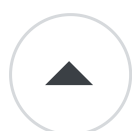


Why does the order in which libraries are linked sometimes cause errors in GCC?

Asked 16 years, 3 months ago Modified 9 months ago

Viewed 267k times



Why does the order in which libraries are linked sometimes cause errors in GCC?

605



gcc

linker



Share



Improve this question

Follow

edited Jun 26, 2019 at 12:31



Konrad Rudolph

545k ● 139 ● 956 ● 1.2k

asked Sep 5, 2008 at 2:24



Landon

15.6k ● 12 ● 39 ● 30

- 1 See also now [stackoverflow.com/questions/7826448/...](https://stackoverflow.com/questions/7826448/) -- TLDR gcc changed to more-strict behavior (relatively) recently. – tripleee Jan 5, 2015 at 6:17

11 Answers

Sorted by:

Highest score (default)





(See the history on this answer to get the more elaborate text, but I now think it's easier for the reader to see real command lines).

717



Common files shared by all below commands



```
// a depends on b, b depends on d
$ cat a.cpp
extern int a;
int main() {
    return a;
}

$ cat b.cpp
extern int b;
int a = b;

$ cat d.cpp
int b;
```

Linking to static libraries

```
$ g++ -c b.cpp -o b.o
$ ar cr libb.a b.o
$ g++ -c d.cpp -o d.o
$ ar cr libd.a d.o

$ g++ -L. -ld -lb a.cpp # wrong order
$ g++ -L. -lb -ld a.cpp # wrong order
$ g++ a.cpp -L. -ld -lb # wrong order
$ g++ a.cpp -L. -lb -ld # right order
```

The linker searches from left to right, and notes unresolved symbols as it goes. If a library resolves the

symbol, it takes the object files of that library to resolve the symbol (b.o out of libb.a in this case).

Dependencies of static libraries against each other work the same - the library that needs symbols must be first, then the library that resolves the symbol.

If a static library depends on another library, but the other library again depends on the former library, there is a cycle. You can resolve this by enclosing the cyclically dependent libraries by `-(` and `-)`, such as `-(-la -lb -)` (you may need to escape the parens, such as `-\\(` and `-\\)`). The linker then searches those enclosed lib multiple times to ensure cycling dependencies are resolved. Alternatively, you can specify the libraries multiple times, so each is before one another: `-la -lb -la`.

Linking to dynamic libraries

```
$ export LD_LIBRARY_PATH=. # not needed if libs go
to /usr/lib etc
$ g++ -fpic -shared d.cpp -o libd.so
$ g++ -fpic -shared b.cpp -L. -ld -o libb.so #
specifies its dependency!
```

```
$ g++ -L. -lb a.cpp # wrong order (works on some
distributions)
$ g++ -Wl,--as-needed -L. -lb a.cpp # wrong order
$ g++ -Wl,--as-needed a.cpp -L. -lb # right order
```

It's the same here - the libraries must follow the object files of the program. The difference here compared with static libraries is that you need not care about the dependencies of the libraries against each other, because *dynamic libraries sort out their dependencies themselves*.

Some recent distributions apparently default to using the `--as-needed` linker flag, which enforces that the program's object files come before the dynamic libraries. If that flag is passed, the linker will not link to libraries that are not actually needed by the executable (and it detects this from left to right). My recent archlinux distribution doesn't use this flag by default, so it didn't give an error for not following the correct order.

It is not correct to omit the dependency of `b.so` against `d.so` when creating the former. You will be required to specify the library when linking `a` then, but `a` doesn't really need the integer `b` itself, so it should not be made to care about `b`'s own dependencies.

Here is an example of the implications if you miss specifying the dependencies for `libb.so`

```
$ export LD_LIBRARY_PATH=. # not needed if libs go
to /usr/lib etc
$ g++ -fpic -shared d.cpp -o libd.so
$ g++ -fpic -shared b.cpp -o libb.so # wrong (but
links)

$ g++ -L. -lb a.cpp # wrong, as above
$ g++ -Wl,--as-needed -L. -lb a.cpp # wrong, as
above
$ g++ a.cpp -L. -lb # wrong, missing libd.so
```

```
$ g++ a.cpp -L. -ld -lb # wrong order (works on  
some distributions)  
$ g++ -Wl,--as-needed a.cpp -L. -ld -lb # wrong  
order (like static libs)  
$ g++ -Wl,--as-needed a.cpp -L. -lb -ld # "right"
```

If you now look into what dependencies the binary has, you note the binary itself depends also on `libd`, not just `libb` as it should. The binary will need to be relinked if `libb` later depends on another library, if you do it this way. And if someone else loads `libb` using `dlopen` at runtime (think of loading plugins dynamically), the call will fail as well. So the "right" really should be a `wrong` as well.

Share Improve this answer

Follow

edited Nov 10, 2022 at 1:20



Louis Go

2,468 ● 2 ● 20 ● 35

answered Jan 3, 2009 at 17:53




Johannes Schaub - litb

506k ● 131 ● 917 ● 1.2k

16 Repeat until all symbols resolved, eh - you'd think they could manage a topological sort. LLVM has 78 static libraries on it's own, with who-knows-what dependencies. True it also has a script to figure out compile/link options - but you can't use that in all circumstances. – user180247 Jul 11, 2010 at 13:27

9 @Steve that's what the programs `lorder` + `tsort` do. But sometimes there is no order, if you have cyclic references. Then you just have to cycle through the libraries list until everything is resolved. – Johannes Schaub - litb Aug 18, 2011 at 19:01 ✎

14 @Johannes - Determine the maximal strongly connected components (e.g. Tarjans algorithm) then topologically sort the (inherently non-cyclic) digraph of components. Each component can be treated as one library - if any one library from the component is needed, the dependency cycle(s) will cause all libraries in that component to be needed. So no, there really is no need to cycle through all the libraries in order to resolve everything, and no need for awkward command-line options - one method using two well-known algorithms can handle all cases correctly. – user180247 Aug 18, 2011 at 22:44

12 I would like to add one important detail to this excellent answer: Using "`-(archives -)`" or "`--start-group archives --end-group`" **is the only sure-fire way of resolving circular dependencies**, since each time the linker visits an archive, it pulls in (and registers the unresolved symbols of) *only the object files that resolve currently unresolved symbols*. Because of this, CMake's algorithm of repeating connected components in the dependency graph may occasionally fail. (See also [Ian Lance Taylor's excellent blog post](#) on linkers for more detail.) – jorgen Oct 25, 2013 at 7:34 

5 Fantastic write up: it gets an up-vote from me. I would add to the section on *resolving cyclic dependencies*, I'll leave it to the original poster if they wish to modify, that when passing the `--start-group` and `--end-group` options using gcc, you must precede with `-Wl,<option>` as in `-Wl,--start-group <archives> -Wl,--end-group`. It is mentioned in the manual page which is referenced, but a quick mention here is helpful too. – Andrew Falanga Sep 4, 2014 at 16:27



The GNU ld linker is a so-called smart linker. It will keep track of the functions used by preceding static libraries, permanently tossing out those functions that are not used

145



from its lookup tables. The result is that if you link a static library too early, then the functions in that library are no longer available to static libraries later on the link line.



The typical UNIX linker works from left to right, so put all your dependent libraries on the left, and the ones that satisfy those dependencies on the right of the link line.

You may find that some libraries depend on others while at the same time other libraries depend on them. This is where it gets complicated. When it comes to circular references, fix your code!

Share Improve this answer

edited Jan 5, 2012 at 3:30

Follow

answered Jan 3, 2009 at 17:21



casualcoder

4,990 ● 6 ● 31 ● 36

3 Is this something with only gnu ld/gcc? Or is this something common with linkers? – [Mike](#) Sep 5, 2008 at 3:17


2 Apparently more Unix compilers have similar issues. MSVC isn't entirely free of these issues, either, but they don't appear to be that bad. – [MSalters](#) Apr 28, 2009 at 14:42

6 The MS dev tools don't tend to show these issues as much because if you use an all-MS tool chain it ends up setting up the linker order properly, and you never notice the issue. – [Michael Kohne](#) Aug 28, 2009 at 20:59

21 The MSVC linker is less sensitive to this issue because it will search all libraries for an unreferenced symbol. Library order still can affect *which* symbol gets resolved if more than one

library have the symbol. From MSDN: "Libraries are searched in command line order as well, with the following caveat: Symbols that are unresolved when bringing in an object file from a library are searched for in that library first, and then the following libraries from the command line and /DEFAULTLIB (Specify Default Library) directives, and then to any libraries at the beginning of the command line"

– [Michael Burr](#) Apr 24, 2012 at 6:19 

26 "... *smart linker* ..." - I believe it is classified as a "single pass" linker, not a "smart linker". – [jww](#) Aug 28, 2018 at 3:34 



63

Here's an example to make it clear how things work with GCC when **static** libraries are involved. So let's assume we have the following scenario:



- `myprog.o` - containing `main()` function, dependent on `libmysqlclient`
- `libmysqlclient` - static, for the sake of the example (you'd prefer the shared library, of course, as the `libmysqlclient` is huge); in `/usr/local/lib`; and dependent on stuff from `libz`
- `libz` (dynamic)

How do we link this? (Note: examples from compiling on Cygwin using gcc 4.3.4)

```
gcc -L/usr/local/lib -lmysqlclient myprog.o
# undefined reference to `_mysql_init'
# myprog depends on libmysqlclient
# so myprog has to come earlier on the command
line
```



```
gcc myprog.o -L/usr/local/lib -lmysqlclient
# undefined reference to `_uncompress'
# we have to link with libz, too

gcc myprog.o -lz -L/usr/local/lib -lmysqlclient
# undefined reference to `_uncompress'
# libz is needed by libmysqlclient
# so it has to appear *after* it on the command
line

gcc myprog.o -L/usr/local/lib -lmysqlclient -lz
# this works
```

Share Improve this answer

answered Jul 16, 2011 at 12:32

Follow



Lumi

15.2k ● 9 ● 61 ● 92



55

If you add `-wl,--start-group` to the linker flags it does not care which order they're in or if there are circular dependencies.



On Qt this means adding:

```
QMAKE_LFLAGS += -wl,--start-group
```



Saves loads of time messing about and it doesn't seem to slow down linking much (which takes far less time than compilation anyway).

Share Improve this answer

edited Jan 10, 2018 at 19:18

Follow



darrenp

4,465 ● 2 ● 28 ● 22

answered Apr 5, 2015 at 12:24



SvaLopLop

979 ● 1 ● 10 ● 14

5 it works because of `/usr/bin/ld: missing --end-group; added as last command line option`
– [leanid.chaika](#) Sep 16, 2021 at 22:10

Though looks like it does not always work because of missing `--end-group` . – [Smar](#) Oct 18, 2022 at 5:47

It works, but makes the linker a dumb linker, it just links everything. – [casualcoder](#) Jan 27, 2023 at 2:21



12



Another alternative would be to specify the list of libraries twice:

```
gcc prog.o libA.a libB.a libA.a libB.a -o prog.x
```

Doing this, you don't have to bother with the right sequence since the reference will be resolved in the second block.

Share Improve this answer

Follow

answered Mar 21, 2014 at 10:07



eckes

66.9k ● 31 ● 175 ● 206



10

A quick tip that tripped me up: if you're invoking the linker as "gcc" or "g++", then using "--start-group" and "--end-group" won't pass those options through to the linker -- nor will it flag an error. It will just fail the link with undefined symbols if you had the library order wrong.



You need to write them as "-Wl,--start-group" etc. to tell GCC to pass the argument through to the linker.



Share Improve this answer

answered Aug 11, 2013 at 14:07

Follow



M.M

141k ● 25 ● 220 ● 387



You may can use -Xlinker option.

6

```
g++ -o foobar -Xlinker -start-group -Xlinker  
libA.a -Xlinker libB.a -Xlinker libC.a -Xlinker -  
end-group
```



is ALMOST equal to



```
g++ -o foobar -Xlinker -start-group -Xlinker  
libC.a -Xlinker libB.a -Xlinker libA.a -Xlinker -  
end-group
```

Careful !

1. The order within a group is important ! Here's an example: a debug library has a debug routine, but the non-debug library has a weak version of the same. You must put the debug library FIRST in the group or you will resolve to the non-debug version.
2. You need to precede each library in the group list with -Xlinker

Share Improve this answer

edited Jan 11, 2012 at 6:44

Follow



sra

24k ● 8 ● 58 ● 89

answered Aug 9, 2011 at 8:06



yundorri

61 ● 1 ● 1



4

Link order certainly does matter, at least on some platforms. I have seen crashes for applications linked with libraries in wrong order (where wrong means A linked before B but B depends on A).



Share Improve this answer

answered Apr 28, 2009 at 13:57



Follow



David Cournapeau

80.6k ● 9 ● 68 ● 71



4

I have seen this a lot, some of our modules link in excess of a 100 libraries of our code plus system & 3rd party libs.



Depending on different linkers

HP/Intel/GCC/SUN/SGI/IBM/etc you can get unresolved functions/variables etc, on some platforms you have to list libraries twice.



For the most part we use structured hierarchy of libraries, core, platform, different layers of abstraction, but for some systems you still have to play with the order in the link command.

Once you hit upon a solution document it so the next developer does not have to work it out again.

My old lecturer used to say, "*high cohesion & low coupling*", it's still true today.

Share Improve this answer

edited Jan 10, 2018 at 19:45

Follow



darrenp

4,465 ● 2 ● 28 ● 22

answered Sep 5, 2008 at 3:56



titanae

2,289 ● 2 ● 21 ● 31



0



It matters where you put your files for a linker to resolve.

It is a general rule that you put files where `symbol` is referenced BEFORE file where `symbol` is defined.



To understand the reason of error you have to understand how linker works:



- All files passed to linker processed from left to right
- Linker collects undefined symbols from files to the pool of undefined symbols
- If object from archive doesn't resolve any symbols from pool of undefined symbols, then it dropped

In a case:

```
gcc -o libfunc2.so -shared -fPIC -lfunc1 func2.o
```

We put archive library where symbol `show1()` is defined before object file `func2.o` where it is referenced. So `libfunc1.a` **doesn't resolve any symbols and is dropped.**

And from [GCC -l library option](#):

It makes a difference where in the command you write this option; the linker searches and processes libraries and object files in the order they are specified.

Thus, `foo.o -lz bar.o`

searches library `z` after file `foo.o` but before `bar.o`. If `bar.o` refers to functions in library `z`, those functions may not be loaded.

Share Improve this answer

edited Dec 20, 2023 at 23:33

Follow

answered Dec 20, 2023 at 23:20



Dražen Grašovec

792 ● 10 ● 18



-1



ld : linker daemon has been around since 1990's and it's not a program we spend a lot of time with because the automake/cmake makefile recipes are pretty easy to obtain most behaviour.

It has a LDFLAGS option known as `--start-group` or `-(`





```
-( archives -)
```

```
--start-group archives --end-group
```

This allows you to search more than once for symbols while recursing through the specific archives looking for symbols.

This is from the linker manual:

```
--start-group archives --end-group
```

The archives should be a list of archive files. They may be either explicit file names, or ``-l'` options. The specified archives are searched repeatedly until no new undefined references are created. Normally, an archive is searched only once in the order that it is specified on the command line. If a symbol in that archive is needed to resolve an undefined symbol referred to by an object in an archive that appears later on the command line, the linker would not be able to resolve that reference. By grouping the archives, they all be searched repeatedly until all possible references are resolved. Using this option has a significant performance cost. It is best to use it only when there are unavoidable circular references between two or more archives.

The linker daemon is like a spaghetti colander, a bunch of symbols arrive at the colander walls and poke through the holes, and the linker must track all symbols and connect together all the symbols by address linkage. If it misses some, the program will crash.

Here's what you need to know about static versus dynamic libraries When linking symbols for static libraries, ld starts from left to right and MAKES ONE PASS through all symbol tables. If you want to resolve dependency loops then you can add the same archive into the linker archive objects list more than once.

When dealing with dynamic libraries, all symbols must be resolved by the dynamic libraries so nothing gets missed.

Therefore, the one time behaviour will fail is when some symbols are static and the dynamic symbols will not be around during the single pass.

The answer then, is place the static libraries TO THE LEFT of the dynamic libraries in the make recipe.

Share Improve this answer

answered Mar 7 at 19:17

Follow



daemondave

305 ● 3 ● 13

I like the substance of your answer and the emphasis on the one-pass nature of the process (by default), but a couple of historical notes: the ld command has been around since at least the first version of Unix, so we are talking early 1970s, not 90s. Also, it is not a daemon (a persistent background process); that seems like a backronym. While I haven't found

any sources to prove it, my guess is that "ld" stands for "loader", a term that was sometimes used interchangeably with "linker" back then, and is indeed present in the v1 ld manpage. – [itub](#) Apr 10 at 12:15



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.