Creating and using shared libraries with different compilers on different operating systems

December 06, 2014 *topic: Libraries* *tagged: C++ · CMake · Linux · Windows*

If you want to make your C/C++ project portable between operating systems and compilers, there are some things to consider. In this article I focus on shared libraries and give a practical overview on how to create and use shared libraries with various compilers on various operating systems, with the goal of portable source code and a unified build process.

> *"A shared library or shared object is a file that is shared by executable files and further shared objects files." A shared library on Linux is called "dynamically linked shared object", and has the file extension* ***.so****. The windows equivalent is the "dynamic link library" usually with file extension* ***.dll****.*[1]

This article is structured as follows:

At first we create and use a minimal shared library on Linux with GCC. We take a look at the necessary gcc options for compiling and linking and check at the exported symbols of the .so file with the tool "nm".

Then we build the same code on Windows with MinGW. At first by directly linking the *dynamic link library*. Then, we also build and use a so-called *import library* - we have to consider some platform/compiler specifics at this point. I give a brief explanation what an *import library* is and when it 's needed.

Topics:
debugging+profiling
design+architecture
libraries
methods+principles
musing
software+tools

Tags:
boost c++ cmake
complexity
dependencies distcc
docker gpferftools
gprof linux python
security valgrind
windows

We go on by building our minimal demo application with other compilers, like the *Microsoft Visual C++ Compiler (MSVC)* and clang/LLVM. To make the build process more convenient and more flexible, we use CMake at this point.

Last but not least, a few words about the interoperability of libraries created with different compilers (ABI compatibility) follow.

The final version of the minimalistic portable code demo, that is gradually developed in this post, is available on github.

Shared libraries with Linux/GCC

Let's start building our demo on Linux with GCC. The header of our shared library is **shared.h** and its implementation is in **shared.cpp**. Our application is implemented in **main.cpp**, where we want to use our shared library:

`shared.h:`

```
#ifndef SHARED_H__
#define SHARED_H__

void f();

class X {
public:
  X();
  void mX();
};

#endif
```

`shared.cpp:`

```
#include "shared.h"
#include <iostream>

void f() {
  std::cout << "f()\n";
}

X::X() {
  std::cout << "X::X()\n";
}

void X::mX() {
  std::cout << "X::mX()\n";
}
```

Now let's compile our library with GCC:

```
g++ -fPIC -shared shared.cpp -o libshared.so
```

The command above compiles *shared.cpp* to a shared library (*-shared*) with position independent code (*-fPIC*). The compiler output (*-o*) is written to *libshared.so*. Let's take a quick look external symbols of our library. We can use "nm" for this purpose:

```
nm -gC libshared.so
```

The option -g tells nm that we are just interested in external symbols. With -C the low-level symbol names are translated to human readable symbols and we can directly see the names of our methods/functions the output (on Linux/GCC, all symbols are exported by default - this is not necessarily the case with oder compilers):

```
...
00000000000008e4 T f()
00000000000008c0 T X::X()
...
```

Fine, our library exports the symbols `f()` and `X::X()` - ready to be used by some application -> `main.cpp:`

```
#include "shared.h"

int main() {
  f();
  X x;
  x.mX();
}
```

We compile *main.cpp* and link our shared library with:

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

The option `-L.` adds the current directory "." to the linker search path – without this the linker will not find our library. The generated executable is "main".

Before we execute our program, let's check what libraries it is linked with. We can do this with the command `ldd`:

```
ldd main
```

Like expected, *libshared.so* is among the linked libraries, but it cannot be found:

```
...
linux-vdso.so.1 (0x00007fff635fe000)
libshared.so => not found
libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00007f334b44d000)
...
```

We will also get an error when trying to execute our program:

```
$> ./main
./main: error while loading shared libraries: libshared.so: cannot open share
```

To solve this problem, we must somehow specify where the library can be found. On Linux, the environment variable LD_LIBRARY_PATH can be used to specify directories where libraries are searched for first (before the standard set of directories)((GNU Dynamic Loader search directories)) => prepending our commands above with LD_LIBRARY_PATH=./ solves the issue of the not found library and the program will execute like expected:

```
$> LD_LIBRARY_PATH=. ./main
f()
X::X()
X::mX()
```

Shared libraries with Windows/MinGW

To be able to compile/link our code with different compilers, we have to consider some compiler specifics.

1. MSVC cannot directly link to a dll -> we need a so-called “import library”.
2. Symbol Visibility: GCC/MinGW exports all symbols by default, MSVC exports no symbol by default.

Linking the DLL directly

If you are on Windows and compile everything with MinGW (the DLL and the application), linking directly to the dll is fine. You can take the source code from above without any changes and build it very similar like on Linux:

```
g++ -fPIC --shared -o shared.dll shared.cpp
g++ -L. -lshared -o main main.cpp
```

The above lines produce *shared.dll* and *main.exe* that links directly against *shared.dll*.

Creating and linking the import library

Like shown above, it is perfectly fine to directly link the DLL if the DLL was also created by MinGW. But this is not the common way to work with shared libraries on windows because just a few compilers support this - the *Microsoft Visual Studio Compiler* is not among them.

On Windows, a DLL usually has a matching so called *"import library"*. If an executable wants to use functions of a DLL it does not link the DLL directly but it's *import library*.

Import libraries are similar to static libraries and usually also have a .lib file ending. In contrast to static libraries, they do not contain the complete function definitions, but just "stubs" of the exported symbols of the DLL to satisfy the linker and to load the DLL at runtime.

We can create the DLL and its corresponding import library with MinGW((Creating import libraries with MinGW)):

```
g++ -c shared.cpp -o shared.o
g++ -shared -o shared.dll shared.o -Wl,--out-implib,libshared_dll.lib
```

The option `-Wl,--out-implib,libshared_dll.lib` tells the compiler to also create the corresponding import library _libshared*dll.lib*. Then we compile our application and link the import library instead of the dll:

```
g++ -o main.exe main.o -L. -llibshared_dll
```

Symbol visibility

The default behavior of MinGW is to export *ALL* symbols of a DLL. But if one or more functions are declared with `__declspec(dllexport)`, only those functions are exported.

The behavior of the Microsoft Visual Studio Compiler is different: Per default *NO* symbols are exported, just symbols explicitly declared with `__declspec(dllexport).`

So let's make the code "MSVC compatible", without loosing the compatibility to the other compilers:

We add an additional file `shared_EXPORTS.h`:

```
#ifndef _SHARED_EXPORTS_H__
#define _SHARED_EXPORTS_H__

#ifdef _WIN32
    #ifdef shared_EXPORTS
        #define SHARED_EXPORT __declspec(dllexport)
    #else
        #define SHARED_EXPORT __declspec(dllimport)
    #endif
#else
    #define SHARED_EXPORT
#endif

#endif /* _SHARED_EXPORTS_H__ */
```

and modify our shared.h as follows:

```
#ifndef SHARED_H__
#define SHARED_H__

#include "shared_EXPORTS.h"

void SHARED_EXPORT f();

class SHARED_EXPORT X {
public:
  X();
  void mX();
};

#endif
```

We compile the code now similar like before but we define `shared_EXPORTS` by passing `-Dshared_EXPORTS` to gcc when compiling `shared.cpp`:

```
g++ -Dshared_EXPORTS -c shared.cpp
g++ -shared -o shared.dll shared.o -Wl,--out-implib,libshared_dll.lib
g++ -o main.exe main.o -L. -llibshared_dll
```

When using the library (i.e. linking it to main.exe), `shared_EXPORTS` is not defined, and `SHARED_EXPORT` is set to `__declspec(dllimport)`. This means when we build the library, our functions are exported and when we want to use our library, the functions are imported from the dll.

Creating shared libraries with CMake

Up to now, we can build the same source code on Linux with GCC and on Windows with MinGW. We also prepared the code to be compilable with the Microsoft Visual Studio Compiler (MSVC).

Now we also want to build and use a DLL with other compilers like MSVC. Therefore we don't create the in Visual Studio solution by hand, instead

we choose a more flexible method and use CMake.

CMake is a cross-platform build tool that tremendously simplifies building C/C++ code with different compilers on different platforms. With just a few steps we can build our code without having to worry about the exact compilation/linking steps. If you don't know CMake, I'll highly recommend you to check it out. This is no CMake tutorial, I'll just show the general procedure:

1. At first we create generic project description in a more or less "declarative", compiler and platform independent way. Therefore we create a file "CMakeLists.txt" in the source root folder with a just few lines (see below).
2. Then, we run "cmake" on the platform where we want to build. This generates a platform and compiler specific, "native" build description (e.g. Makefiles, a Visual Studio Solution or others) from the generic CMakeLists.txt.
3. To actually build the code, we have to invoke the "native build command", e.g. "make" if we use Makefiles.

ad 1.) The "generic project description" in CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
project(sharedLibsDemo)                 # create a project with the given nam
add_library(shared SHARED shared.cpp)   # compile "shared.cpp" and create a S
add_executable(main main.cpp)           # compile "main.cpp" the the executab
include (GenerateExportHeader)
GENERATE_EXPORT_HEADER(shared           # generates the export header shared_
    BASE_NAME shared
    EXPORT_MACRO_NAME shared_EXPORTS
    EXPORT_FILE_NAME shared_EXPORTS.h
    STATIC_DEFINE SHARED_EXPORTS_BUILT_AS_STATIC)
target_link_libraries(main shared)      # link our previously created shared
```

By using the `GENERATE_EXPORT_HEADER` CMake macro like above, you don't need to create the `shared_EXPORTS.h` on your own - it's created automatically by CMake.

ad 2.+3.) Let CMake generate the platform and compiler specific project description with CMake:

on **Linux/GCC**, generate Unix Makefiles and build with make:

```
cmake .
make
```

on **Windows/MinGW,** to generate MinGW Makefiles and build with mingw32-make.exe:

```
cmake -G "MinGW Makefiles" .
mingw32-make.exe
```

on **Windows/MSVC** to generate Visual Studio 12 Solution and build the ALL_BUILD target:

```
cmake -G "Visual Studio 12 2013" .
cmake --build . --target ALL_BUILD --config Release
```

CMake makes it quite easy to build with **any other of the supported compilers**. E.g., building with clang and Ninja can be done as simple as:

```
export CXX=/usr/bin/clang++
export CC=/usr/bin/clang
cmake -G "Ninja"
ninja
```

Interoperability of Libraries (ABI compatibility)

Libraries written in C++, compiled with different compilers, or even just with different releases of the same compiler, often cannot be linked together[2]. Sometimes there exists basic ABI (Application Binary Interface) compatibility (e.g. GCC and clang) but it's likely that you have to recompile your library and your application from source with the same compiler to be able to link them.

DLLs are slightly different: If the DLL is written in C it can be linked even with applications written in C++. DLLs written in C++ work too, as long as you expose its symbols through a C interface declared with `extern "C"`. Extern "C" makes a function-name in C++ have 'C' linkage and no compiler specific name mangling is applied. For C++ classes this means you have to make some kind of "C wrapper" and expose those functions declared with `extern "C"`. If you try to use the C++ interface of a library compiled with a different compiler you will likely get linker errors because different compilers often use incompatible name mangling schemes of the exported C++ symbols (the exported symbols are named differently).

For our demo shared library from above this means: We cannot build our demo with MinGW and link it to our application "main" built with MSVC.

But we can easily declare our C function `f()` with `extern "C"`. This would allow us to use function f() of the DLL even in applications build with different compilers:

```
...
extern "C" void SHARED_EXPORT f();
...
```

To also make the functionality of class X available to be used by other compilers (e.g. you want to distribute the library in binary form), the most robust way would be to create a C wrapper: wrap the methods into C functions, and declare them as `extern "C"`. There are also other techniques to create a binary compatible C++ interface (at least to a certain degree) - if you are interested in how this can be done, I suggest you to read this post[3].

Of course there is no ABI compatibility between libraries/applications compiled on different operating systems unless you *cross compile* - but that's another topic.

The final version of the minimalistic portable code demo, that is gradually developed in this post, is available on github.

If you have questions or any kind of feedback please leave a comment below.

---

1. Wikipedia: Shared Library ↑
2. Interoperability of Libraries Created by Different Compiler Brands ↑
3. Binary-compatible C++ Interfaces ↑

Ilsa • 4 years ago

Hi, I am trying to follow your article, but when I try to compile the main executable, I get 'undefined reference' errors. At first I thought I had made a typo, so to be sure I copy and pasted the individual shared.h, shared.cpp and main.ccp files, as well as the commands to compile them. Yet I still get the same error. When I use your final version built with CMake, it compiles and executes just fine.

$ g++ -L. -lshared -o main main.cpp
/tmp/ccmhqt2X.o: In function `main':
main.cpp:(.text+0x18): undefined reference to `f()'
main.cpp:(.text+0x24): undefined reference to `X::X()'
main.cpp:(.text+0x30): undefined reference to `X::mX()'
collect2: error: ld returned 1 exit status

Do you have any suggestions? I am using the defailt g++ included with Linux Mint, which is 5.4.0, if that helps.

2 ^ | v • Reply • Share ›

Liam ➚ Ilsa • 4 years ago

For anyone else with this issue, theres a small error in this post (or at least g++ syntax has changed since it was written). you need to specify the library AFTER the file that uses it.
e.g.
g++ -L. -o main main.cpp -lshared

4 ^ | v • Reply • Share ›

John Freeman • 4 months ago • edited

This works for an in-source build. For an out-of-source build, authors will need to add `target_include_directories(shared PRIVATE ${CMAKE_CURRENT_BINARY_DIR})`.

Small typo in the call to `generate_export_header`: `EXPORT_MACRO_NAME` should be `SHARED_EXPORT` based on the code. (It is fixed on GitHub.)

^ | v • Reply • Share ›

trivk • 3 years ago

Thanks for the article ... Very clear explanation!

^ | v • Reply • Share ›

Pavel Rudko • 5 years ago

Hi, nice article!
I've got a question.
Imagine I am building a shared library A (A.lib is an import library and A.dll is the library itself) and an executable (C.exe) that depends on

A. The library A depends on a 3rd party library B (which consists of B.lib and B.dll). The library A is built successfully after linking B.lib. Is there a way to build C.exe linking A.lib and without linking B.lib?

Reply • Share ›

Gernot Klingler ➚ Pavel Rudko • 5 years ago

Hi Pavel,
you don't have to link B to C.exe if C.exe does not call anything from B. C.exe should be linked against A and A should be linked agains B (the import libraries respectively).
Take a look at his simple demo I just commited to github: https://github.com/gklingle....

Reply • Share ›

Pavel ➚ Gernot Klingler • 5 years ago

Thank you very much! You've helped me a lot.

Reply • Share ›

James Hamilton • 5 years ago

Great example! Very useful.Thanks for your contribution

Reply • Share ›

Louis Louca • 5 years ago

Thank you Gernot,

I looked everywhere for a clear and simple explanation and it was no where to be found.
You made a complex subject easy to understand.

Thanks again.

Louis

Reply • Share ›

Ben Phillips • 6 years ago