Static/shared libraries and factories

L. Formaggia

October 2013

Contents

1 Introduction

With the generic term *library* one normally indicates a set of functionalities (class, variables, functions etc.) provided by a third party so that they can possibly be integrated into a program. They are not executable, i.e. they do not provide a `main()`.

C and C++ libraries can be given as a set of *header files*, which define the public interface of the library, and *library files*, which contain the implementation (already compiled). Notable exception are the *template libraries* which are formed by just header files (like the `Eigen`).

In the following, with the term *library* we will refer just to the *library files*, being understood that they are normally complemented with corresponding header files (at least those used for *development*).

Any item (function, class, variable) contained in a library is addressed by its *symbol* and indeed it is common usage to use just the term *symbol* to indicate an item stored in a library.

1.1 Static and shared libraries

There are two types of libraries, *static libraries* and *shared libraries* (also called dynamic libraries). More precisely,

- Static libraries have extension `.a` in Linux and `.lib` in Windows. They are basically archives of object code ready to be assembled by the *linker* to produce an executable. The symbols referenced in the program are loaded from the library and assembled into the executable.

- Shared (dynamic) libraries, with extension `.so` (shared object) in Linux and `.dll` (dynamic-link libraries) in Windows. They may be used in two ways:
  - They are indicated when creating an executable (in the linking phase) as their static counterpart. Yet, the referenced symbols are not assembled into the executable, but are instead *loaded run-time*. The latter operation is carried out by the *loader*, a program that is launched whenever a program is loaded into memory to be executed.
  - Dynamic loading. The loading of the symbols of the shared libraries may be controlled by the program dynamically. It is the mechanism at the base of the so-called *plugins*. It may be used, for instance, to load a function or a class object dynamically.

Note: In many Linux distributions many packages come in two forms: the standard package and the development version. The development version contains also the header files and the static libraries, while the standard package usually provides only dynamic libraries.

Let's look at the two type of libraries in more details.

1.2 Static libraries

Software is usually subdivided into *compilation units*. Each compilation unit is formed by a single source file (`.cpp`) and all code contained in the included header files (`.hpp`). The compiler (more precisely the pair of tools *pre-processor+compiler*) translates a compilation unit into an *object file* (`.o`). One or (typically) more object files are then *linked* to produce an executable.

Object files may, for convenience, be collected into a *static library.* For instance, the library `libumfpack.a` collects all the object files that make up the UMFPACK software. In this way all the functionalities of UMFPACK are provided with a single file.

A static library is created using the command `ar`:

```
ar -r -s libmylib.a a.o b.o c.o d.o
```

The Unix command `ar -t` allows us to list all object files stored in a static library. While, the command `nm` (for C++ code you should use the option `--demangle`) allows us to examine all *symbols* stored in the library.

If the item has been defined the symbol is said to be *resolved* (or *defined*). If the item has only been declared the symbol is said to be *unresolved*, or *undefined.*

```
nm --demangle libmesh.a
....
                 U operator new(unsigned long)
....
0000000000000000 T Geometry::Domain1D::Domain1D(double const&, double const&)
```

In the library `libmesh.a` the symbol corresponding to the `operator new` is undefined (it will be obviously provided by a system library), and this is indicated by the presence of the character `U`. While, the constructor of `Domain1D` is resolved, this is indicated by the character `T` (text).

Let us make an example. Our code may call the function `umfsolve` of UMFPACK, this function is only declared but not defined in our code. The resolution of the corresponding symbol is made by the linker when we indicate the library `libumfpack.a` using the option `-lumfpack`.

Conversely, `umfsolve` may need a function of the BLAS (Basic Linear Algebra System), for instance `saxpy`, whose definition is contained in `libblas.a`.

Clearly all symbols must be resolved for an application to be executable. Therefore, in this example the code must be linked with the option `-lumfpack -lblas` (if we use static libraries). *The order is important.*

To summarize, when using static libraries the command

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

creates an executable, called `main`, using the source code `main.cpp` as well as the libraries `libumfpack.a` and `libblas.a`.

If the libraries are stored in one of the directory of the standard path there is no need to indicate the directory. Otherwise, one should use the option `-L<DIRNAME>` to add a directory to the library search path.

The general rule is that the linker *resolves the undefined symbol in the order they are found.* Therefore, the order is important: `-lblas -lumfpack` will give an "unresolved symbol" error.

The final executable will indeed contain the machine code provided by the libraries. It is then self-contained: its execution is independent from the presence of the libraries.

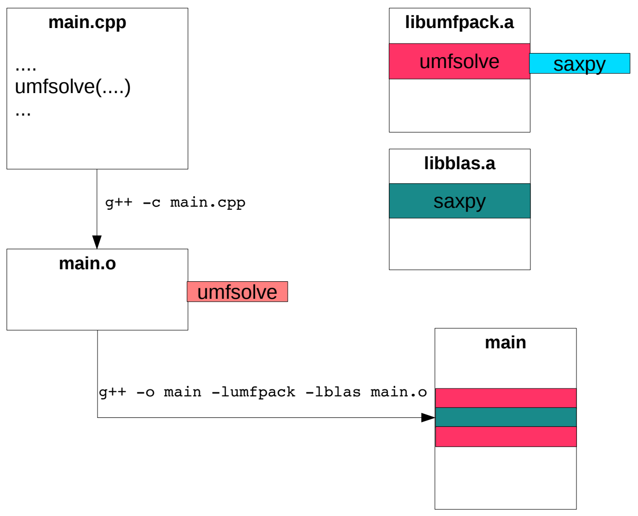


Figure 1: An example of usage of static libraries

1.2.1 Where does the linker look for static libraries?

By default, the linker uses the following rule to search for static libraries:

- it searches in all directories `DIRNAME` that have been indicated by the option `-L<DIRNAME>` placed before the occurrence of the `-l<LIB>` option;
- it searches in the system directories, typically `/usr/local/lib`, `/usr/lib` and `/lib` (in this order);
- a library may also be given by providing its full path, without using the `-l<LIB>` option. In that case the indicated file is used directly.

1.3 Shared libraries and the loader

Linking with static libraries has the advantage of producing a stand-alone executable. But there is an important disadvantage. If a new, improved release of UMFPACK is made available we need to *recompile* our code to take advantage of it (or at least redo the linking step if we have kept the object files of our code somewhere). Another disadvantage of static libraries is the duplication of machine code. All software that uses the same functionalities of UMFPACK will actually contain, replicated, the corresponding code.

The executable would be lighter in terms of size if we could keep just a copy of the common code. Think to all the applications in your PC that use graphical facilities. It would be rather inconvenient to have to recompile them every time we upgrade the graphical libraries (or impossible if the source code is not available).

There is also another important point. We may want to expand or modify the capability of an application run time, i.e. without recompiling it. This is the case of the so-called *plugins*.

A characteristic of a shared library is that it can be loaded run-time, making its symbols at disposal to the code. For example, when you add a plugin to mozilla you are in fact loading a shared library.

1.3.1 Linking against a shared library

In Unix shared libraries have extension `.so` and we will describe their naming scheme later on.

If we just need to link our code against a shared library the instructions are identical to those needed for a static library. Let's consider the command

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

and let's assume that `libumfpack.so` is present in one of the system directories (since we have not used the option `-L<DIRNAME>` here).

By default *shared libraries have precedence over static libraries*, so the linker will consider just the latter (of course this behavior may be changed by using the appropriate options as explained in a later section) It will then check whether the library `libumfpack.so` is able to resolve some of the unreferenced symbols of the `main`, program but *it will not load the corresponding machine code!*.

Please note that we have not specified `-lblas`. This should not be necessary, since the developers of the shared library `libumfpack.so` have normally linked it against `libbblas.so` (we will see this detail later on), thus the connection between libraries is implicit.

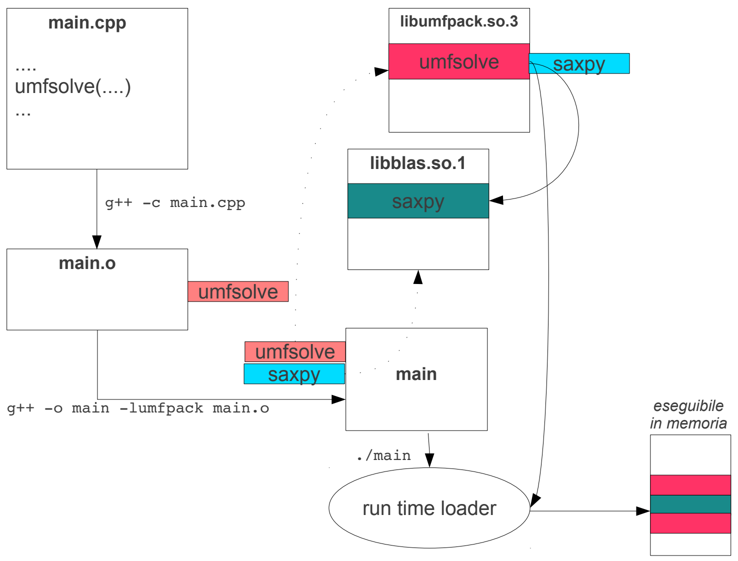


Figure 2: An example of use of shared libraries

The command `ldd main` shows the shared libraries used by the `main` program[1]. In our case we would obtain something of the sort

[1]Do not use `ldd` on unsafe programs.

```
...
libumfpack.so.5 => /lib/libumfpack.so.5.1
libblas.so.3 => /lib/libblas.so.3.2
..
```

The meaning of the numbers after `.so` are explained later on.

When we launch the executable the *loader* (called `ld.so` in Linux systems) will complete the assembly of the code by collecting the relevant parts from `/lib/libumfpack.so.5.1` and `/lib/libblas.so.3.2`. If we install a new release of UMFPACK, for instance release 5.3, the command `ldd main` will now return

```
...
libumfpack.so.5 => /lib/libumfpack.so.5.3
..
```

and the next time the program is run the loader fetches the new UMFPACK code!. No compilation is needed. Of course the new release must have a public interface that complies with the previous one (as it should!!).

1.3.2 Creating a shared library

To better understand how the loader works let's now create a shared library!. Let's consider the very simple example contained in the following c++ code [2]

```cpp
#include<iostream>
#include "smalllib.hpp"
foo::foo():a(4),b(7)
{
  std::cout<<" Building a foo object using release 0 of smalllib V. 1"<<std::endl;
}
```

with the corresponding header file

```cpp
#ifndef H_SMALLLIB_HPP
#define H_SMALLLIB_HPP
#include <iosfwd>
class foo
{
public:
  foo();
private:
  int a;
  int b;
};

struct foo2
{
  float c;
```

[2] All codes are available from the Examples/src directory of the git repository https://github.com/pacs-course/pacs.

```
  float d;
};

enum pluto {first ,second ,third};

#endif
```

that declares a class and its constructor and a few other items. We want to produce a shared library from this file. The commands are

```
g++ -Wall -fPIC -c smalllib.cpp
g++ -shared -Wl,-soname,libsmall.so.1 -o libsmall.so.1.0 smalllib.o
```

They create the shared library `libsmall.so.1` from the object files`smallib.o`. Obviously, in more realistic cases several object files contribute to form a library.

Let's have a closer look.

- The compiler option `-fPIC` indicates that the compiler should produce *position independent code*. This option in *compulsory* if you want to create an object file for a shared library. An analogous option is `-fpic`, which produces a more optimized but less portable code.

- The second command produces the shared library, whose name `libsmall.so.1.0` is given with the option `-o`.

- The most important option is `-shared` that tells `g++` (in fact the linker!) to produce a shared library and not an executable.

- The option `-Wl,soname,libsmall.so.1` is not strictly necessary, it indicates the version of the library, more precisely its *soname*. Beware, the commas are necessary, the general format is `-Wl,soname,SONAME`. The use of `SONAME` is explained in the next paragraph.

Please note that the command that creates a shared library is in many respect similar to that used to create an executable. For instance, if the code in `smalllib.cpp` requires some utilities from the UMFPACK library one would write

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

This way, the generated library `libsmall.so.1.0` will link to the UMFPACK library (either the dynamic or static version, depending on the availability) and we will not need to indicate `-lumfpack` explicitly when linking `libsmall`.

1.3.3 Versioning

We know try to explain the use of `-Wl,-soname,libsmall.so.1` and why we have called the library `libsmall.so.1.0` and not simply `libsmall.so`. When dealing with dynamic libraries the handling of the versions becomes important. Indeed, shared libraries can be changed "on the fly" and we need to be sure that our program uses the correct version. Of course versioning is important for libraries that are distributed to the large public. For library of local use one can avoid it!.

A shared library (we are referring to Unix-type system) is characterised by different names:

- The *generic name*, also called *linking name*. It has the form `libNAMEso`. It is the name the linker searches when using the `-l<NAME>` option. If we are not implementing version control, this is the only name to use.

- The *soname* (shared object name), which is usually obtained by adding `.Version` to the generic name. (in our example it would be `libsmall.so.1`). `Version` is (typically) a integer number that indicates the version of the library.

- A version may have different releases. The difference between versions and releases is that the releases of a given version should have a *compatible public interface*. This is not guaranteed among different versions. Therefore, we need to indicate the name of the file that contains the current release. It is called *real name*, since it is usually the name of the actual file storing the library (the others are usually just symbolic links, as we will see). It has the form `libNAME.so.Version.Release`. Here `Release` is an integer, or sometimes a couple of integers, that identify the release number and the bugfix number.

This naming scheme allows to keep in the same machines different versions of the same library.

The option `-l<LIB>` tells the linker to use the library named `libLIB.so`, but if this library has been assembled with the option `-Wl,soname,SONAME` the *loader will look for the file called SONAME when loading the library.*

Indeed the option `-Wl,-soname,libsmall.so.1` inserts in the library file a special symbol that refers to the soname, in this case `libsmall.so.1`. To make all the names available we do

```
ln -s libsmall.so.1.0 libsmall.so.1   symbolic links al soname
ln -s libsmall.so.1 libsmall.so       symbolic links al linking name
```

creating two symbolic links to the file with the *real name*. In this case we have only a version so all names point to the same real name. At this point we can create an executable that uses the library.

We consider this simple program:

```cpp
#include <iostream>
#include <cstdlib>
#include "smalllib.hpp"
using namespace std;
int main()
{
  foo one;
}
```

that creates an object named `foo` of the class type defined in the library. We compile it with the command

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

where With `-L.` we indicate that the libraries must first be searched in the current directory, `-lsmall` gives the generic name to the linker, which then looks for `./libsmall.so` to check whether the symbols are resolved.

When we launch `ldd main` we get (among other things)

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

This result tells us two things:

1. the loader is looking for `libsmall.so.1` and not just `libsmall.so`. This is because we have indicated the soname `libsmall.so.1` when creating the library;

2. the loader is not finding the library file! Indeed *the places where the loader looks for library files is not necessarily the same used by the linker when creating the executable.* In other words, the option `-L` is an option of the linker, it is *not* passed to the loader (and for very good reasons!).

1.3.4 The rules followed by the loader to search libraries

The loader follows these rules to search for shared libraries.

1. If the option `-Wl,-rpath=PATH` is given at the linking stage of our program, where `PATH` is a list of directories separated by `:`, then the loader will search the shared library in those directories first. This option is often used during the *development phase* of a program to address the loader to the directories where the libraries that are being developed are kept.

   ```
   g++ myfile.o -LMyDir -lmylyb -Wl,-rpath=MyDir:/usr/local/mylibs/lib \
      -o myfile
   ```

2. It then looks at the directories possibly contained in the *environment variable* `LD_LIBRARY_PATH`, again separated by columns (`:`)

   ```
   export LD_LIBRARY_PATH=/home/myuser/lib:/home/myfriend/lib
   ```

   If you want to **add** the path do:

   ```
   export LD_LIBRARY_PATH+=:/home/myuser/lib:/home/myfriend/lib
   ```

   or

   ```
   export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/home/myuser/lib:/home/myfriend/lib
   ```

3. It looks in the list contained in the cache file `/etc/ld.so.cache`, which is generated using the command `ldconfig` (you need to be superuser). In turn, `ldconfig` generates the cache file by analyzing the content of the file `/etc/ld.so.conf` and/or of the files contained in the directory `/etc/ld.so.conf.d` (the details may depend on the Linux/Unix distribution used). To make any changes in those files active you must then launch `ldconfig`. For example to add the directory `/home/me/lib` "permanently" to the search path one may add a file in `/etc/ld.so.conf.d` containing the name of the directory:

   ```
   sudo echo ‘‘/home/me/lib’’ > /etc/ld.so.conf.d/personal_lib
   sudo ldconfig
   ```

Note that `ldconfig` is launched at every reboot, regenerating the cache file.

4. Finally, the loader searches the system directories `/usr/lib` and `/lib`.

Note: All methods, a part the first two, require superuser privilege. Moreover, the first two methods should be used in the development phase.

As an alternative, when the shared libraries for a specific program are kept in unusual directories it is common to launch the program through a *shell script* that sets up the search path:

```
#!/bin/bash
export LD_LIBRARY_PATH=/usr/local/myprogs/lib:/opt/umfpack/lib
./mycode $*
```

This technique is used by several programs, for instance MATLAB.

Now, if in our example we type

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

then `ldd main` returns

```
libsmall.so.1=> /MYDIRECTORY/libsmall.so.1
```

where `MYDIRECTORY` is the directory where the library is stored. If I launch the program I obtain

```
>./main
Building a foo object using release 0 of smalllib V. 1
```

1.3.5 Installing a new revision

Let's consider now the following variant

```
#include<iostream>
#include "smalllib.hpp"
foo::foo():a(4),b(7)
{
  std::cout<<" Building a foo object using release 1 of smalllib V. 1"<<std::endl;
}
```

with which we create a new release of the library

```
g++ -shared -Wl,-soname,libsmall.so.1 -o libsmall.so.1.1 smalllib2.o
rm libsmall.so.1
ln -s libsmall.so.1.1  libsmall.so.1
```

Now version 1 is associated to release 1.1. The old file `libsmall.so.1.0` can be eliminated. The symbolic links are now

```
libsmall.so -> libsmall.so.1
libsmall.so.1 -> libsmall.so.1.1
```

and if I launch the program I have

```
>./main
Building a foo object using release 1 of smalllib V. 1
```

I get the new result. No recompilation needed!

Clearly, this example is of no practical use. Yet it describes the main features of a shared library and its basic usage.

1.3.6 Advantages and disadvantages of shared libraries

We have already described some advantages of shared libraries vs static ones. There are also a few disadvantages as well.

1. The program now relies on the existence of the library. If by mistake the library is eliminated or becomes inaccessible the program does not run anymore.
2. The handling of shared libraries is more complicated, an incorrect installation may cause program failure or incorrect behavior.
3. The system to handle shared libraries is more complex and may cause some overheads, that's the reason why operative systems for high performance computing sometimes do not support shared libraries.

We want to point out again an important advantage of shared libraries. The fact that a shared library may be itself linked to the another shared libraries on which it depends, thus reducing the number of libraries to be indicated in the linking stage of our program. For instance, UMFPACK depends on the BLAS, thus the developer of the UMFPACK library may use a command of the sort

```
g++ <files *.o of UMFPACK> <other options> -lblas -shared -o libumfpack.xx.so
```

If the BLAS are present as static library the corresponding code will be integrate in that of `libumfpack.xx.so`, otherwise the same rules that govern the handling of shared libraries for an executable are applied in this case. Indeed, in my computer

```
ldd /usr/lib/libumfpack.so | grep libblas
```

returns

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

that indicates that the umfpack library loads the shared library `libblas.so.3gf`. Here `gf` indicates a particular version of the BLAS that exploits the presence of graphic hardware.

The big advantage is that I need to use only `-lumfpack` when linking my program. I may even not be aware of the fact that UMFPACK needs the BLAS (but I need the have the BLAS installed, otherwise the program won't compile).

1.4 The options `-static` and `-dynamic`

The linker by default prefers shared libraries. Yet the behavior may be changed using the `-static` and `-dynamic` options. Let suppose we have both static and shared libraries available. the command

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

will link against `libumfpack.a` and `libblas.so`.

Beware however that your program will usually link also several system libraries (the one providing the c++ standard library, for instance). To use the option `-static` you must have also those libraries as static libraries!

2 Dynamic loading

Another important characteristics of shared libraries is that it is possible to implement *dynamic loading* of the contained symbols. We will describe this feature first by looking at the loading of functions, then on how to load more complex objects dynamically. We will refer to a set of class that implement numerical quadrature rules. The objectives are

1. To allow the user to change the integrand functions, with no recompilation of the main program;
2. Be able to define the quadrature rule at run time, by specifying its name in a file;
3. Be able to add new quadrature rules with no need of recompiling the code.

So let's assume that we have a set of classes that implement different quadrature rules on a given function $f : I \subset \mathbb{R} \rightarrow \mathbb{R}$.

We want to be able to choose the integrand function run-time, by indicating its name. We can operate in two ways, either through an object factory or, in this case, via a direct loading from a shared library. We will first examine the latter technique. We will create one (or more) shared libraries with all the possible integrands of interest, we will load the library we need, as well as the symbol corresponding to the integrand we wish to use.

The mechanism for dynamic loading of shared libraries in Linux (and in general in Unix-type) system is based on set of C tools that are available by including the header file `dlfcn.h` and linking the executable with the system library `libdl.so`. The main functions at disposal are `dlopen`, `dlsym`, `dlclose` and `dlerror`.

A problem with this technique in C++ is due to *name mangling*. In C++ the identifier of a function is *not* its name but its *signature*. To distinguish two functions with the same name but different signatures the name is "mangled" by adding some extra characters that encode the type of arguments. Since (unfortunately) the mangling algorithm is not standard if we want to load functions from a library correctly, we need to make sure that name mangling is NOT activated.

This can be achieved by *C-type linking*, using the `extern "C"` directive. Of course at this point we will not be able to apply function oferloading to the integrand functions, yet this is not a limitation since the number and type of arguments of the integrand function are fixed.

The file `integrands.hpp` contains an example of the declaration of possible integrands.

```
#ifndef H_INTEGRANDS_HPP
#define H_INTEGRANDS_HPP
#include <cmath>
/*! \file integrands.hpp
  \brief contains the integrand funcctions
 */
```

```
extern "C" {
  double fsincos(double const &);
  double square(double const &);
  double pippo(double const &);
}
#endif
```

The definition is in the corresponding `cpp` file.

```
#include "integrands.hpp"

extern "C"{

double fsincos(double const & x)
{
  using namespace std;
  return sin(x)*cos(x);
};

double square(double const& x)
{
  using namespace std;
  return x*x;
};

  double pippo(double const & x)
  {
  using namespace std;
  return x*std::exp(x);
  }


}
```

. We then create a shared library

```
g++ -fPIC -c integrands.cpp
g++ -shared -Wl,-soname,libintegrands.so \
               -o libintegrands.so integrands.o
```

Of course, we may also use a single command:

```
g++ -fPIC  integrands.cpp -shared -Wl,-soname,libintegrands.so \
 -o libintegrands.so
```

Since there is no need of implementing version control we have used the generic name also as soname and indeed the option `-Wl,-soname,libintegrands.so` is here useless and can be omitted.

the command `nm libintegrands.so` returns, among other things,

```
00000000000006e7 T fsincos
0000000000000664 T square
```

showing that, thanks to the C linkage, the symbols corresponding to our function coincide with the function name.

We now want to be able to give to the program the name of the library containing the integrands, as well as the desired integrand function. We can use the tool GetPot to parse this information from a file. The name of the integrand will be given by `integrand=NAME`, while the name of the library by `library=NAME`. Dynamically loaded libraries are searched using the same rules adopted by the loader. *However, if the library name contains the path, i.e. `NAME` contains a `/`, the loader will use that name directly!*.

Let's look at how the main program in Material/Examples/src/QuadratureRule/DynamicIntegrands/main_integration.cpp handle the loading of the library.

Note that to compile the main *we do not need* to indicate `-lintegrand`, since the library is loaded dynamically, nor it is necessary to include `integrands.hpp`.

The principal instructions are

```
void readParameters(const int argc, char** argv,
                    double &a, double &b, int & nint,
                    std::string & library,std::string & integrand,
                    std::ostream & out){
  GetPot cl(argc, argv);
  const double pi=2*std::atan(1.0);
  if(cl.search(2, "--help", "-h")){
    out<<"Compute integral from a to b with nint intervals"<<"\n";
    out<<"Possible arguments:"<<"\n";
    out<<"a=Value  (default 0)"<<"\n";
    out<<"b=Value (default pi)"<<"\n";
    out<<"nint=Value (default 10)"<<"\n";
    out<<"library=string (default libintegrands.so)"<<"\n";
    out<<"integrand=string (default fsincos)"<<"\n";
    out<<"-h or --help : this help"<<"\n";
    std::exit(1);
  }
  a=cl("a", 0.);
  b=cl("b", 1*pi);
  nint=cl("nint", 10);
  library=cl("library", "./libintegrands.so");
  integrand=cl("integrand", "fsincos");
}
```

This is an helper function that processes the options passed to the program and in particular, stores the names of the library and of the integrand into two strings.

Then we have

```
...
#include <dlfcn.h>
...
int main(int argc, char** argv){
```

```
...
readParameters(argc,argv, a, b,nint,library,integrand,cout);
...

void * lib_handle;
lib_handle=dlopen(library.c_str(),RTLD_LAZY);
```

`dlopen` is the command the opens the library and it returns a `void *`. If the library does not exist or cannot be loaded the returned pointer is null. You may test with `lib_handle==nullptr` ore `lib_handle==0`. The macro `RTLD_LAZY` (defined in `dlfcn.h`) is used to indicate that the referenced symbols in the library should be loaded only at the moment they are actually used in the code (lazy loading). We will indicate later on other possibilities, for instance how to load all symbols when the library is opened.

The command `dlsym` can now be used to load specific symbols. `FunPoint` is defined in the file Material/Examples/src/QuadratureRule.hpp as a function wrapper (you can use a pointer to function if you do not want to use C++11 syntax):

```
  typedef std::function<double (double const &)> FunPoint;
```

The loading of the integrand is then simply:

```
void * ip=dlsym(lib_handle,integrand.c_str());
if(ip==0 || (error = dlerror()) != 0 ){
    std::cerr<<"Error, invalid integrand "<< error<<std::endl;
    std::exit(2);
  }
double (*fp) (double const &); // pointer to function
fp  = reinterpret_cast<double (*) (double const &)>(ip);
FunPoint f(fp);
```

The function `dlsym` takes a C-style string (a `char *`) containing the name of the symbol and it returns a `void *` to the corresponding object. Again the pointer is null if the symbol is not present or if another error has occurred. The function `dlerror()` may also be used to test for errors (as it is indeed the case in the example) It returns a `char *` which is null in case of no error, or is a string containing a error message.

The piece of code

```
fp  = reinterpret_cast<double (*) (double const &)>(ip);
```

is horrible but is inevitable. We need to recast the void pointer to the correct type (C++ is a strongly typed language!) and the only possibility is here a reinterpreted cast. Therefore, it is of *fundamental importance* to do the casting to the correct type: a segmentation fault is almost guaranteed otherwise.

One may also use a C-style cast:

```
fp  = (double (*) (double const &)) (ip);
```

which gives the same result.

A technique that guarantees a better control on the types is that of the object factory, we will see it later on. We recall that to use the dynamic loading facility we need to add `-ldl` at the linking stage, in order to link the library `libldl.so` that contains `dlopen` etc.:

```
g++ main_integration.o  <other options> -ldl -o main_integration
```

We recall the rules followed by `dlopen` to search for dynamically loaded libraries:

- If the name of the library is provided without giving the path, i.e. just the name is given, then `dlopen` follows the same rules of the loader. In particular, you may use the environment variable `LD_LIBRARY_PATH`, the liner option `-Wl,-rpath=` etc.
- If the path is given, i.e. the name contains a "`/`", that path is used. The path may be relative or absolute.

The command

```
main_integration library=./libintegrands.so
```

will then use the library `libintegrands.so` in the current directory. **We can now change the integrand without recompiling our code.**

2.1 Setting the quadrature rule run-time: an object factory

The quadrature rule is an object of class type, more precisely of a class derived from `QuadratureRule`, and not a function. The technique illustrated in the previous section cannot (in general) be used in this case.

We want to resort to an *object factory*. An object factory is an object whose role is to create polymorphic objects according to an input, which may be a string, an integer any other identifier associated to the object we want to create.

We with then to create a factory of objects of type `QuadratureRule`, which is the base class of our quadrature rules. We need to define the *builder*, that is the component of the factory that actually creates the object of the correct derived type.

In the simplest case a builder is just function that returns a `QuadratureRule*` or, if we want to limit the risk of memory leaks, a `std::unique_ptr<QuadratureRule>`, for instance

```
std::unique_ptr<QuadratureRule> simpsonBuilder()
{
  return std::unique_ptr<QuadratureRule>(new Simpson);
}
```

The easiest way (but not the most effective one) to implement an object factory is to use a map

```
std::unique_ptr<QuadratureRule> (*Builder)();
std::map<std::string,Builder> theFactory;
theFactory["Simpson"]=&simpsonBuilder;
theFactory["Gauss3pt"]=&gauss3ptBuilder;
...
```

Builder è un puntatore ad una funzione che non riceve nulla in ingresso e restituisce unique_ptr<QuadratureRule>

To use the quadrature rule corresponding to the `std::string ruleName` we write

```
auto myRule = theFactory[ruleName]();
```

Yet normally a Factory is implemented as a class. And normally as a *singleton*, i.e. a class of which *at most one object may exist during program execution.* Therefore, before coming back to a more complete implementation of an object factory we will digress on Singletons.

3 Singletons

In some cases one may want to ensure that there is only at most one object of a class available to the program. Typically this happens when some data or functionalities should be centralized. For instance, we need to communicate to a data base or with a serial port. Object factories are also often implemented as singletons.

It is well known that if we declare the constructors, copy constructors and copy-assignment operators private (ore we "delete" them in C++11) we have a class whose object cannot be created or reassigned by functions other that *methods of the class*.

A possibility of creating a singleton is then to make constructors, copy constructors and copy-assignment operators private (or deleted) and exploit the fact that a *static variable of a function* is *initialized* the first time the function is called and then kept in memory for successive usage. Exactly what we want! And if the function is a member of the class it can construct the object even if the constructor is private! This technique is not the only one to create a singleton, but it is particularly simple and it is called *Meyers' trick*[6] from the name of the ideator.

Meyer's idea is then to have an instance of the singleton as a static variable of a *static method* of the singleton class itself! We need a static method because we need to call it without the object (since it is the method that creates the singleton object!).

We will colla the metro `Instance()` since it creates the only instance of the singleton object.

In conclusion, a singleton class has the following structure

Listing 1: Example of a singleton

```
class Singleton{
public:
  static Singleton & Instance();
  ...
private:
  Singleton()=default;
  Singleton(Singleton const &)=delete;
  Singleton & operator =(Singleton const &)=delete;
  ...
}
```

Listing 2: Definizion of Singleton::Instance()

```
Singleton & Singleton::Instance(){
  static Singleton theSingleton;
  return theSingleton;
}
```

Posso ritornare una reference perchè la variabile restituita è static

Essendo la variabile static, viene creata solo la prima volta che chiamo Instance, se chiamassi Instance più volte mi restituirebbe sempre la stessa variabile (indirizzo di memoria)

In a program that uses our singleton class we have

Listing 3: Usage of a singleton

```
#include ``Singleton.hpp''
...
Singleton & a(Singleton::Instance()); // Creates the only
              //intance of a Singleton object
...
```

```
Singleton & b(Singleton::Instance())// b in a reference
             //to the rame object as a!!
```

Note that we can access the singleton only through the static method Instance(). There is no way of creating a `Singleton` object otherwise.

Variables `a` and `b` are indeed references to the same object, the one contained in the static variable `theSingleton`.

Other implementations are possible, you may consult [1, Chapter 6] for a detailed discussion.

3.1 A problem with this implementation in C++98

If you use a C++98 compiler Mayer's trick is not thread safe. It means that if you work in a multi-threaded environment, like `OpenMP`, there is no guarantee that an concurrent invocation of Instance() in two (or more) threads will not cause the creation of multiple instances of the Singleton.

To avoid this you should use different implementations, like the one described in the cited reference, where other, more technical (and in most cases not relevant) issues are addressed as well.

We just give a sketch of a possible thread-safe implementation using `OpenMP`.

```
class Singleton{
public:
  static Singleton & Instance();
  ...
private:
  Singleton(){};
  Singleton(Singleton const &);
  Singleton & operator =(Singleton const &);
  static volatile Singleton* pInstance_;
};
Singleton * Singleton::pInstance_=0;
```

Listing 4: Definition of Instance() (OpenMP)

```
Singleton & Singleton::Instance(){
#pragma omp critical{
 if (!Singleton::pInstance_)
  Singleton::pInstance_= new Singleton;
}
return *Singleton::pInstance_;
}
```

This solution works since eventually only one thread will call the **new** statement. However it is not efficient because the critical section is entered also when the singleton has been created. A better solution implements the *double checked pattern*:

Listing 5: Definition of Instance() (double-checked)

```
Singleton & Singleton::Instance(){
if (Singleton::pInstance_){
```

```
#pragma omp critical{
  if (!Singleton::pInstance_)
   Singleton::pInstance_= new Singleton;
}
}
return *Singleton::pInstance_;
}
```

This way if pInstance_ is not null the threads do not enter the critical section (I recall that critical sections are rather heavy constructs). We have declared pInstance_ as **volatile** to avoid problems linked to optimization of memory by some compilers. More discussion on [1].

IN C++11 the issue has been solved. The standard impose that the initialization of static function variables be an atomic operation, thus thread-safe

4 Back to the factory

We will then implement out factory as a singleton class. We will use, as seen in Section 2.1 a std:map! (we could use and unordered_map in C++11) to associate the builder with a string.

In general I need

- A hierarchy of objects derived from an abstract class that I indicate with `AbstractProduct`. In our case the `AbstractProduct` is the `QuadratureRule` class.
- A builder for each concrete object (`ConcreteProduct`) of the hierarchy. The builders will be usually (but not necessarily) stored in a *shared library*, so it is simpler to add more concrete products and possibly load then dynamically. Our concrete products are the concrete quadrature rule classes (`Simpson` etc.). The builder, as we have seen, may be just a function returning a unique pointer to a newly created object. Yet it may be convenient to have a specially devised class, as we will see later on.
- I need to find a way to *register* the concrete products into the factory, i.e. to fill the map. The idea is to have something as extensible as possible, so that I can register new concrete products without having to recompile my program. A possibility is to use *the same class used for the builder to take care also of the registration.* Again the key to flexibility is to use a shared library.

4.1 A Generic object factory

A generic object factory may be found in
Material/Examples/src/AbstractFactory/Factory.hpp

Let's have a look to the main ingredients

```
template
<
  typename AbstractProduct,
  typename Identifier,
  typename Builder=std::function<std::unique_ptr<AbstractProduct> ()>
>
class Factory{...
```

```
 private:
  typedef std::map<Identifier,Builder> Container_type;
  Factory()=default;
  Factory(Factory const &)=delete;
  Factory & operator =(Factory const &)=delete;
  Container_type _storage;
  };
```

Is a template with three arguments: the AbstractProduct, the Identifier (the type used to address the concrete products) and the Builder.

Among the private members we have the map that associates the `Identifier` with the `Builder` of the corresponding `Concrete Product`. *To create a Singleton I have here used the C++11 syntax*, otherwise I could have put

```
private:
 Factory(){}; // default
 Factory(Factory const &);
 Factory & operator =(Factory  const &);
```

since by writing just the declaration of copy-constructor and assignment I am blocking the possibility of using them.

The main methods are: public methods

```
static
Factory<AbstractProduct,Identifier,Builder> &
  Factory<AbstractProduct,Identifier,Builder>::Instance() {
    static Factory theFactory;
    return theFactory;
  }
```

I have used Meyer's trick explained in Sec. 3

The registration in the factory is made by the method add():

```
  void
  Factory<AbstractProduct,Identifier,Builder>::add(Identifier const &
  Builder_type const & func){
    auto f =
      _storage.insert(std::make_pair(name, func));
   if (f.second == false)
      throw std::invalid_argument("Double␣registration␣in␣Factory");
}
```

To create a concrete product:

```
  std::unique_ptr<AbstractProduct>
  Factory<AbstractProduct,Identifier,Builder>::create(Identifier const & name)
    const {
    auto f = _storage.find(name);
    return (f == _storage.end()) ? std::unique_ptr<AbstractProduct>():
      std::unique_ptr<AbstractProduct>(f->second());
  }
```

It returns a smart pointer to the desired object [3] If the identifier does not correspond to a concrete product registered in the factory we return the null pointer (the default constructor of unique_prt creates a null pointer). Alternatively, we could have launched an exception.

4.2 Registering a QuadratureRule

Now we can specialize our factory for our `QuadratureRule` classes.

```
typedef GenericFactory::Factory<NumericalIntegration::QuadratureRule,std::string>
 RulesFactory;
```

Since the structure of a builder for a quadrature rule is always the same we can use a function template.

Listing 6: A template for the builder of quadrature rules

```
#include "numerical_rule.hpp"
typedef std::unique_ptr<QuadratureRule> QuadRuleHandler;
template<class T>
QuadRuleHandler Build(){
return  QuadRuleHandler(new T);
}
};
```

Alternatively I can use functors

How to insert a new rule? A possibility is to use a function:

```
#include "Factory.hpp"
#include "QuadRuleBuilder.hpp"
typedef GenericFactory::Factory<NumericalIntegration::QuadratureRule,std::string>
 RulesFactory;

void registerRules{
 RulesFactory & factory = RulesFactory::Instance();
 factory.add("Simpson",&Build<Simpson>);
 factory.add("Trapezoidal",&Build<Trapezoidal>);
 ...
}
```

and in the main() call registerRules().

Listing 7: A template for the builder of quadrature rules (functor version)

```
...
template<class T>
struct QuadBuilder{
QuadRuleHandler operator(){
return  QuadRuleHandler(new T);}
};
```

And then the registration becomes

[3] We are using C++11 syntax but the equivalent C++98 constructs is obvious

```
...

void registerRules{
 RulesFactory & factory = RulesFactory::Instance();
 factory.add("Simpson",QuadBuilder<Simpson>());
 factory.add("Trapezoidal",QuadBuilder<Trapezoidal>());
 ...
}
```

In this case the builder is a function object that is then wrapped into std::function. *If you do not use C++11 you may use boost::function instead, or design your own function wrapper.*

4.2.1 A Proxy class for registering rules

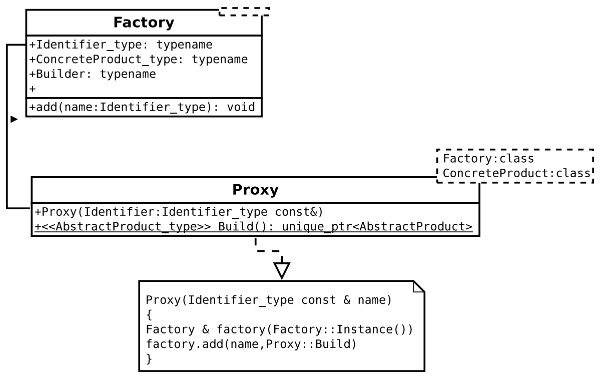
The previous method has a some drawbacks: It requires to add a line to registerRules() for every new rule. And the registering of the rules must be concentrated in a single file. Moreover, I need to call registerRules() in the main().

It would be great if I could

1. be able to distribute the registration of concrete products in different files. This way we can add new rules without having to recompile the other files;
2. be able to register the rules in the main just by loading a shared library, without the need of calling a function in the main. This way I effectively implement a simple plugin mechanism.

The second point could be obtained by using the `__attribute__((constructor))` attribute for the registerRules() function. This attribute orders the execution of the function before the first instruction of the main is executed. But it is not portable. Moreover, by using functions to register products in the factory it is more complex to satisfy the first requirement.

The idea is then to have a particular template class that registers a product whenever I create a object of the class. Let's see how it can be made by looking at the example in Material/Examples/src//AbstractFactory/Proxy.hpp.


Factory
+Identifier_type: typename
+ConcreteProduct_type: typename
+Builder: typename
+
+add(name:Identifier_type): void
Factory:class
ConcreteProduct:class
Proxy
+Proxy(Identifier:Identifier_type const&)
+<<AbstractProduct_type>> Build(): unique_ptr<AbstractProduct>

```
Proxy(Identifier_type const & name)
{
Factory & factory(Factory::Instance())
factory.add(name,Proxy::Build)
}
```

Here the main statements:

```
template
<typename Factory, typename ConcreteProduct>
```

```
  class Proxy {
  public:
    typedef typename  Factory::AbstractProduct_type AbstractProduct_type;
    typedef typename  Factory::Identifier_type Identifier_type;
    typedef typename  Factory::Builder_type Builder_type;
    typedef           Factory Factory_type;
    // The builder
    Proxy(Identifier_type const &);
    static std::unique_ptr<AbstractProduct_type> Build(){
      return std::unique_ptr<AbstractProduct_type>(new ConcreteProduct());
    }

  private:
    Proxy(Proxy const &)=delete; // only C++11
    Proxy & operator=(Proxy const &)=delete; // only C++11
  };
```

The essence of this template is all in the constructor:

```
  template<typename F, typename C>
  Proxy<F,C>::Proxy(Identifier_type const & name) {
    // get the factory. First time creates it.
    Factory_type & factory(Factory_type::Instance());
    // Insert the builder. The & is not needed.
    factory.add(name,&Proxy<F,C>::Build);
  }
```

The creation of a Proxy<F,C> object register in the factory `F` the builder for the concrete product `C`, here defined as static member of the class, but I could have implemented it as a call operator (**operator**()()) and stored an object of Proxy<F,C> type, as illustrated before in Sect.4.2

Thus in Material/Examples/src/QuadratureRule/AllDynamic/ruleProxy.hpp I can specialize the factory and the proxy:

```
#include <string>
#include "QuadratureRule.hpp"
#include "Proxy.hpp"
#include "Factory.hpp"
//!Specialization of Factory and Proxy for QuadratureRules
namespace QuadratureRuleFactory
{
  typedef
      GenericFactory::Factory<NumericalIntegration::QuadratureRule,std::string>
                                                          RulesFactory;
  //! Only C++11
  template <typename ConcreteRule>
      using RuleProxy=GenericFactory::Proxy<RulesFactory,ConcreteRule>;
}
```

While, in Material/Examples/src/QuadratureRule/AllDynamic/myRules.cpp we use the `Proxy` to register the rules:

```
#include "numerical_rule.hpp"
#include "ruleProxy.hpp"
// Registration in the factory I use an unnamed namespace
namespace
{
  using namespace NumericalIntegration;
  using QuadratureRuleFactory::RuleProxy;

  RuleProxy<Simpson> SH("Simpson");
  RuleProxy<Trapezoidal> TH("Trapezoidal");
  RuleProxy<MidPoint> MH("MidPoint");
}
```

In questo source file sto riempendo la factory creando degli oggetti Proxy che poi non utilizzerò più, non mi serve che siano visibili anche dall'esterno (ossia dalle altre compilation unit) per questo gli inserisco all'interno di un unnamed namespace
--> Ciò che mi rimane (e mi serve) è la factory, il resto (cioè i proxy) poi li butto

I have used and anonymous namespace because the object have been created only with the objective of registering products in the factory. No need of them anywhere else. Now I need to create a dynamic library:

```
g++ -std=c++1 -fPIC myRules.cpp
g++ -shared myRules.o -o libmyrules.so
```

Loading the library register the rules. I can use the option `-lmyrules` or load the library dynamically with dlopen().

Important note: Since the library with the rules will be leaded dynamically, I need to use the linker option `-Wl,-E` or `-Wl,-export-dynamic` when creating the executable. This the linker adds all symbols to the dynamic symbol table. The dynamic symbol table is the set of symbols which are visible from dynamic objects at run time. If you do not use either of this option (or use the –no-export-dynamic option to restore the default behavior), the dynamic symbol table will normally contain only those symbols which are referenced by some dynamic object mentioned in the link. And since the `RuleProxy` objects are not references in the main, they won't be loaded and thus created. So the rules won't be stored in the factory.

4.3 The main()

Let's have a look to the fundamental statements of Material/Examples/src/QuadratureRule/All-Dynamic/main_integration.cpp. We use `GetPot` to get some options:

```
  GetPot    cl("quadratura.getpot");
  string quadlib=cl("library","libmyrules.so");
  void * dylib=dlopen(quadlib.c_str(),RTLD_NOW);
```

The macro `RTLD_NOW` is here preferable to `RTLD_LAZY` since it will resolve all symbols in the library.

We can now access the factory in the usual way

```
  RuleFactory & rulesFactory( RuleFactory::Instance());
```

The desired rule can be read from file:

```
string rule=cl("rule","Simpson");
QuadratureRuleHandler theRule=rulesFactory.create(rule);
```

At this point we should have a in `QuadratureRuleHandler` a `unique_ptr` to a specific quadrature rule.

A last point concern the compilation of the example. All files concerning the base classes and general methods produce a library stored in the subdirectory `lib.` of `Examples`. More precisely `libquadrules.so` contains the base quadrature rule class while the classes for composite integration are in `libnumint.so`.

4.3.1 Compilation

The command `make dynamic` compiles the libraries for registering the rules and the integrand functions. The only difference with respect to the example is that the handling of the integrands is delegated to an object of the class `udfHandler`

The main commands launched by `make`:

```
clang++ -std=c++11 -fPIC -I. -I<ROOTINCLUDE> -o myRules.o myRules.cpp
clang++ -shared -g -O0  myRules.o -L<ROOTLIB> -lquadrules -o libmyrules.so
clang++ -std=c++11 -fPIC -I. -I<ROOTINCLUDE> -c -o udf.o udf.cpp
clang++ -shared  udf.o -o libudf.so
```

Here `<ROOTINCLUDE>` and `<ROOTLIB>` contain the name of the directories where we have the include files and the libraries installed by the command `make install`, respectively, launched in the root directory of the Examples.

The executable is generated by `make exec`. The file `udfHandler.cpp` contains the utility to access the library with the integrands. It is compiled with the main (a cleaner organization would have used another library...).

```
clang++ -std=c++11 -I. -I<ROOTINCLUDE> -c main_integration.cpp
clang++ -std=c++11 -I. -I<ROOTINCLUDE> -c udfHandler.cpp
clang++ main_integration.o udfHandler.o -Wl,-E  -L<ROOTLIB>
 -lnumint -lMesh1D -ldl -o main_integration
```

Here `-lnumint` loads the library for the composite quadrature and `-ldl` is required for the dynamic loading. Alternatively, if I do not want dynamic loading I could have added `-lmyrules` (of course in this case I cannot decide what to lead at run time).

References

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ in depth. Addison-Wesley, 2001.

[2] Boost. The Boost c++ library. http://www.boost.org, 2013.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[4] M.S. Joshi. *C++ Design Patterns and Derivatives Pricing*. Cambridge University Press, second edition, 2008.

[5] N.M. Josuttis. *The C++ standard library: a tutorial and reference.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] S. Meyers. *More effective C++.* Addison-Wesley, 1996.

[7] J. Norton. Dynamic class loading in C++. *Linux J.*, 73(38):1–6, 2000.

[8] D. Vandevoorde and N.M Josuttis. *C++ templates: the Complete Guide.* Addison-Wesley, 2006.

[9] D.A Weeler. Program library HOWTO, 2009. In particular Section 3 on shared libraries.

[10] YoLinux. Static, shared dynamic and loadable linux libraries. YoLinux.com Tutorials, 2009.