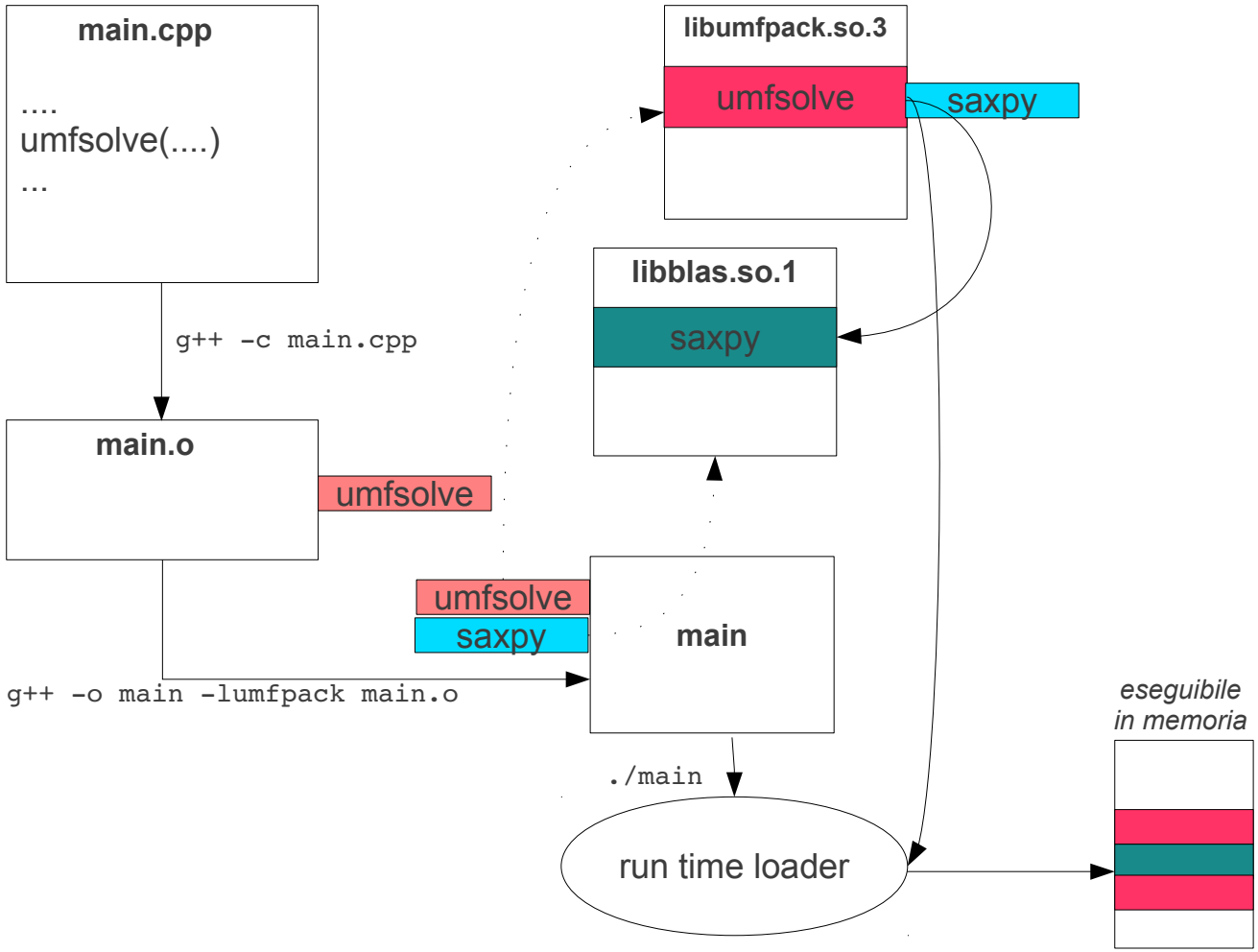
Dynamic (shared) libraries

With shared libraries the mechanism by which code stored in the library is handled is completely different than the static case.

The linker only makes sure that the symbols that are still unresolved are indeed provided by the library, with no ambiguities. But the code is not linked. Instead, the name of the library is stored in the executable for the later use by the loader.

When the executable is launched, a special program, called loader is called, which searches for the libraries and loads into the executable the code corresponding to the symbols still unresolved.

The handling of shared libraries



Advantages and disadvantages of shared libraries

PROS

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

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 requires care (see next slides).

Using shared libraries

We need to distinguish the bare *use* of a library and how to develop a library.

Furthermore, in the latter case we need to distinguish between the *development* phase (i.e. the library is not yet available to the "general public") and the *release* phase (i.e. when you make your library available to others).

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.

We will treat the linking aspects later on, yet to fully understand how the handling of shared libraries (and the selection of the correct version) works on Linux (and generally in POSIX-Unix systems) we need to understand the difference between *version* and *release* and the corresponding naming scheme.

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 (mayor 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.3.2.4` 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.
- real name. It's the name of the actual file that stores the library. For instance `libfftw3.so.3.2.4`

Symbolic links

Before going further we need to introduce a very important feature of Unix systems: the symbolic link.
A symbolic link is a *special file* that works like a *pointer* to another file. It is created with the `ln -s` command:

```
ln -s afile alink
```

Now `alink` is a symbolic link to `afile`. If we type

```
ls -l alink
```

we get

```
alink -> afile
```

Accessing `alink` is indeed the same than accessing `afile`.

NOTE: Remember the `-s`, otherwise you create a hard link (we do not discuss hard links here).

How it works?

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

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

It means that the version of octave I have has been linked (by its developers) against version 3 of the `libfftw3` library, as indicated by the `soname`.
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 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, 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!

Where the loader searches 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 I want to permanently add a directory in the search path of the loader I 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).

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

Another nice things about shared libraries

A shared library is for certain aspect similar to an executable. In particular, when creating a shared library that has symbols provided by another shared library, you need to specify the latter library so that its `soname` is known. For instance

```
>ldd /usr/lib/libumfpack.so.5.4.0
...
libblas.so.3gf => /usr/lib/libblas.so.3gf
```

The UMFPACK library, version 5.4.0 is linked against version 3gf of the BLAS library (a particular implementation of the BLAS).

This fact helps in avoiding using incorrect version of libraries!

Note: This is not true for static libraries, which may be created with completely unresolved symbols.

How to link with a shared library

Let assume we are developing a code, for simplicity contained in a single file `main.cpp` (the treatment of more complex situations is immediate) that needs to link to a library, for instance the `fftw3` seen before, and we want to use the shared library.

We may note that in the `/usr/lib/` directory we have also a file called `libfftw3.so`, this is the *linker name* of the library and in fact it is a link to the real library.

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

This is the file used by the linker to verify the existence of the library, to control of the symbols required by our code, and find the `soname`.

How to link with a shared library

It is now sufficient to proceed as usual

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

(note that `-L/usr/lib` is not strictly necessary since the linker always look in that directory.)
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 => /usr/lib/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.

Note: 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.

How to create a dynamic library

We now face the problem of how to create a shared library. We will use the example in the SharedLibrary directory.
We want to create a shared library called `libsmall.so` from one file `smalllib.cpp` (this is just an example, normally libraries are made by several source files!).

Extract of smalllib.hpp

```
class foo{
public:
  foo();
..};
```

Extract of smalllib.cpp

```
foo::foo() {
std::cout<<"Using release 0 of smalllib V. 1"
         <<std::endl;}
```

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 give problem in certain platforms.

Here is the command:

```
g++ -Wall -fPIC -c smalllib.cpp
```

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`. If we want to indicate the `soname` we need to use the option `-Wl,-soname,libsmall.so.1` (`Wl` means that in fact it is a special option for the linker).

```
g++ -shared -Wl,-soname,libsmall.so.1 -o libsmall.so.1.0
smalllib.o
```

The library real name is `libsmall.so.1.0`.

Note: For simplicity we are creating the library in the same directory of the sources. In general one stores the library in another directory.

A final touch

If we want to experiment version control we make the symbolic links,

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

the first for the link name and the second for the `soname`.

The executable

The file main.cpp

```
#include "smalllib.hpp"
int main(){
  foo one;
}
```

We compile it linking the library

```
g++ main.cpp -o main -L. -lsmall
```

The linker finds the library and produces the executable. However the file won't run:

```
./main:error while loading shared libraries:libsmall.so.1
```

Indeed

```
> ldd main
...
libsmall.so.1 => not found
```

Why?

The loader does not find the library because it is not in one of the directory searched by the loader!

However moving libraries to the correct directories is normally done only at the end of the development cycle (when our code is verified and validated). Only then we release the library to the general public. In the meantime we need to have other ways to direct the loader.

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.

  `export LD_LIBRARY_PATH=dir1:dir2`

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

  `g++ main.cpp -o main_dev -Wl,-rpath=. -L. -lsmall`

  (we recall that `.` indicates the working directory. Alternatively one may use `` `pwd` ``). Note the "inverted accents".

- Launching the command `ldconfig -n directory` which adds `directory` to the loader search path (you need to be superuser). This addition remains valid until the next reboot of the computer. NOTE: prefer the other alternatives!

Directing the loader

Let's then re-compile our program with

```
g++ main.cpp -o main_dev -Wl,-rpath=. -L. -lsmall
```

Now:

```
>./main_dev
Using release 0 of smalllib V. 1
```

It works as expected!

Changing release

Assume we have a new, improved, release:

Extract of smalllib2.cpp

```
foo::foo(){
std::cout<<"Using release 1 of smalllib V. 1"
         <<std::endl;}
```

We compile the new library

```
g++ -Wall -fPIC -c smalllib2.cpp
g++ -shared -Wl,-soname,libsmall.so.1 -o libsmall.so.1.1
          smalllib2.o
```

Now, we just need to change the links

```
ln -f libsmall.so.1.1 libsmall.so
ln -f libsmall.so.1.1 libsmall.so.1
```

to obtain (without any recompilation)

```
  >./main_dev
Using release 1 of smalllib V. 1
```

To summarize

- Object files should be compiled with `-fPIC` or `-fpic` option;
- The link name is the name used by the linker to verify symbol matching;
- The `soname` is the name looked after by loader. It is indicated when creating the library;
- By the use of symbolic links the loader may be directed to the real library we want to use;
- The command `ldconfig` rebuilds the data base used by the loader to search for libraries in the "standard" directories.
- The use of `-Wl,-rpath`at the linking stage, or that of the environmental variable `LD_LIBRARY_PATH` allows us to direct the search to other directories, and this is what is normally done in the *development* phase.

A note

What we have described is the structure of a "professional" library, where one wants to set up version control.

For small projects one may decide to use the same name for the link name, the `soname` and the real name (in our example the will be all equal to `libsmall.so`). In this case we can also avoid the `-Wl,soname` option, as well as the symbolic links!

By doing so, of course, we imply that there is no versioning mechanism.

Dynamic loading

Shared libraries allow also two very interesting features: (1) dynamic loading of the library and (2) dynamic loading of the objects in the library. This features are at at the base of *plugins*.

Loading objects from the library will be considered together with the use of *object factories*, which we will describe later, but it may also used directly with functions, with a little trick to avoid name mangling.
An example of loading functions dynamically is in SharedLibrary/DynamicLoading/main_dynlib.cpp