

Link 5A Library Search using RPATH

Young W. Lim

2024-07-11 Thr

- 1 Based on
- 2 1. Background of RPATH / RUNPATH
 - TOC: Overview of RPATH / RUNPATH
 - 1. Using `-rpath-link` and `-rpath`
 - 2. Using `-Wl,-rpath,.`
- 3 2. Example
 - TOC: 2. Example
 - 1. Example source code and dependencies
 - 2. `-rpath-link` examples
 - 3. `-rpath` examples
 - 4. Summary
- 4 3. More Examples
 - TOC: 3. More Examples
 - 1. More Examples

"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`

- 1 Using `-rpath-link` and `-rpath`
- 2 Using `-Wl,-rpath,.`

TOC: Using `-rpath-link` and `-rpath`

- What is RPATH
- Dependency related entries of the `.dynamic` section
- Handling *direct* and *nested* dependencies
- Specifying `-L` and `-l` handles *direct* dependencies
- Specifying `-rpath-link` handles *nested* dependencies
- `-rpath-link` v.s. `-rpath`
- `-rpath-link` does not create `RUNPATH` / `RPATH` entries
- `-rpath` creates `RUNPATH` / `RPATH` entries
- RPATH v.s. RUNPATH
- `-rpath-link` in `bfd` and `gold` linkers
- `bfd ld` and `-rpath-link`
- `gold ld` and `-rpath-link`

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, i.e., 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 are searched)
 - 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**
- ignored if **RUNPATH** is present (**RPATH** is deprecated)

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)

- 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 (5)

- 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)

- 2 there may be different *versions* of the same library installed, e.g. `/opt/kde3/lib/libkdecore.so` and `/opt/kde4/lib/libkdecore.so`
- cases for some programs `/opt/kde3/lib` has to be searched and for other applications `/opt/kde4/lib` has to be searched, but never both directories
- the only way to have an executable-dependent library search path is by using **RPATH** (deprecated) or **RUNPATH** (not always supported)

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

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
-

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 makes only a successful linkage

- `-rpath-link=dir`
 - does not guarantee us a *runnable prog*
but only a *successful linkage*

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

<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` does not create `RUNPATH` / `RPATH` entries

- there are many other ways to specify the **runtime search path**
- `-rpath-link=dir` does not give any information of **runtime search path**
 - does not creates `RUNPATH`
 - does not creates `RPATH`
 - therefore, for a *successful execution*, explicit specification of **runtime search path** may be needed.

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

`-rpath` does create `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-1>

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-l>

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>

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

	bfd ld	gold ld
-rpath-link	(0)	(X) ignored
DT_NEEDED	(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>

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>

- 1 Example source code and dependencies
- 2 `-rpath-link` examples
- 3 `-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()`

```
main()    → foobar()
foobar()  → foo(), bar()
```

```
main()    in prog
foobar()  in libfoobar.so
foo()     in libfoo.so
bar()     in libbar.so
```

<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>

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-l>

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. Summary

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

① More Exammples

TOC: 5. More Examples

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>-</code>	<code>d_nest</code>	<code>-</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`

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>

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 `-lfoobar` (*direct dependency* of `prog`)
 - specify `-lfoo -lbar` (*nested dependencies* of `prog`)

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