

Link 6. Library Search Examples

Young W. Lim

2024-07-01 Mon

- 1 Based on
- 2 Examples of search libraries
 - TOC: Examples of search libraries
 - 1. Example source code and dependencies
 - 2. -L and -l examples
 - 3. Using -rpath-link and -rpath
 - 4. -rpath-link examples
 - 5. -rpath examples
 - 6. Summary and more examples
 - 7. Using -Wl, -rpath, .

"Study of ELF loading and relocs", 1999

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

I, the copyright holder of this work, hereby publish it under the following licenses: GNU head Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

CC BY SA This file is licensed under the Creative Commons Attribution ShareAlike 3.0 Unported License. In short: you are free to share and make derivative works of the file under the conditions that you appropriately attribute it, and that you distribute it only under a license compatible with this one.

Compiling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

TOC: Examples of search libraries

- 1 Example source code and dependencies
- 2 `-L` and `-l` examples
- 3 `-rpath-link` examples
- 4 `-rpath` examples
- 5 `-Wl,-rpath,.` examples

TOC: 1. Example source code and dependencies

- Example source codes of `foo()`, `bar()`, `foobar()`
- Function dependencies of `foo()`, `bar()`, `foobar()`

Example source codes of foo(), bar(), foobar()

1. foo.c

```
#include <stdio.h>

void foo(void)
{
    puts(__func__);
    // puts("foo");
}
```

2. bar.c

```
#include <stdio.h>

void bar(void)
{
    puts(__func__);
    // puts("bar");
}
```

3. foobar.c

```
extern void foo(void);
extern void bar(void);

void foobar(void)
{
    foo();
    bar();
}
```

4. main.c

```
extern void foobar(void);

int main(void)
{
    foobar();
    return 0;
}
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

Function dependencies of `foo()`, `bar()`, `foobar()`

<code>main()</code>	<code>→</code>	<code>foobar()</code>
<code>foobar()</code>	<code>→</code>	<code>foo(), bar()</code>

<code>main()</code>	in	<code>prog</code>
<code>foobar()</code>	in	<code>libfoobar.so</code>
<code>foo()</code>	in	<code>libfoo.so</code>
<code>bar()</code>	in	<code>libbar.so</code>

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

direct and nested dependencies of a binary

binary	direct dependencies	nested dependencies
<code>libfoobar.so</code>	<code>→ libfoo.so,</code> <code>→ libbar.so</code>	
<code>prog</code>	<code>→ libfoobar.so</code>	<code>→ libfoo.so,</code> <code>→ libbar.so</code>

Specifying dependencies and search paths (1)

	dependencies	link time search paths	runtime search paths
<code>-l</code>	<input type="radio"/>		
<code>-L</code>		<input type="radio"/>	
<code>-rpath-link</code>		<input type="radio"/>	
<code>-rpath</code>		<input type="radio"/>	<input type="radio"/>

Specifying dependencies and search paths (2)

	for direct dependencies	for nested dependencies
--	-------------------------	-------------------------

Method 1	<code>-L d_direct -l direct</code>	<code>-L d_nest -l nest</code>
----------	------------------------------------	--------------------------------

Method 2	<code>-L d_direct -l direct</code>	<code>-rpath-link d_nest</code>
----------	------------------------------------	---------------------------------

Method 3	<code>-L d_direct -l direct</code>	<code>-rpath d_nest</code>
----------	------------------------------------	----------------------------

Method 4	<code>-L d_direct -l direct</code>	<code>-rpath d_direct</code>
----------	------------------------------------	------------------------------

Specifying dependencies and search paths (3)

Method 1	<code>-L d_direct -l direct -L d_nest -l nest</code>
Method 2	<code>-L d_direct -l direct -rpath-link d_nest</code>
Method 3	<code>-L d_direct -l direct -rpath d_nest</code>

need to specify *runtime* search paths, e.g.,
export LD_LIBRARY_PATH=dir1:dir2

Method 4	<code>-L d_direct -l direct -rpath d_direct</code>
----------	--

no need to specify *runtime* search paths
`-rpath` enables each binary to *record*
its *direct* search paths in the `RUNPATH` entry
of its `.dynamic` section

TOC: 2. -L and -l examples

- Example summary using -L and -l
- Making `libfoo.so`, `+libbar.so=`
- Making `libfoobar.so`
 - Using -L. `-lfoo -lbar` to make `libfoobar.so`
- Making an application prog that uses `libfooba.so`
 - Not specifying nested dependencies
 - Warning and error messages
 - Using -L and -l to make an application
 - Need to specify runtime search paths
 - More experiment with nested dependencies
- Specifying the runtime shared library paths
 - Using `LD_LIBRARY_PATH` to run an application

Example summary using `-L` and `-l`

- 1 Make two shared libraries, `libfoo.so` and `libbar.so`:

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o
```

- 2 Make a third shared library, `libfoobar.so`

```
$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar
```

- 3 Make `prog` that depends on `libfoobar.so`:

```
$ gcc -c -Wall main.c
$ gcc -o prog main.o -L. -lfoobar -lfoo -lbar
```

- 4 Execute using `LD_LIBRARY_PATH`

```
$ export LD_LIBRARY_PATH=.
$ ./prog
foo
bar
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

Making libfoo.so and libbar.so

- Make two shared libraries, `libfoo.so` and `libbar.so`:

```
$ gcc -c -Wall -fPIC foo.c bar.c
```

```
$ gcc -shared -o libfoo.so foo.o
```

```
$ gcc -shared -o libbar.so bar.o
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

Making libfoobar.so

- Make a third shared library, `libfoobar.so` that depends on the first two (`libfoo.so`, `libbar.so`)

```
$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -lfoo -lbar
/usr/bin/ld: cannot find -lfoo
/usr/bin/ld: cannot find -lbar
collect2: error: ld returned 1 exit status
```

- *direct dependencies* (`libfoo.so` and `libbar.so`) were specified by `-lfoo` and `-lbar`
- but could not find the libraries (`libfoo.so` and `-libbar.so`)
`/usr/bin/ld: cannot find -lfoo`
`/usr/bin/ld: cannot find -lbar`
- because the linker (`ld`) didn't know where to look to *resolve* `-lfoo` or `-lbar` thus were not able to *resolve* them

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

Using `-L. -lfoo -lbar` to make `libfoobar.so`

- The `-L.` informs where to look to resolve `-lfoo` and `-lbar`

```
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar
```

- the `-L` option (`-Ldir`) tells the linker (`ld`) to search `dir` for libraries to resolve **dependencies** that are specified by the `-l` option
 - the linker (`ld`) searches the `-L` directories, in their command line order;
eg. when multiple `-L` options are used like `-Ldir1 -Ldir2` `dir1` is searched first, then `dir2`
- then it searches its configured default directories, in their configured order.

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

Making an application prog that uses libfoobar.so

- make a program `prog` that depends on `libfoobar.so`:

```
$ gcc -c -Wall main.c  
$ gcc -o prog main.o -L. -lfoobar
```

- `libfoo.so` and `libbar.so` are the *direct* dependencies of `libfoobar.so`, and thus the *nested* dependencies of `prog`
- only *direct* dependency is specified (`-lfoobar`) with the correct search path (`-L.`)
- *nested* dependencies are not specified (`-lfoo -lbar`) but `libfoo.so` and `libbar.so` can be found in the specified search path (`-L.`)

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

Not specifying *nested* dependencies

- although `-lfoo` and `-lbar` are not specified,

```
$ gcc -c -Wall main.c  
$ gcc -o prog main.o -L. -lfoobar
```

- by looking into `NEEDED` entry of the `.dynamic` section of `libfoobar.so`,
- the linker (`ld`) detects the *nested dynamic dependencies* but they were not specified with `-lfoo -lbar`
warning : not found libfoo.so, not found libbar.so
- the linker (`ld`) did not resolve the *nested* dependencies because they were not specified
error: undefined reference to foo, undefined reference to bar

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

Warning and error messages

- make a program `prog` that depends on `libfoobar.so`:
 - the *nested* dependencies are not specified (`-lfoo -lbar`) though with the correct search path (`-L.`)
 - not found `libfoo.so` ← `-lfoo` not specified
 - not found `libbar.so` ← `-lbar` not specified
 - undefined reference to `bar` ← `-lbar` not resolved
 - undefined reference to `foo` ← `-lfoo` not resolved

```
$ gcc -c -Wall main.c
$ gcc -o prog main.o -L. -lfoobar
/usr/bin/ld: warning: libfoo.so, needed by ./libfoobar.so, not found
(try using -rpath or -rpath-link)
/usr/bin/ld: warning: libbar.so, needed by ./libfoobar.so, not found
(try using -rpath or -rpath-link)
./libfoobar.so: undefined reference to `bar'
./libfoobar.so: undefined reference to `foo'
collect2: error: ld returned 1 exit status
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

Using only `-L` and `-l` to make an application

- to resolve the *nested dependencies*, we will consider the following ways
 - 1 `-L` and `-l`
 - 2 `-rpath-link`
 - 3 `-rpath`
- let us first ignore the gcc compiler's advice
try using `-rpath` or `-rpath-link`
- to handle *nested dependencies*, try first using `-L` and `-l`
 - search path for *nested dependencies* : `-L.`
(the same directory specified for `libfoobar.so`)
 - *nested dependencies* : `-lfoo -lbar`

```
$ gcc -o prog main.o -L. -lfoobar -lfoo -lbar
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

Need to specify runtime search paths

- now, the application `prog` can be made, but cannot be made to run:

```
$ gcc -o prog main.o -L. -lfoobar -lfoo -lbar
```

```
$ ./prog
```

```
./prog: error while loading shared libraries: libfoobar.so:\ncannot open shared object file: No such file or directory
```

- at the `runtime`, the loader (`ld.so`) could not find `libfoobar.so` nor `libfoo.so` nor `libbar.so`
- need to specify the `runtime` search paths

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

More experiment with nested dependencies

- before specifying runtime search paths, let's experiment more with *nested dependencies*
- move `libfoo.so` and `libbar.so` libraries to `lib2`

```
$ mkdir lib2
$ mv libfoo.so libbar.so lib2
```

- then, make `prog` as before

```
$ gcc -o prog main.o -L. -lfoobar -lfoo -lbar
```

- the *nested dependencies* were specified (`-lfoo -lbar`)
- but the linker (`ld`) could not find `libfoo.so` and `libbar.so` at the specified directory (`-L.`)

```
/usr/bin/ld: cannot find -lfoo
/usr/bin/ld: cannot find -lbar
collect2: error: ld returned 1 exit status
```

- the correct search path `-Llib2` must also be specified

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

Specifying the runtime shared library paths

- now move `libfoo.so`, `libbar.so` back to the current directory . and make `prog` again

```
mv lib2/libfoo.so lib2/libbar.so .  
$ gcc -o prog main.o -L. -lfoobar -lfoo -lbar
```

- the `-L` option is used to tell the linker (`ld`) where to *find the libraries* (shared objects) at the compile, and link time
- lots of ways to tell the runtime linker (dynamic loader `ld.so`) where to *find the libraries* (shared objects) at the runtime
 - `-R`
 - `LD_LIBRARY_PATH`
 - `LD_RUN_PATH`

<https://stackoverflow.com/questions/31455979/how-to-specify-libraries-paths-in-gcc>

Using LD_LIBRARY_PATH to run an application

- `prog` is made by using `-L` and `-l` only
not by using `-rpath` nor `-rpath-link`

```
$ gcc -o prog main.o -L. -lfoobar -lfoo -lbar
```

- `prog` is made run by us `LD_LIBRARY_PATH`

```
$ export LD_LIBRARY_PATH=.  
$ ./prog  
foo  
bar
```

- at the runtime, `LD_LIBRARY_PATH` enables the loader (`ld.so`)
to find `libfoobar.so`, `libfoo.so`, and `libbar.so`
in the current directory .

```
export LD_LIBRARY_PATH=.
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

TOC: 3. -rpath-link

Dependency related entries of the `.dynamic` section

- DT_NEEDED**
- created by `-L -l` options of `gcc` compiler
 - specifies direct dependencies
 - can be used to find nested dependencies
-

- DT_RPATH / DT_RUNPATH**
- created by `-rpath` option of `ld` linker
 - specifies runtime search path
 - **DT_RPATH** is deprecated
 - searches direct and nested dependency paths
 - **DT_RUNPATH** is not supported by all systems
 - searches only direct dependency paths
-

What is RPATH (1)

- if an executable `foo` links to the shared library `bar`, the library `bar` has to be *found* and *loaded* when the executable `foo` is *executed*.
- this (searching and loading the shared library) is done by the linker, `ld.so`.
- the linker *searches* a set of directories for the library `bar`, `libbar.so`

<https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling>

What is RPATH (2)

- The linker will search the shared libraries in the following directories in the given order:
 - 1 **RPATH** (deprecated)
 - 2 **LD_LIBRARY_PATH**
 - 3 **RUNPATH** (only direct dependency paths)
 - 4 **/etc/ld.so.conf**

<https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling>

What is RPATH (3)

1 RPATH

- a list of directories which is linked into the executable
- it is ignored if **RUNPATH** is present.

2 LD_LIBRARY_PATH

- an environment variable which holds a list of directories

3 RUNPATH

- same as **RPATH**, but searched after **LD_LIBRARY_PATH**, supported only on most current Linux systems

4 `/etc/ld.so.conf`

- configuration file for **ld.so** which lists additional library directories (builtin directories) basically `/lib` and `/usr/lib`

<https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling>

What is RPATH (4-1)

- different reasons for needs for other directories to be searched than the builtin ones
 - ① a user may install a library *privately* into his *home directory*, e.g. `~/lib/`
 - ① there may be different *versions* of the same library installed, e.g. `/opt/kde3/lib/libkdecore.so` and `/opt/kde4/lib/libkdecore.so`

<https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling>

What is RPATH (4-2)

- 1 a user may install a library *privately* into his *home directory*, e.g. `~/lib/`
 - in this case, `LD_LIBRARY_PATH` can be set
`export LD_LIBRARY_PATH=$HOME/lib:$LD_LIBRARY_PATH`

<https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling>

What is RPATH (6)

- This will break for the second case, where for some programs `/opt/kde3/lib` has to be searched and for other applications `/opt/kde4/lib` has to be searched, but in no case both.
- The only way to have an executable-dependent library search path is by using RPATH (or RUNPATH, but this isn't supported everywhere).

<https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling>

Handling *direct* and *nested* dependencies

- *direct dependency* must be handled by specifying `-L` and `-l`
- *nested dependencies* can be handled by specifying `-rpath-link` or `-rpath`

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

Specifying `-L` and `-l` handles *direct* dependencies

- the *direct* **dependencies** of the current binary must be handled by `-L` and `-l`
 - specifying `-L` and `-l` creates **NEEDED** entries in `.dynamic` section of the current binary
 - by specifying `-rpath-link` or `-rpath`
 - the **NEEDED** entries are not created, but
 - the **NEEDED** entries of each binary can be utilized to find the *nested* **dependencies** of a given binary

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

Specifying `-rpath-link` handles *nested* dependencies

- the `-rpath-link=dir` option tells the linker (`ld`) that when *dynamic nested dependencies* are requested, directory `dir` is searched to *resolve* them.
- only for a successful linkage,
`-rpath-link` specifies the *directories* where the *nested* dependencies of the current binary can be found

```
$ gcc -o prog main.o -L. -lfoo -Wl,-rpath-link=$(pwd)
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

-rpath-link v.s. -rpath

- `-rpath-link=dir`
 - provides the linker with `runtime search path` information
 - but does not instruct the linker to write that information into `RUNPATH` or `RPATH` entries in the `.dynamic` section
- `-rpath=dir`
 - also provides the linker with `runtime search path` information
 - and instructs the linker to write that information into `RUNPATH` or `RPATH` entries in the `.dynamic` section

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

`-rpath-link` does not create `RUNPATH` / `RPATH` entries

- `-rpath-link=dir`

- does not guarantee us a *runnable prog*
but only a *successful linkage*

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
$ ./prog
./prog: error while loading shared libraries: libfoobar.so
cannot open shared object file: No such file or directory
```

- creates neither `RUNPATH` nor `RPATH`
- therefore, in order to *execute prog*,
runtime search path must be specified explicitly

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

`-rpath` creates `RUNPATH` / `RPATH` entries

- `-rpath=dir`
 - creates `RUNPATH` or `RPATH` entries in the `.dynamic` section to specify `runtime search path`
 - `RUNPATH` (for modern gcc)
 - `RPATH` (for older gcc)
 - guarantees us a *runnable prog*
 - no need to specify `runtime search path` explicitly

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

RPATH v.s. RUNPATH (1)

- in the `.dynamic` section of a binary (*executable* or *shared library*)
 - the `RPATH` entry is used by default in the older versions of gcc
 - `RPATH` allows nested dependencies to inherit the specified search path
 - the `RUNPATH` entry is used by default in modern versions of gcc
 - `RUNPATH` applies the search path only to the direct dependencies of the *current binary* (no recursive application)

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

RPATH v.s. RUNPATH (2)

older gcc **RPATH** all dependencies (direct, nested)
utilize the specified path

modern gcc **RUNPATH** only direct dependencies
utilize the specified path

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

Runtime search path precedence (1)

- at the **runtime**, to locate a **dynamic library** (**.so** file), the loader (dynamic linker) will try the followings
 - search **RPATH** (older versions of gcc)
 - search **LD_LIBRARY_PATH**
 - search **RUNPATH** (modern versions of gcc)
 - directories on the **system search path**, which consists of the entries in **/etc/ld.so.conf** plus **/lib** and **/usr/lib**

<https://unix.stackexchange.com/questions/22926/where-do-executables-look-for-shared-libraries>

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-and-runpath>

Runtime search path precedence (2)

- there are many other ways to specify the **runtime search path**
- **-rpath-link=dir** does not give any information of **runtime search path**

<https://unix.stackexchange.com/questions/22926/where-do-executables-look-for-shared-libraries>
<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-link-and-rpath>

`-rpath-link` in `bfd` and `gold` linkers

	<code>bfd ld</code>	<code>gold ld</code>
<code>-rpath-link</code>	(0)	(X) ignored
<code>DT_NEEDED</code>	(0)	(X) not used

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

bfd ld and -rpath-link (1)

- The `--rpath-link` option is used by `bfd ld` to add to the search path used for finding `DT_NEEDED` shared libraries
(direct dependencies of a given binary)
when doing link-time symbol resolution
 - by following `DT_NEEDED` entries recursively indirect (nested) dependencies can be found

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

bfd ld and -rpath-link (2)

- It's basically telling the linker what to use as the **runtime search path** when attempting to mimic what the dynamic linker would do when **resolving symbols**
- as the **runtime search path** set by **--rpath** options or the **LD_LIBRARY_PATH** environment variable

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

gold ld and -rpath-link

- **gold linker** does not follow **DT_NEEDED** entries when resolving symbols in shared libraries,
- so the **--rpath-link** option is ignored when **gold linker** is used
- this was a deliberate design decision;
indirect (nested) dependencies
do not need to be present
or in their runtime locations during the link process.

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-link>

Example summary using `-rpath-link`

- 1 Make two shared libraries, `libfoo.so` and `libbar.so`

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o
```

- 2 Make a third shared library, `libfoobar.so`

```
$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar
```

- 3 Make `prog` that depends on `libfoobar.so`

```
$ gcc -c -Wall main.c
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
```

- 4 Execute using `LD_LIBRARY_PATH`

```
$ export LD_LIBRARY_PATH=.
$ ./prog
foo
bar
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-link>

Making libfoo.so and libbar.so

- Make two shared libraries, `libfoo.so` and `libbar.so`:

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o
```

- no *direct* dependencies
- no specification with `-l`
- no **NEEDED** entries for *direct* dependencies that are specified by a user

```
$ readelf -d libfoo.so | grep NEEDED
Tag                Type                Name/Value
0x0000000000000001 (NEEDED)    Shared library: [libc.so.6]
```

```
$ readelf -d libbar.so | grep NEEDED
Tag                Type                Name/Value
0x0000000000000001 (NEEDED)    Shared library: [libc.so.6]
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

Making libfoobar.so

- Make a third shared library, `libfoobar.so` that depends on the first two (`libfoo.so`, `libbar.so`)

```
$ gcc -c -Wall -fPIC foobar.c
```

```
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar
```

- *direct dependencies* were specified by `-lfoo -lbar`
- these *dependencies* were recorded as the `NEEDED` entries in the `.dynamic` section of `libfoobar.so`

```
$ readelf -d libfoobar.so | grep NEEDED
```

Tag	Type	Name/Value
0x0000000000000001	(NEEDED)	Shared library: [libfoo.so] <---
0x0000000000000001	(NEEDED)	Shared library: [libbar.so] <---

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

Making an application prog that uses libfooar.so

- make a program `prog` that depends on `libfooar.so`:

```
$ gcc -c -Wall main.c
```

```
$ gcc -o prog main.o -L. -lfooar -Wl,-rpath-link=$(pwd)
```

- only *direct* dependency was specified (`-lfooar`) with the correct search path (`-L.`)
- *nested* dependencies were not specified (`-lfoo -lbar`) but can be handled by `-rpath-link=$(pwd)`
 - `libfoo.so` and `libbar.so` are the *direct* dependencies of `libfooar.so`, and thus the *nested* dependencies of `prog`

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

Creating **NEEDED** entries

- make a program **prog** that depends on **libfoobar.so**:

```
$ gcc -c -Wall main.c
```

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
```

- in the **.dynamic** section of **prog**
 - *direct dependency* specified by **-lfoobar** was recorded as **NEEDED** entries
 - *nested dependency*, even though specified by **-lfoo -lbar**, are not recorded as **NEEDED** entries

```
$ readelf -d prog | grep NEEDED
```

Tag	Type	Name/Value
0x0000000000000001	(NEEDED)	Shared library: [libfoobar.so] <---
0x0000000000000001	(NEEDED)	Shared library: [libc.so.6]

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

NEEDED entries and nested dependencies (1)

- `libfoo.so`, `libbar.so` :
 - these are the *direct dependencies* of `libfoobar.so`
 - thus, these are the *nested dependencies* of `prog`
 - when `libfoobar.so` was made, its *direct dependencies* were specified with `-lfoo -lbar`
 - this allows the *direct dependencies* of `libfoobar.so` to be recorded as **NEEDED** entries in the `.dynamic` section of `libfoobar.so`

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

NEEDED entries and nested dependencies (2)

- `-rpath-link=dir`
 - the *nested dependencies* of `prog` can be found through the **NEEDED** entries in the `.dynamic` section of the *direct dependency* of `prog`
 - when `prog` was made, its *direct dependency* were specified with `-lfoobar`
 - the *direct dependencies* of `libfoobar.so` can be found by looking the **NEEDED** entries in the `.dynamic` section of `libfoobar.so`
 - the directory `dir` will be searched for these *nested dependencies* of `prog`

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

NEEDED entries of each binary

binary	dependencies	entry	section
<code>prog</code>	<code>libfoobar.so</code>	NEEDED	<code>.dynamic</code>
<code>libfoobar.so</code>	<code>libfoo.so,</code> <code>libbar.so</code>	NEEDED	<code>.dynamic</code>

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

Using LD_LIBRARY_PATH to specify a runtime search path

- but the **loader** might be able to locate them
 - through the **ldconfig** cache or
 - a setting of the **LD_LIBRARY_PATH** environment variable, e.g:

```
$ export LD_LIBRARY_PATH=.; ./prog  
foo  
bar
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

-rpath-link example (6)

- `-rpath-link=dir`
 - gives the linker (`ld`) the directory information that the loader (`ld.so`) *would* need to resolve some of the **dynamic dependencies** of `prog` at **runtime**
 - assuming that the directory information remained true at **runtime**
 - but does not write that directory information into the **.dynamic** section of `prog`
 - only the *direct* dependency (`libfoobar.so`) is written in the **.dynamic** section of `prog`

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

TOC: 5. -rpath examples

Example summary using `-rpath` (1)

- 1 Make two shared libraries, `libfoo.so` and `libbar.so`:

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o
```

- 2 Make a third shared library, `libfoobar.so` that depends on the first two;

```
$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar -Wl,-rpath=$(pwd)
```

- 3 Make an application, `prog` that depends on `libfoobar.so`

```
$ gcc -c -Wall main.c
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath=$(pwd)
```

- 4 Make `prog` run

```
# to show that this environment variable is not used
export LD_LIBRARY_PATH= # clear the env variable
$ ./prog
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

-rpath example (4)

- **prog** contains the **runtime search path** information for shared libraries that **prog** depends on

```
$ gcc -c -Wall main.c
gcc -o prog main.o -L. -lfoobar -Wl,-rpath=$(pwd)
```

```
# $(pwd) --> /home/imk/develop/so/scrap
```

```
$ readelf -d prog
```

```
Dynamic section at offset 0xe08 contains 26 entries:
```

Tag	Type	Name/Value
0x0000000000000001	(NEEDED)	Shared library: [libfoobar.so]
0x0000000000000001	(NEEDED)	Shared library: [libc.so.6]
0x000000000000000f	(RUNPATH)	Library rpath: [/home/imk/develop/so/scrap]
...	
...	

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

-rpath example (5)

- `libfoobar.so` (direct dependency) will be found at **runtime**, but `libfoo.so` and `libbar.so` (nested dependencies) won't,
 - because `libfoobar.so` does not inherit **RUNPATH** information of `prog`
- `-rpath=$(pwd)` must be specified also for `libfoobar.so` to write *runtime search path* information into **RUNPATH** entry of the **.dynamic** section of `libfoobar.so`

```
$ gcc -c -Wall -fPIC foobar.c
```

```
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar -Wl,-rpath=$(pwd)
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

-rpath example (6)

- check what libraries are needed by `libfoobar.so` could be:

```
$ readelf -d ./libfoobar.so
```

```
Dynamic section at offset 0xe38 contains 22 entries:
```

Tag	Type	Name/Value
0x0000000000000001	(NEEDED)	Shared library: [libfoo.so]
0x0000000000000001	(NEEDED)	Shared library: [libbar.so]
0x0000000000000001	(NEEDED)	Shared library: [libc.so.6]
0x000000000000000f	(RPATH)	Library rpath: [/home/imk/develop/so/scrap]
(...)		

<https://unix.stackexchange.com/questions/571861/is-there-an-rpath-for-dynamic-linking>

-rpath example (7)

- `prog` executable depends on `libfoobar.so` shared object
`RUNPATH` entry of `.dynamic` section of `prog` set by

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath=$(pwd)
```

- `libfoobar.so` shared object depends on
`libfoo.so` and `libbar.so` shared objects
`RUNPATH` entry of `.dynamic` section of `libfoobar.so` set by

```
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar -Wl,-rpath=$(pwd)
```

- to run `prog` does not need to set `LD_LIBRARY_PATH`

```
$ LD_LIBRARY_PATH=  
$ ./prog  
foo  
bar
```

<https://unix.stackexchange.com/questions/571861/is-there-an-rpath-for-dynamic-linking>

-rpath example (8*)

- **RPATH** is searched in before **LD_LIBRARY_PATH**
- **RUNPATH** is searched in after **LD_LIBRARY_PATH**
 - ① search **RPATH** (older versions of gcc)
 - ② search **LD_LIBRARY_PATH**
 - ③ search **RUNPATH** (modern versions of gcc)
 - ④ search **ldconfig**-ed directories

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

<https://refspecs.linuxbase.org/elf/gabi4+/ch5.dynamic.html>

-rpath example (9*)

- if `-Wl,--disable-new-dtags` is specified
`RPATH` is used as if 'older versions' of gcc were used,
instead of `RUNPATH`
 - makes *nested* dependencies inherit the specified search path
 - thus, `-rpath=$(pwd)` need not be specified for `libfoobar.so`

```
$ export LD_LIBRARY_PATH=
```

```
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar
```

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath=$(pwd) -Wl,--disable-new-dtags
```

```
$ ./prog
```

```
foo
```

```
bar
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-l>

TOC: 5. Summary and more examples

Directories for example shared libraries

- 1 using `-L` and `-l`
- 2 using `-rpath-link`
- 3 using `-rpath` (like `-rpath-link`)
- 4 using `-rpath` (using `RUNPATH`)

binaries	compile time	run time 1,2,3	run time 4
<code>libfoo.so</code>	<code>./lib2</code>	<code>./lib2</code>	<code>./librun</code>
<code>libbar.so</code>	<code>./lib2</code>	<code>./lib2</code>	<code>./librun</code>
<code>libfoobar.so</code>	<code>./lib</code>	<code>./librun</code>	<code>./librun</code>
<code>prog</code>	<code>.</code>	<code>.</code>	<code>.</code>

Specifying dependencies and search paths (1)

Method 1	<code>-L d_direct</code>	<code>-l direct</code>	<code>-L d_nest</code>	<code>-l nest</code>
Method 2	<code>-L d_direct</code>	<code>-l direct</code>	<code>-rpath-link d_nest</code>	
Method 3	<code>-L d_direct</code>	<code>-l direct</code>	<code>-rpath d_nest</code>	
Method 4	<code>-L d_direct</code>	<code>-l direct</code>	<code>-rpath d_direct</code>	

<code>libfoobar.so</code>	<code>direct</code>	<code>foo</code> and <code>bar</code>	<code>d_direct</code>	<code>lib2</code>
	<code>nested</code>	-	<code>d_nest</code>	-
<code>prog</code>	<code>direct</code>	<code>libfoobar</code>	<code>d_direct</code>	<code>lib</code> or <code>librun</code>
	<code>nested</code>	<code>foo</code> and <code>bar</code>	<code>d_nest</code>	<code>lib2</code>

Specifying dependencies and search paths (2)

- for `libfoobar.so`

- ① `-Llib2 -lfoo -lbar`

- ② `-Llib2 -lfoo -lbar`

- ③ `-Llib2 -lfoo -lbar`

- ④ `-Llib2 -lfoo -lbar -Wl,-rpath=lib:librun`

- for `prog`

- ① `-Llib -lfoobar -Llib2 -lfoo -lbar`

- ② `-Llib -lfoobar -Wl,-rpath-link=lib2`

- ③ `-Llib -lfoobar -Wl,-rpath=lib2`

- ④ `-Llib -lfoobar -Wl,-rpath=lib:librun`

Example2 summary using -L and -l

1 Make `libfoo.so` and `libbar.so` in `./lib2`

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o
$ mv libfoo.so libbar.so lib2
```

2 Make `libfoobar.so` in `./lib`

```
$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -Llib2 -lfoo -lbar
$ mv libfoobar.so lib
```

3 Make `prog` in `.`

```
$ gcc -c -Wall main.c
$ gcc -o prog main.o -Llib -lfoobar -Llib2 -lfoo -lbar
```

4 Execute using `LD_LIBRARY_PATH` (libraries in `librun`, `lib2`)

```
$ mv lib/libfoobar.so librun
$ export LD_LIBRARY_PATH=librun:lib2
$ ./prog
```

Specifying *dependencies* using `-l` only

- unless `rpath-link` or `rpath` is used
 - *all dependencies* must be specified with `-l` (*direct* and *nested dependencies*)
 - specify `-lfoo` (*direct dependency* of `prog`)
 - specify `-lfoo -lbar` (*nested dependencies* of `prog`)

```
$ gcc -o prog main.o -L. -lfoo -lbar
```
 - the *direct dependency* of `prog` : `libfoo.so`
 - the *nested dependencies* of `prog` : `libfoo.so`, `libbar.so`
(the *direct dependencies* of `libfoo.so`)

Specifying *link time* search paths using `-L` only

- `-Ldir1 -Ldir2 ...`
 - when an input file *requests* **dynamic dependencies** the linker *searches* the specified directories to *resolve* them.
 - specify all the search paths for *direct* and *nested* **dependencies**
 - since *nested* dependencies does *inherit* the search path
 - the linker *searches* **dir1**, **dir2**, etc., only to *resolve* references
 - only for a successful linkage, not for a successful execution
 - in this example, to link successfully, `$(pwd)` is searched
 - for **libfoobar.so** (the *direct* dependency)
 - for **libfoo.so** and **libbar.so** (the *nested* dependencies)

```
$ gcc -o prog main.o -L. -lfoobar -lfoo -lbar
```


Example2 summary using `-rpath-link`

- 1 Make `libfoo.so` and `libbar.so` in `./lib2`

```
gcc -c -Wall -fPIC foo.c bar.c
gcc -shared -o libfoo.so foo.o
gcc -shared -o libbar.so bar.o
mv libfoo.so libbar.so lib2
```

- 2 Make `y`, `libfoobar.so` in `./lib`

```
gcc -c -Wall -fPIC foobar.c
gcc -shared -o libfoobar.so foobar.o -Llib2 -lfoo -lbar
mv libfoobar.so lib
```

- 3 Make `prog` in `.`

```
gcc -c -Wall main.c
gcc -o prog main.o -Llib -lfoobar -Wl,-rpath-link=lib2
```

- 4 Execute using `LD_LIBRARY_PATH` (libraries in `librun`, `lib2`)

```
mv lib/libfoobar.so librun
export LD_LIBRARY_PATH=librun:lib
./prog
```

Specifying *dependencies* using `-rpath-link=dir`

- when `rpath-link` or `rpath` is used
 - specify only *direct dependencies* using `-l` and their search paths with `-L`
 - no need to specify *nested dependencies*
 - *nested dependencies* can be found by the `NEEDED` entry in the `.dynamic` section of a given *direct dependency*
 - `-lfoobar` necessary
 - `-lfoo -lbar` unnecessary
- ```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
```
- the *direct dependency* of `prog : libfoobar.so`
  - the *nested dependencies* of `prog : libfoo.so, libbar.so` (the *direct dependencies* of `libfoobar.so`)

# Specifying *link time* search paths using `-rpath-link=dir`

- when `-rpath-link=dir` is used
  - since *nested* dependencies do inherit the search path
  - specify all the search paths for *direct* and *nested* dependencies using `rpath-link=dir1:dir2` or multiple `rpath-link` options
  - only for a successful linkage, not for a successful execution
  - in this example, to link successfully, `$(pwd)` is searched
    - for `libfoobar.so` (the *direct* dependency)
    - for `libfoo.so` and `libbar.so` (the *nested* dependencies)

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
```

## Example2 summary using `-rpath` (like `-rpath-link`)

- 1 Make `libfoo.so` and `libbar.so` in `./lib2`

```
gcc -c -Wall -fPIC foo.c bar.c
gcc -shared -o libfoo.so foo.o
gcc -shared -o libbar.so bar.o
mv libfoo.so libbar.so lib2
```

- 2 Make `libfoobar.so` in `./lib`

```
gcc -c -Wall -fPIC foobar.c
gcc -shared -o libfoobar.so foobar.o -Llib2 -lfoo -lbar
mv libfoobar.so lib
```

- 3 Make `prog` in `.`

```
gcc -c -Wall main.c
gcc -o prog main.o -Llib -lfoobar -Wl,-rpath=lib2
```

- 4 Execute using `LD_LIBRARY_PATH` (libraries in `librun`, `lib2`)

```
mv lib/libfoobar.so librun
export LD_LIBRARY_PATH=librun:lib
./prog
```

## Example2 summary using `-rpath` (using `RUNPATH`)

- 1 Make `libfoo.so` and `libbar.so` in `./lib2`

```
gcc -c -Wall -fPIC foo.c bar.c
gcc -shared -o libfoo.so foo.o
gcc -shared -o libbar.so bar.o
mv libfoo.so libbar.so lib2
```

- 2 Make `libfoobar.so` in `./lib`

```
gcc -c -Wall -fPIC foobar.c
gcc -shared -o libfoobar.so foobar.o -Llib2 -lfoo -lbar -Wl,-rpath=lib:librun
mv libfoobar.so lib
```

- 3 Make `prog` in `.`

```
gcc -c -Wall main.c
gcc -o prog main.o -Llib -lfoobar -Wl,-rpath=lib2:librun
```

- 4 Execute without `LD_LIBRARY_PATH` (now all libraries in `librun`)

```
mv lib/libfoobar.so lib2/libfoo.so lib2/libbar.so librun
export LD_LIBRARY_PATH=
./prog
```

# Specifying *dependencies* using `-rpath=dir`

- when `rpath-link` or `rpath` is used
  - specify only *direct dependencies* using `-l` and their search paths with `-L`
  - no need to specify *nested dependencies*
    - *nested dependencies* can be found by the **NEEDED** entry in the `.dynamic` section of a given *direct dependency*

- `-lfoobar` necessary
- `-lfoo -lbar` unnecessary

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath=$(pwd)
```

- the *direct dependency* of `prog : libfoobar.so`
- the *nested dependencies* of `prog : libfoo.so, libbar.so` (the *direct dependencies* of `libfoobar.so`)

# Specifying *link time* search paths using `-rpath=dir`

- when `-rpath` is used, there are two approaches for specifying the *link time* search paths
  - 1 specify *all* the search paths for *direct* and *nested dependencies* of a given binary using `-rpath`
    - for a successful linkage only, not for a successful execution
    - since *nested* dependencies *inherit* the search path
    - as long as specifying *link time* search paths are concerned, the `rpath` option is the same as the `rpath-link` option
  - 2 let each binary be specified with search paths using `-rpath` for its *direct dependencies* only
    - those paths are recorded as *runtime* search paths in the `RUNPATH` entry of `.dynamic` section of a binary

# Specifying *run time* search paths using `-rpath=dir`

- `-rpath=dir`
  - the `ld` searches directory `dir` to *resolve* references
  - the `ld.so` searches directory `dir` to *load* shared libraries
  - to load shared libraries, *nested* dependencies may not inherit the search path
  - for modern versions of `gcc` that use `RUNPATH` instead `RPATH` do not allow the search path to be *inherited*
    - thus, each binary should be specified with search paths for its *direct dependencies*, using `-rpath`
    - that those paths may be recorded as *runtime* search path in the `RUNPATH` entry of `.dynamic` section of the binary

```
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar -Wl,-rpath=$(pwd)
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath=$(pwd)
```



# TOC: 5. -Wl,-rpath, . examples

## Using `-Wl, rpath .` (1)

- in order to pass `-rpath .` to the linker, consider them as two arguments (`-rpath` and `.`) to the `-Wl`
- you can write `(-Wl, arg1, arg2)` or `(-Wl, arg1, -Wl, arg2)`
  - `-Wl, -rpath, .`
  - `-Wl, -rpath -Wl, .`

<https://stackoverflow.com/questions/6562403/i-dont-understand-wl-rpath-wl>

## Using `-Wl, -rpath, .` (2)

- the `-Wl,xxx` option for `gcc` passes a **comma**-separated list of tokens as a **space**-separated list of arguments to the linker (`ld`)
- to pass `ld aaa bbb ccc` (space separated)  
`gcc -Wl,aaa,bbb,ccc` (comma separated)
- to pass `ld -rpath .` (space separated)  
`gcc -Wl,-rpath,.` (comma separated)

<https://stackoverflow.com/questions/6562403/i-dont-understand-wl-rpath-wl>

## Using `-Wl,-rpath,.` (3)

- alternatively, **repeat instances** of `-Wl` can be specified
- to pass `ld aaa bbb ccc` (space separated)  
`gcc -Wl,aaa -Wl,bbb -Wl,ccc` (repeated instances)
  - there is no comma between `-Wl,aaa` and the second `-Wl,bbb`  
but there is space
- thus, to pass `ld -rpath .`
  - `gcc -Wl,-rpath,.` (comma separated)
  - `gcc -Wl,-rpath -Wl,.` (repeated instances)

<https://stackoverflow.com/questions/6562403/i-dont-understand-wl-rpath-wl>

## Using `-Wl,-rpath, .` (4)

- can remove the comma by using `=`  
`gcc -Wl,-rpath=.`
  - arguably more readable than adding extra commas
  - exactly what gets passed to `ld`
- thus, to pass `ld -rpath .`
  - `gcc -Wl,-rpath, .` (comma separated)
  - `gcc -Wl,-rpath -Wl, .` (repeated instances)
  - `gcc -Wl,-rpath=.` (using `=` instead of `,`)

<https://stackoverflow.com/questions/6562403/i-dont-understand-wl-rpath-wl>

## Using `-Wl,-rpath, .` (5)

- You may need to specify the `-L` option as well

```
-Wl,-rpath,/path/to/foo -L/path/to/foo -lbaz
```

or you may end up with an error like

```
ld: cannot find -lbaz
```

<https://stackoverflow.com/questions/6562403/i-dont-understand-wl-rpath-wl>