Shared Libraries and Where To Find Them

LUIS CARO CAMPOS

Shared Libraries and Where to Find Them

CONAN
C/C++ Package Manager

JFrog

Luis Caro Campos
R&D Team Lead, JFrog

Introduction

From the point of view of a C++ developer, **compiled code** typically ends up in either:

- The executables themselves
- Static libraries
- **Shared libraries**

Libraries* are a vehicle for “reusable” code code that can be invoked by other libraries or applications (even from other languages)

* libraries: not defined by the standard! A word of many meanings…

How developers deal with shared libraries

LNK1104 wl

Asked 16 days ago

I have a c++ DLL proj
to use this library in n

Here is what I did so f

1. Added MyDll project as a reference in the console app.
2. Added MyDll.lib to the console's **Linker > Input**
3. Added MyDll's header path to the **Additional Include Directories**

Both projects hav
`$(SolutionDir)`
are being genera

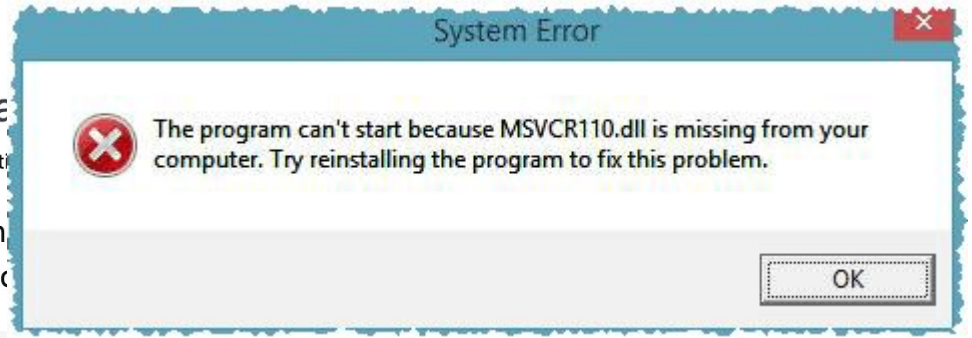
I was able to h
directory. This
for library? Ho

ld: warning: `Could not find or use auto-linked framework 'CoreAudioTypes': framework 'CoreAudioTypes' not found` Undefined symbols for architecture arm64: "_main", referenced from: ld: symbol(s) not found for architecture arm64 clang: error: `linker command failed with exit code 1 (use -v to see invocation)`

Why do recompile attempts fa

Asked 9 days ago Modified 9 days ago Viewed 36 t

When attempting to recompile an app th
running AlmaLinux 8 the following error




```
/usr/bin/ld: cannot find -lgfortran
collect2: error: ld returned 1 exit status
make: [pest.mak:307: pest] Error 1
```

About this talk

- What this talk covers:
  - WHEN shared libraries are needed
  - WHICH programs needs to locate shared libraries
  - TOOLS to query, troubleshoot
  - WHERE to find shared libraries

- What this talk is NOT about:
  - Shared vs static linking
  - Internal structures of shared libraries
  - Linker specifics, relocations, advanced techniques

WHEN shared libraries are needed

When creating other shared libraries and executables

When running executables:

- Launching them
- Loading loadable plugins

When configuring the build

When packaging application for distribution

WHEN shared libraries are needed

When configuring the build

When creating other shared libraries and executables

BUILD

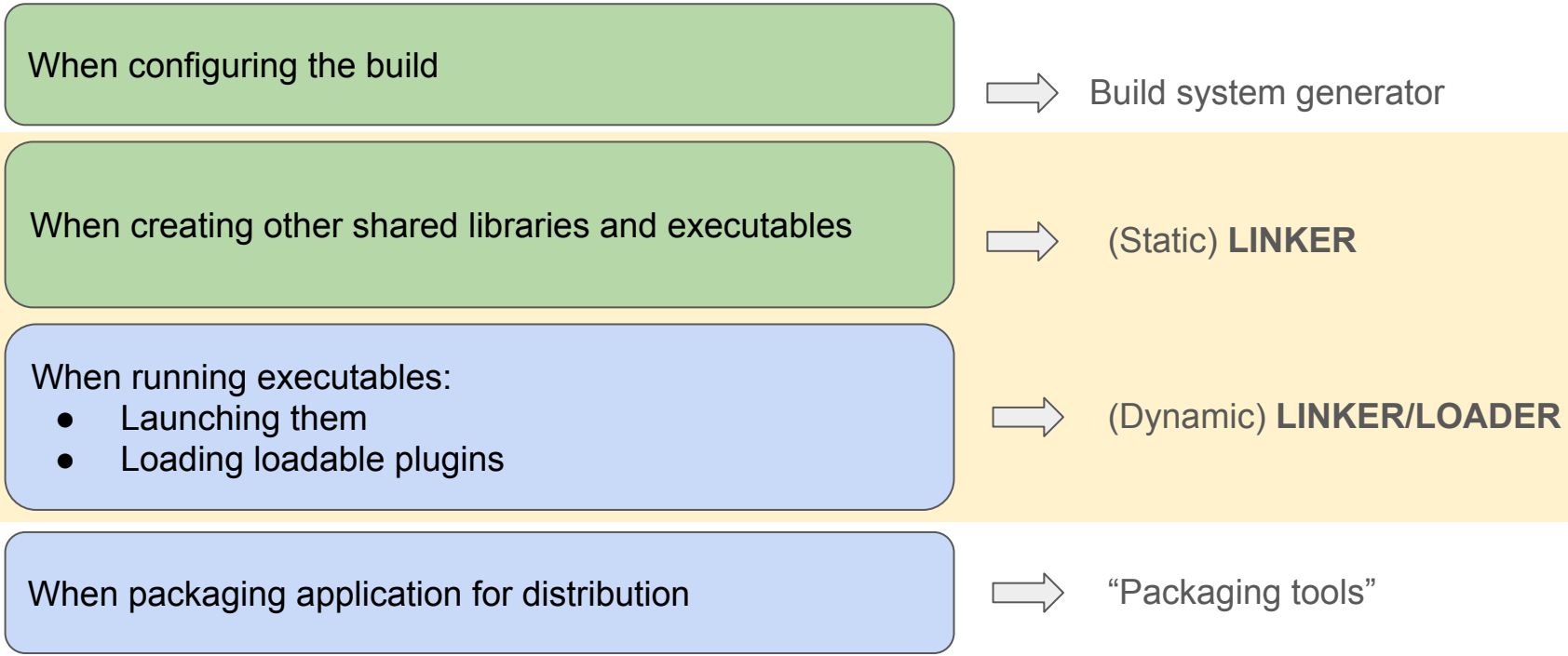
When running executables:

- Launching them
- Loading loadable plugins

When packaging application for distribution

RUN

WHICH programs need to locate shared libraries



GNU Linux

Linux example

We want to create an executable that links to libfoo

```cpp
#include <foo.h>

int main() {
   i_am_foo();
   return 0;
}
```

my_program.cpp

Library exists and is installed:

- /usr/include/foo.h
- /usr/lib/libfoo.so

```
$ g++ my_program.cpp -o my_program
/usr/bin/ld: /tmp/cc8r2SFr.o: in function `main':
my_program.cpp:(.text+0x8): undefined reference to `i_am_foo()'
```

Linux example

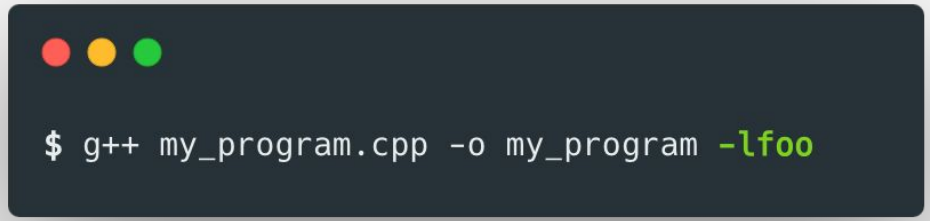
We want to create an executable

```cpp
#include <foo.h>

int main() {
   i_am_foo();
   return 0;
}
```

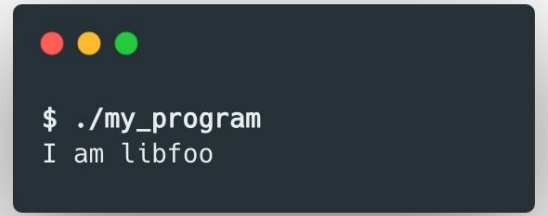
my_program.cpp

Library exists and is installed:

- /usr/include/foo.h
- /usr/lib/libfoo.so



Linux example



We've shown both stages:

- The **linker** (`ld`) finds `libfoo.so` when creating the executable
  - The linker is indirectly invoked by gcc
  - Part of the compiler toolchain
- The **runtime linker-loader** (`ld.so`) found the library when launching it
  - Launched the application
  - Part of the system

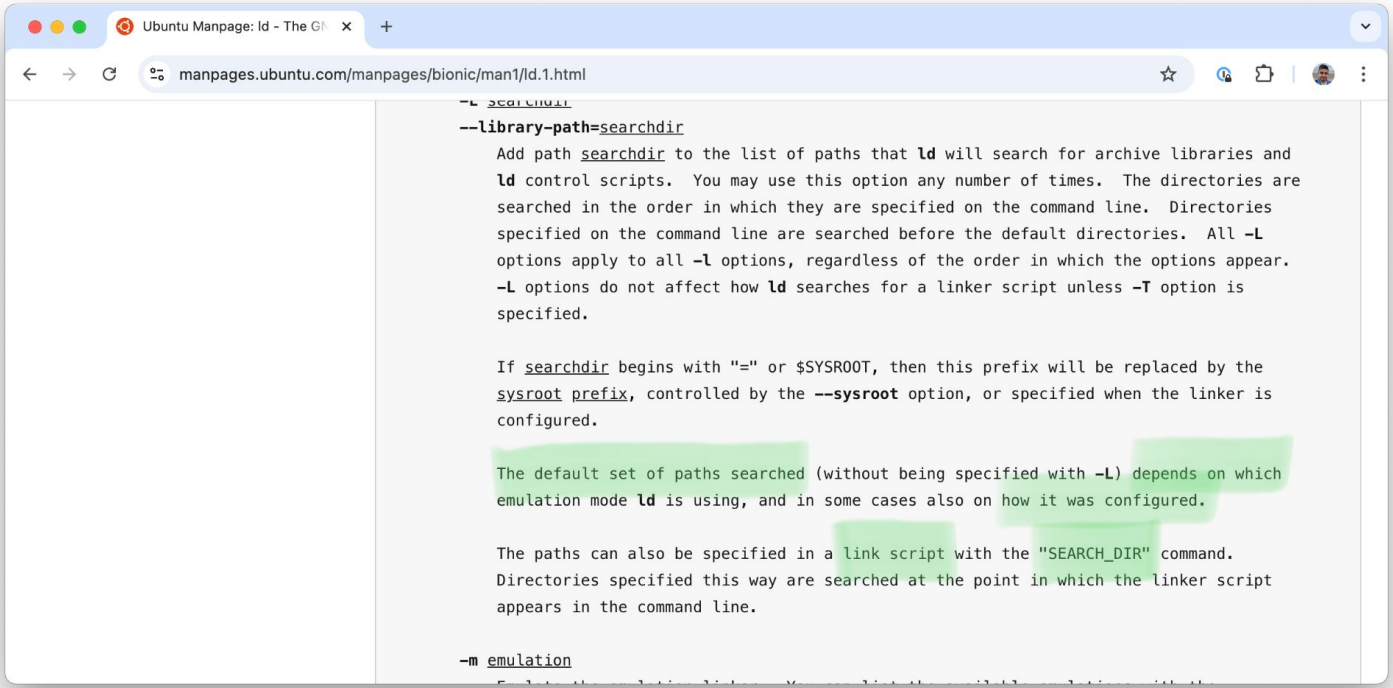
The linker [ld]

- Program is typically called ld
  - other implementations: ld.gold, lld, mold
- On Linux, typically invoked by `gcc` itself via the `collect2` tool

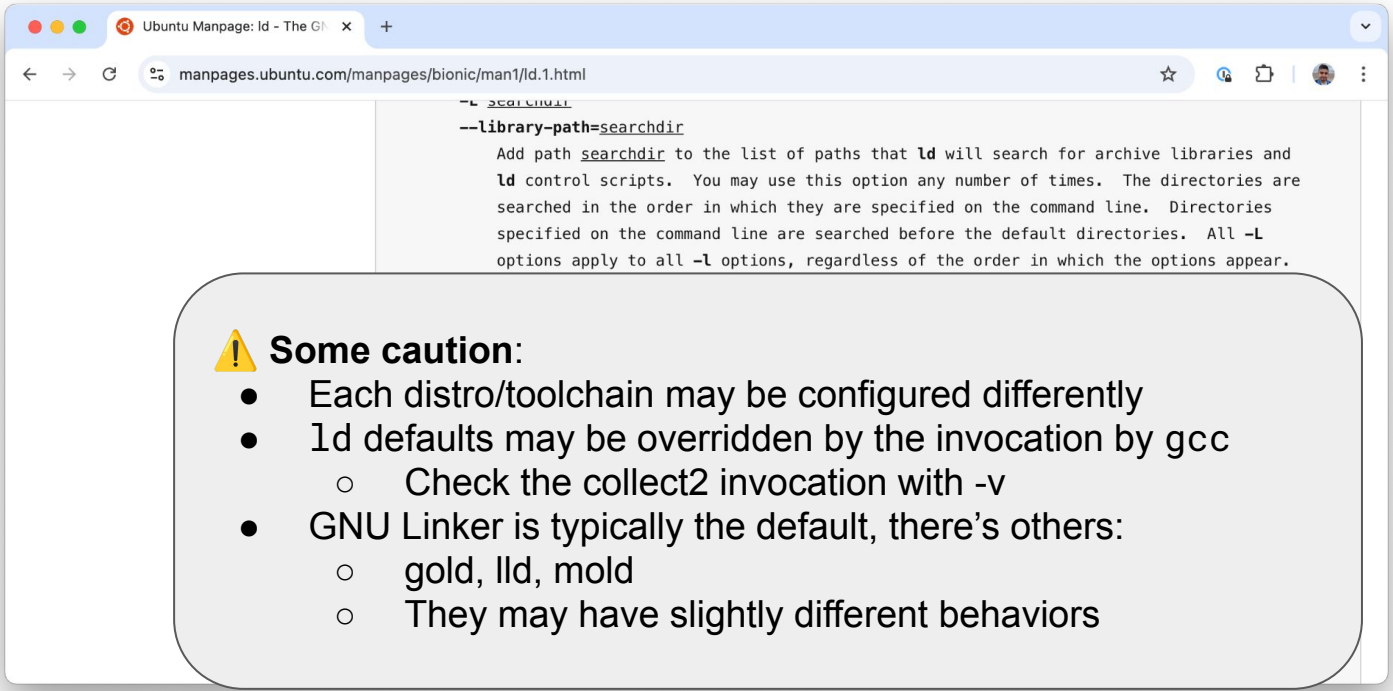
```
g++ my_program.cpp -o my_program -lfoo -v
```

```
4-linux-gnu/13/:/usr/lib/gcc/aarch64-linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/aarch64-linux-gnu/13/:/usr/lib/gcc/aarch64-linux-gnu/13/../../../aarch64-linux-gnu/:/usr/lib/gcc/aarch64-linux-gnu/13/../../../lib/:/lib/aarch64-linux-gnu/:/lib/../lib/:/usr/lib/aarch64-linux-gnu/:/usr/lib/../lib/:/usr/lib/gcc/aarch64-linux-gnu/13/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-o' 'my_program' '-v' '-shared-libgcc' '-mlittle-endian' '-mabi=lp64' '-dumpdir' 'my_program.'
 /usr/libexec/gcc/aarch64-linux-gnu/13/collect2 -plugin /usr/libexec/gcc/aarch64-linux-gnu/13/liblto_plugin.so -plugin-opt=/usr/libexec/gcc/aarch64-linux-gnu/13/lto-wrapper -plugin-opt=-fresolution=/tmp/ccKKJULW.res -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lgcc --build-id --eh-frame-hdr --hash-style=gnu --as-needed -dynamic-linker /lib/ld-linux-aarch64.so.1 -X -EL -maarch64linux --fix-cortex-a53-843419 -pie -z now -z relro -o my_program /usr/lib/gcc/aarch64-linux-gnu/13/../../../aarch64-linux-gnu/Scrt1.o /usr/lib/gcc/aarch64-linux-gnu/13/../../../aarch64-linux-gnu/crti.o /usr/lib/gcc/aarch64-linux-gnu/13/crtbeginS.o -L/usr/lib/gcc/aarch64-linux-gnu/13 -L/usr/lib/gcc/aarch64-linux-gnu/13/../../../aarch64-linux-gnu -L/usr/lib/gcc/aarch64-linux-gnu/13/../../../lib -L/lib/aarch64-linux-gnu -L/lib/../lib -L/usr/lib/aarch64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/aarch64-linux-gnu/13/../../.. /tmp/cceD6aXp.o -lfoo -lstdc++ -lm -lgcc_s -lgcc -lc -lgcc_s -lgcc /usr/lib/gcc/aarch64-linux-gnu/13/crtendS.o /usr/lib/gcc/aarch64-linux-gnu/13/../../../aarch64-linux-gnu/crtn.o
COLLECT_GCC_OPTIONS='-o' 'my_program' '-v' '-shared-libgcc' '-mlittle-endian' '-mabi=lp64' '-dumpdir' 'my_program.'
```

The linker [ld] - search procedure



The linker [ld] - search procedure



The linker [ld] - search procedure (cont'd)

```
g++ my_program.cpp -o my_program -lfoo -Xlinker --verbose

[...]
using internal linker script:
==================================================
/* Script for -pie -z combreloc -z relro -z now */
/* Copyright (C) 2014-2024 Free Software Foundation, Inc.
   Copying and distribution of this script, with or without modification,
   are permitted in any medium without royalty provided the copyright
   notice and this notice are preserved.  */
OUTPUT_FORMAT("elf64-littleaarch64", "elf64-bigaarch64",
          "elf64-littleaarch64")
OUTPUT_ARCH(aarch64)
ENTRY(_start)
SEARCH_DIR("=/usr/local/lib/aarch64-linux-gnu");
SEARCH_DIR("=/lib/aarch64-linux-gnu");
SEARCH_DIR("=/usr/lib/aarch64-linux-gnu");
SEARCH_DIR("=/usr/local/lib");
SEARCH_DIR("=/lib");
SEARCH_DIR("=/usr/lib");
SEARCH_DIR("=/usr/aarch64-linux-gnu/lib");
SECTIONS
[...]
```

The dynamic linker/loader [ld-linux.so]

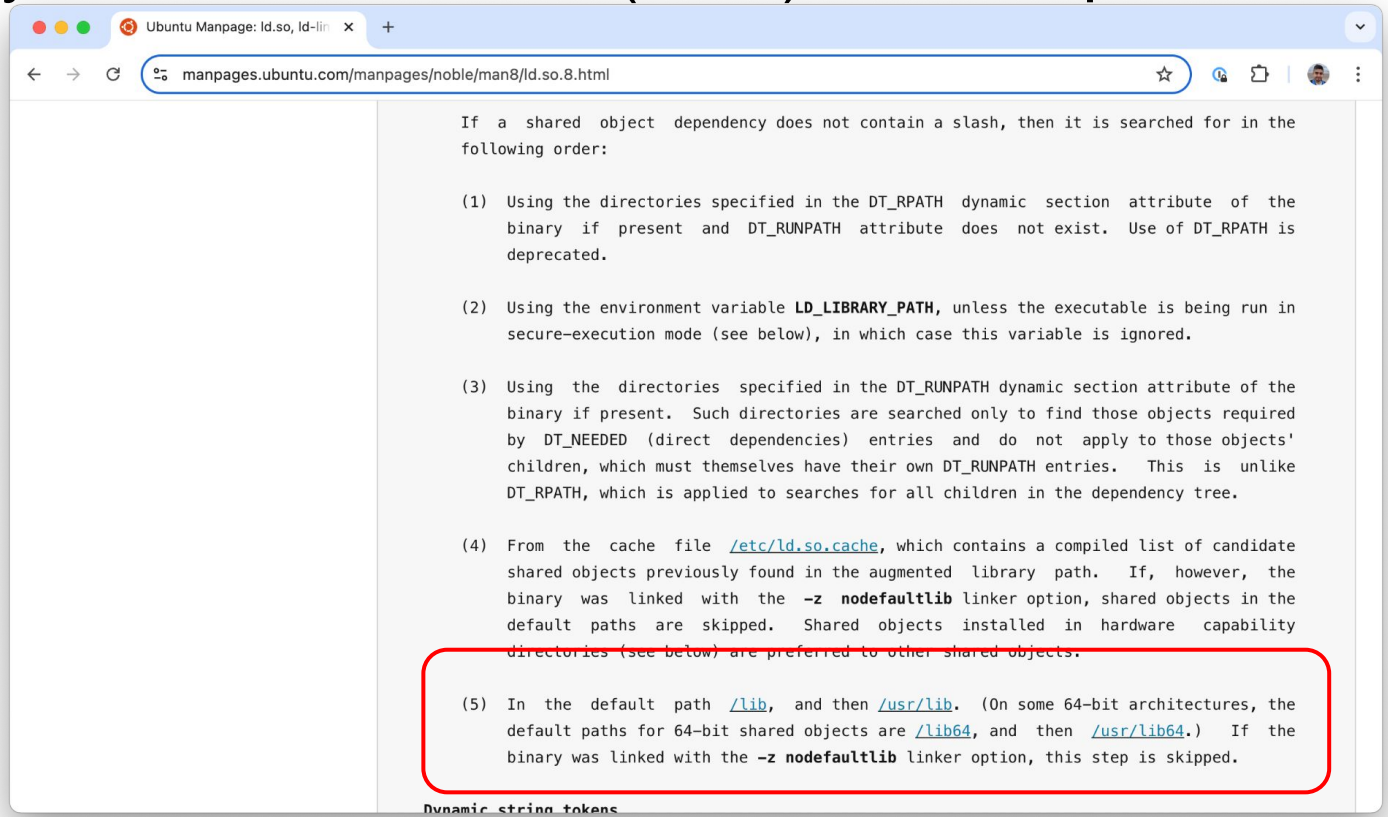
ELF executables are “interpreted” by an application: `ld-linux.so`

The full path to it is embedded in the executable ELF headers:

```
$ file my_program

my_program: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux-aarch64.so.1,
BuildID[sha1]=e9c0fb22d483b33ba9fac2b1574dbdf09e3421ce, for GNU/Linux 3.7.0, not
stripped
```

```
$  readelf -l my_program | grep interpreter
      [Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
```

The dynamic linker loader (ld.so) - search procedure



If a shared object dependency does not contain a slash, then it is searched for in the following order:

(1) Using the directories specified in the DT_RPATH dynamic section attribute of the binary if present and DT_RUNPATH attribute does not exist. Use of DT_RPATH is deprecated.

(2) Using the environment variable **LD_LIBRARY_PATH**, unless the executable is being run in secure-execution mode (see below), in which case this variable is ignored.

(3) Using the directories specified in the DT_RUNPATH dynamic section attribute of the binary if present. Such directories are searched only to find those objects required by DT_NEEDED (direct dependencies) entries and do not apply to those objects' children, which must themselves have their own DT_RUNPATH entries. This is unlike DT_RPATH, which is applied to searches for all children in the dependency tree.

(4) From the cache file /etc/ld.so.cache, which contains a compiled list of candidate shared objects previously found in the augmented library path. If, however, the binary was linked with the **-z nodefaultlib** linker option, shared objects in the default paths are skipped. Shared objects installed in hardware capability directories (see below) are preferred to other shared objects.

(5) In the default path /lib, and then /usr/lib. (On some 64-bit architectures, the default paths for 64-bit shared objects are /lib64, and then /usr/lib64.) If the binary was linked with the **-z nodefaultlib** linker option, this step is skipped.

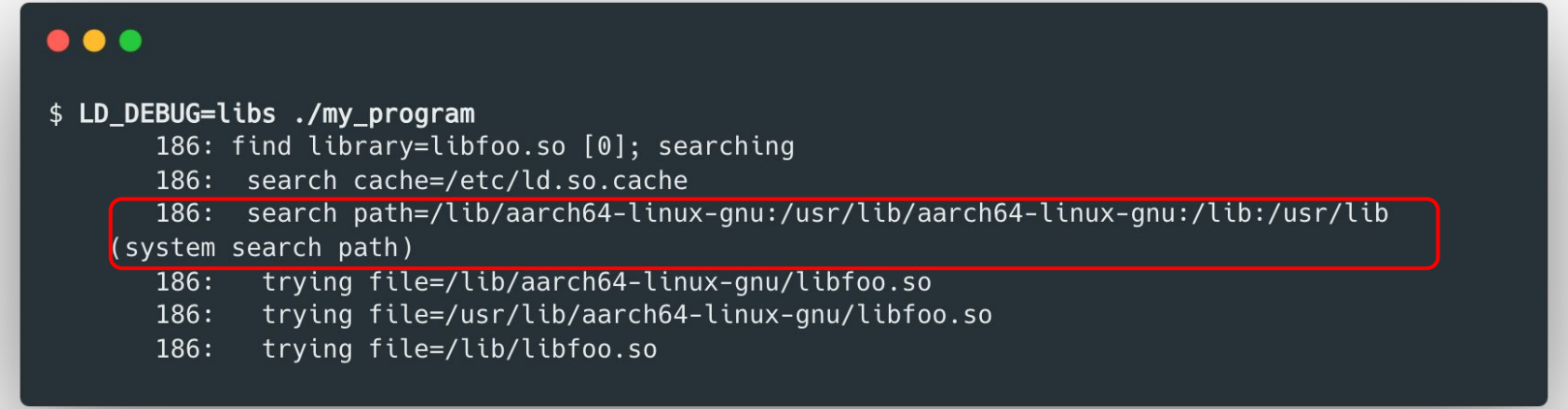
Dynamic string tokens

The dynamic linker loader (ld.so) - search procedure

```
$ objdump -p my_program | grep NEEDED
  NEEDED               libfoo.so
  NEEDED               libc.so.6
```

```
$ readelf -d my_program

Dynamic section at offset 0xfd90 contains 28 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libfoo.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 [...]
```

The dynamic linker loader (ld.so) - search procedure



The `ldd` tool

```
$ ldd my_program
        linux-vdso.so.1 (0x0000ffff98184000)
        libbar.so => /opt/foobar/lib/libbar.so (0x0000ffff980f0000)
        libstdc++.so.6 => /lib/aarch64-linux-gnu/libstdc++.so.6 (0x0000ffff97e70000)
        libc.so.6 => /lib/aarch64-linux-gnu/libc.so.6 (0x0000ffff97cb0000)
        libfoo.so => /opt/foobar/lib/libfoo.so (0x0000ffff97c80000)
        libm.so.6 => /lib/aarch64-linux-gnu/libm.so.6 (0x0000ffff97bd0000)
        /lib/ld-linux-aarch64.so.1 (0x0000ffff98147000)
        libgcc_s.so.1 => /lib/aarch64-linux-gnu/libgcc_s.so.1 (0x0000ffff97b90000)
```

Summary so far

Static linker [`ld`]

- Default search path
  - Default locations in `SEARCH_DIR` in linker script (pass -Xlinker –verbose to g++)
  - Compiler may be configured to pass additional locations (check -v output of g++)

Dynamic linker/loader [`ld-linux.so`]

- Default search path
  - Man ld.so
  - /etc/ld.so.conf.d/
  - (a cache)
- LD_DEBUG=libs ./executable (for search procedure)
- ldd ./executable (to list locations)

Linux example: library in non-default location

```
/opt/foo
|-- include
|   `-- foo.h
`-- lib
    `-- libfoo.so
```

Why?

- Library local to our project (not “installed”)
- We may be testing a different version of a library already installed in a system location
- Provided by package managers (Conan, vcpkg, Conda) …
- We may not have root privileges to write in system locations
- Library is vendored-in
- …

Linux example: library in non-default location

```
/opt/foo
|-- include
|   `-- foo.h
`-- lib
    `-- libfoo.so
```

```
g++ my_program.cpp -lfoo -o my_program
/usr/bin/ld: cannot find -lfoo: No such file or directory
collect2: error: ld returned 1 exit status
  return go(f, seed, [])
}
```

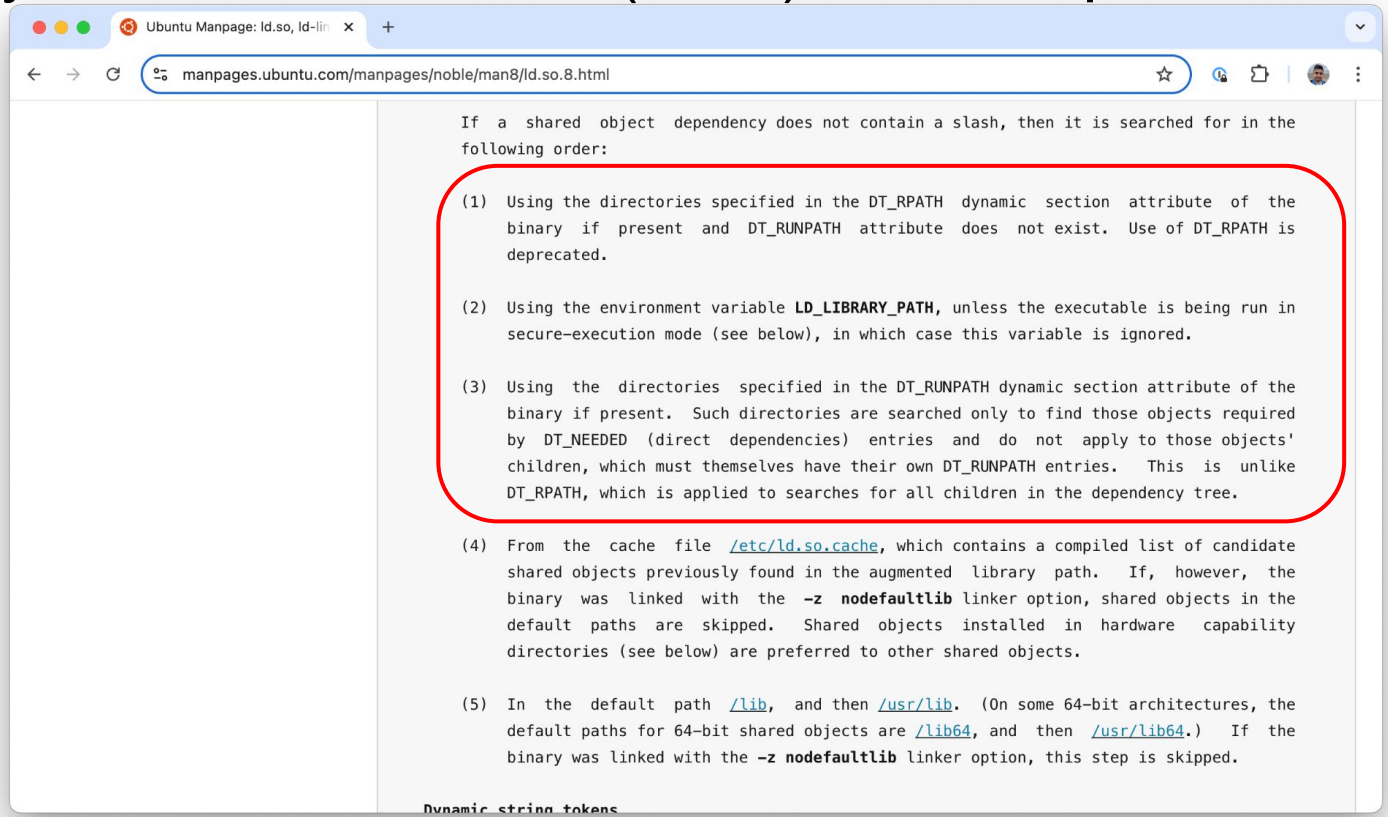
```
g++ my_program.cpp -L/opt/foo/lib -lfoo -o my_program
```

Linux example: library in non-default location

```
$ ./my_program
./my_program: error while loading shared libraries: libfoo.so: cannot open
shared object file: No such file or directory
```

The runtime linker/loader can not find the library now

The dynamic linker loader (ld.so) - search procedure



If a shared object dependency does not contain a slash, then it is searched for in the following order:

(1) Using the directories specified in the DT_RPATH dynamic section attribute of the binary if present and DT_RUNPATH attribute does not exist. Use of DT_RPATH is deprecated.

(2) Using the environment variable **LD_LIBRARY_PATH**, unless the executable is being run in secure-execution mode (see below), in which case this variable is ignored.

(3) Using the directories specified in the DT_RUNPATH dynamic section attribute of the binary if present. Such directories are searched only to find those objects required by DT_NEEDED (direct dependencies) entries and do not apply to those objects' children, which must themselves have their own DT_RUNPATH entries. This is unlike DT_RPATH, which is applied to searches for all children in the dependency tree.

(4) From the cache file /etc/ld.so.cache, which contains a compiled list of candidate shared objects previously found in the augmented library path. If, however, the binary was linked with the **-z nodefaultlib** linker option, shared objects in the default paths are skipped. Shared objects installed in hardware capability directories (see below) are preferred to other shared objects.

(5) In the default path /lib, and then /usr/lib. (On some 64-bit architectures, the default paths for 64-bit shared objects are /lib64, and then /usr/lib64.) If the binary was linked with the **-z nodefaultlib** linker option, this step is skipped.

Dynamic string tokens

Linux example: library in non-default location

```
$ LD_LIBRARY_PATH=/opt/foo/lib LD_DEBUG=libs ./my_program
       225:     find library=libfoo.so [0]; searching
       225:      search path=/opt/foo/lib              (LD_LIBRARY_PATH)
       225:       trying file=/opt/foo/lib/libfoo.so
       [...]
```

```
$ ld.so --library-path /opt/foo/lib ./my_program
```

Runtime linker/loader - rpath

- Runtime search path embedded in the ELF executable or library

```
g++ my_program.cpp -I/opt/foo/include -L/opt/foo/lib -lfoo -Wl,-rpath,/opt/foo/lib -o my_program
```

```
readelf -d my_program

Dynamic section at offset 0xfd80 contains 29 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libfoo.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000001d (RUNPATH)            Library runpath: [/opt/foo/lib]
```

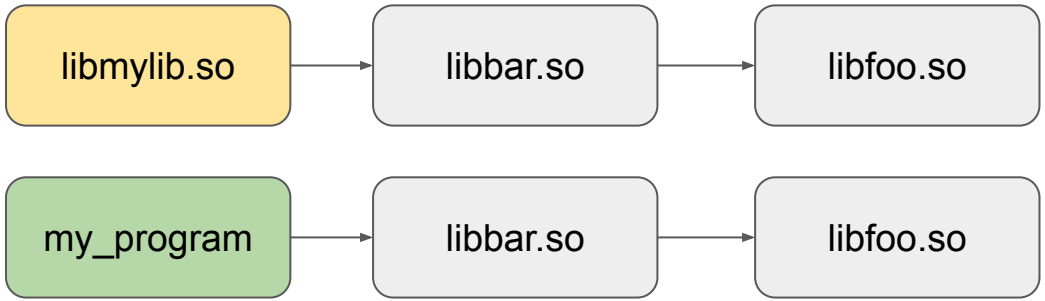
Runtime linker/loader - rpath

- Runtime search path embedded in the ELF executable or library
- The $ORIGIN keyword: the directory containing the executable (allows for relocatability)

```
$ LD_DEBUG=libs ./my_program
       255:     find library=libfoo.so [0]; searching
       255:      search path=/opt/foo/lib              (RUNPATH from file ./my_program)
       255:       trying file=/opt/foo/lib/libfoo.so
```

Linux example: transitive dependencies

```
/opt/foobar
|-- include
|   |-- bar.h
|   `-- foo.h
`-- lib
    |-- libbar.so
    `-- libfoo.so
```



Linux example: transitive dependencies

```
/opt/foobar
|-- include
|   |-- bar.h
|   `-- foo.h
`-- lib
    |-- libbar.so
    `-- libfoo.so
```


```
$ g++ my_lib.cpp -shared -o libmylib.so -I/opt/foobar/include
```

✅ ❓

Linux example: transitive dependencies

```
/opt/foobar
|-- include
|   |-- bar.h
|   `-- foo.h
`-- lib
    |-- libbar.so
    `-- libfoo.so
```


```
$ readelf -d libmylib.so

Dynamic section at offset 0xfe00 contains 24 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libstdc++.so.6]
 [...]
```

⚠️ No recorded dependency on libfoo.so!

Linux example: transitive dependencies


```
$ g++ my_lib.cpp -shared -o libmylib.so -I/opt/foobar/include -L/opt/foobar/lib -lbar
```

Linux example: transitive dependencies


```
$ readelf -d libmylib.so

Dynamic section at offset 0xfdf0 contains 25 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libbar.so]
 0x0000000000000001 (NEEDED)             Shared library: [libstdc++.so.6]
 [...]
```

Linux example: transitive dependencies

```
/opt/foobar
|-- include
|   |-- bar.h
|   `-- foo.h
`-- lib
    |-- libbar.so
    `-- libfoo.so
```

An executable!

⚠️ ???

```
$ g++ my_program.cpp -o my_program -I/opt/foobar/include -L/opt/foobar/lib -lbar
/usr/bin/ld: warning: libfoo.so, needed by /opt/foobar/lib/libbar.so, not found (try using
-rpath or -rpath-link)
/usr/bin/ld: /opt/foobar/lib/libbar.so: undefined reference to `i_am_foo()'
collect2: error: ld returned 1 exit status
```

Linux example: transitive dependencies

```
/opt/foobar
|-- include
|   |-- bar.h
|   `-- foo.h
`-- lib
    |-- libbar.so
    `-- libfoo.so
```


```
$ g++ my_program.cpp -o my_program -I/opt/foobar/include -L/opt/foobar/lib -lbar -Wl,-rpath,/opt/foobar/lib
```

Linux example: transitive dependencies

--allow-shlib-undefined
--no-allow-shlib-undefined
Allows or disallows undefined symbols in shared libraries. This switch is similar to
--no-undefined except that it determines the behaviour when the undefined symbols are
in a shared library rather than a regular object file. It does not affect how
undefined symbols in regular object files are handled.

The default behaviour is to report errors for any undefined symbols referenced in shared libraries if the linker is being used to create an executable, but to allow them if the linker is being used to create a shared library.

Linux example: transitive dependencies

```
   -L searchdir
       --library-path=searchdir
           Add path searchdir to the list of paths that ld will search for archive libraries
and
           ld control scripts.  You may use this option any number of times.  The directories
are
           searched in the order in which they are specified on the command line.  Directories
           specified on the command line are searched before the default directories.  A  ll -L
           options apply to all -l options, regardless of the order in which the options
appear.
           -L options do not affect how ld searches for a linker script unless -T option is
           specified.
```

Linux example: transitive dependencies

```
/opt/foobar
|-- include
|   |-- bar.h
|   `-- foo.h
`-- lib
    |-- libbar.so
    `-- libfoo.so
```


```
readelf -d my_program

Dynamic section at offset 0xfd70 contains 30 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libbar.so]
 0x0000000000000001 (NEEDED)             Shared library: [libstdc++.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000001d (RUNPATH)            Library runpath: [/opt/foobar/lib]
```

Linux example: transitive dependencies

```
/opt/foobar
|-- include
|   |-- bar.h
|   `-- foo.h
`-- lib
    |-- libbar.so
    `-- libfoo.so
```


```
$ ./my_program
./my_program: error while loading shared libraries: libfoo.so: cannot open
shared object file: No such file or directory
```

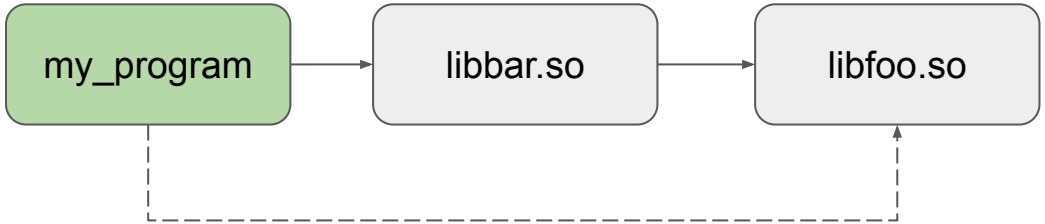
Linux example: RPATH vs RUNPATH

- `RUNPATH` only applies to direct dependencies.
- `RPATH` (legacy) can be used for indirect dependencies too
  - ⚠️ Problem: `RPATH` has priority over `LD_LIBRARY_PATH`

Possible solutions here:

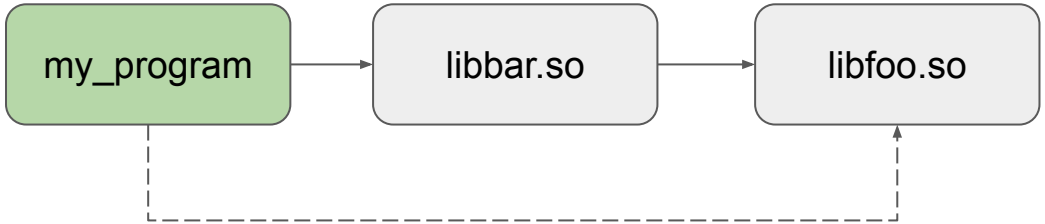
- Use the old ELF tag instead (pass `-Wl,--disable-new-dtags`)
- Embed a `RUNPATH` to `libbar.so`
  - When creating it:
  - After the fact: use `patchelf`
- Use `LD_LIBRARY_PATH` at runtime

Example 4: executable makes call to function in indirect library



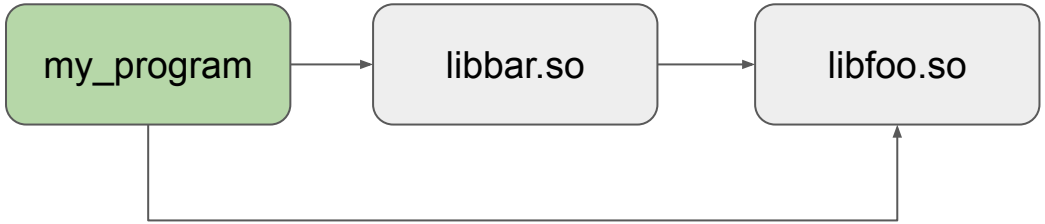

```
$ g++ my_program.cpp -o my_program -I/opt/foobar/include -L/opt/foobar/lib -lbar -Wl,-rpath,/opt/foobar/lib
```

Example 4: call to function in (indirect) transitive library




```
$ g++ my_program.cpp -o my_program -I/opt/foobar/include -L/opt/foobar/lib -lbar -Wl,-rpath,/opt/foobar/lib
/usr/bin/ld: /tmp/ccG1fIWE.o: undefined reference to symbol '_Z15function_in_foov'
/usr/bin/ld: /opt/foobar/lib/libfoo.so: error adding symbols: DSO missing from command line
collect2: error: ld returned 1 exit status
```

Example 4: call to function in (indirect) transitive library




```
$ g++ my_program.cpp -o my_program -I/opt/foobar/include -L/opt/foobar/lib -lbar -lfoo
```

Apple OS platforms

macOS summary

- (static) linker: ld
  - `ld64`
  - `ld-prime` (available since Xcode 15)
- Dynamic linker/loader: `dyld`
- File extensions:
  - dylib, so, framework, (tbd)
- File inspection:
  - otool, vtool
- Post-build modifications:
  - install_name_tool

macOS static linker [ld] search path

- Similar to Linux: use -L for additional search paths
- Differences:
  - Only /usr/local/lib is searched by default in the local computer
  - System libraries are in the SDK
  - Frameworks have their own search paths

```
$ clang++ my_program.cpp -o my_program -lfoo -Xlinker -v
@(#)PROGRAM:ld PROJECT:ld-1053.12
BUILD 10:15:51 Mar 29 2024
[...]
Library search paths:
        /usr/local/lib
        /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/lib
        /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/lib/swift
Framework search paths:
        /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/System/Library/Frameworks
```

Mach-o binaries

`otool`: object file displaying tool

`-L`: display shared library dependencies

Dynamic library identification

*Every dynamic library has an **install name**. This name identifies the library to the dynamic linker [dyld]. When you link to a dynamic library, the static linker [ld] records the library's install name in your Mach-O image [executable, dynamic library]*

Dynamic Library Identification, Apple Developer Forums

https://developer.apple.com/forums/thread/736719

Dynamic library identification (cont’d)

Install name needs to be provided at link time.
Alternatively, after being built

```
clang++ libfoo.cpp -o libfoo.dylib -shared -Wl,-install_name,@rpath/libfoo.dylib
```

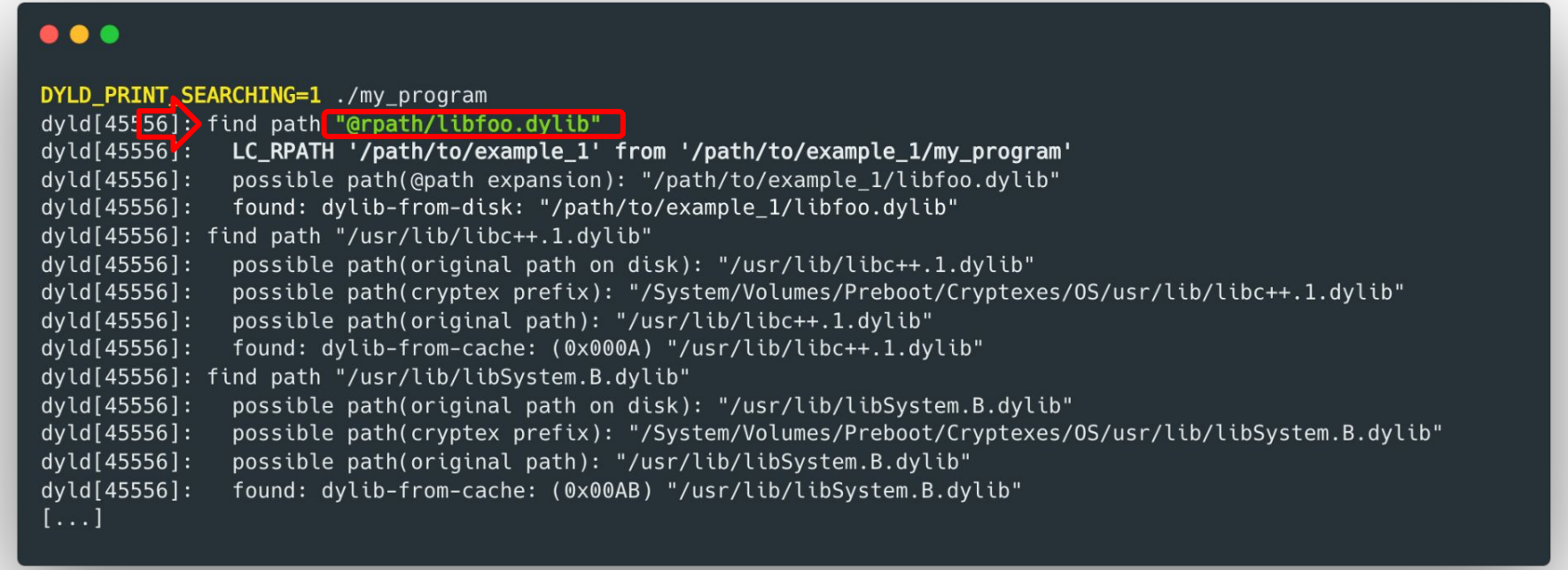
```
install_name_tool -id @rpath/libfoo.dylib libfoo.dylib
```

Dynamic loader [dyld] search procedure

Will depend on install name of dependency:

- `/usr/lib/libc++.1.dylib`
- `@rpath/libfoo.dylib`
  - `Search in LC_RPATH entries in Mach-O header`
  - RPATH is a stack of directories built from LC_RPATH entries of the chain that led to the current dyld load
    - Start at the executable
    - Similar to legacy DT_RPATH on Linux
- `libfoo.dylib` (leaf name)
  - For historical reasons, but no longer recommended on macOS

Dynamic loader [dyld] search procedure (cont'd)



Dynamic loader [dyld] search procedure (cont'd)

```
$ clang++ my_program.cpp -o my_program -L/path/to/foo -lfoo -Wl,-rpath,/path/to/foo
```

```
$ install_name_tool -add_rpath /path/to/lib my_program
```

Dynamic loader [dyld] search procedure (cont'd)

```
$ otool -l my_program
[...]
Load command 13
          cmd LC_LOAD_DYLIB
      cmdsize 48
         name @rpath/libfoo.dylib (offset 24)
   time stamp 2 Wed Dec 31 17:00:02 1969
      current version 0.0.0
compatibility version 0.0.0
Load command 14
          cmd LC_LOAD_DYLIB
      cmdsize 48
         name /usr/lib/libc++.1.dylib (offset 24)
   time stamp 2 Wed Dec 31 17:00:02 1969
      current version 1700.255.5
compatibility version 1.0.0
Load command 15
          cmd LC_LOAD_DYLIB
      cmdsize 56
         name /usr/lib/libSystem.B.dylib (offset 24)
   time stamp 2 Wed Dec 31 17:00:02 1969
      current version 1345.120.2
compatibility version 1.0.0
Load command 16
          cmd LC_RPATH
      cmdsize 80
         path /path/to/libfoo/ (offset 12)
[...]
```

Indirect dependencies / missing symbols


```
$ g++ my_lib.cpp -shared -o libmylib.so -I/opt/foobar/include
```

Linux ✅ ❓

Indirect dependencies / missing symbols


```
clang++ libbar.cpp -I./libfoo/include -shared -o libbar.dylib
Undefined symbols for architecture arm64:
  "i_am_foo()", referenced from:
      i_am_bar() in libbar-f688bb.o
ld: symbol(s) not found for architecture arm64
```

macOS: error

Indirect dependencies / missing symbols

-undefined *treatment*
Specifies how undefined symbols are to be treated. Options are: error, warning, suppress, or dynamic_lookup. The default is error.

`ld` manpages on macOS

The two level namespace

Mach-O uses a two-level namespace. When a Mach-O image imports a symbol, it references the symbol name and the library where it expects to find that symbol.

An Apple Library Primer, Apple Developer Forums

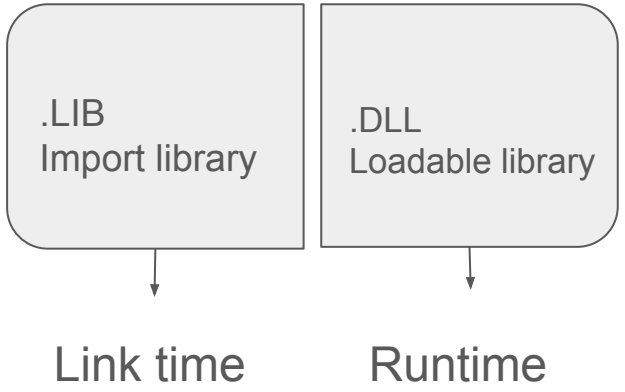
https://developer.apple.com/forums/thread/736719

Windows

Dynamically Loaded Libraries (DLL)

File format:

- PE: portable executable
- Shared libraries are “split” in two files:
  - **`.lib`** - import library, used at link time
    - Not to be confused with “.lib” static libraries
  - **`.dll`** - loaded at runtime
    - .pyd is used for loadable python modules



Tools:

- `cl.exe` - MSVC compiler
- `link.exe` - Linker
- `dumpbin.exe` - binary file dumper

`Link.exe` search path

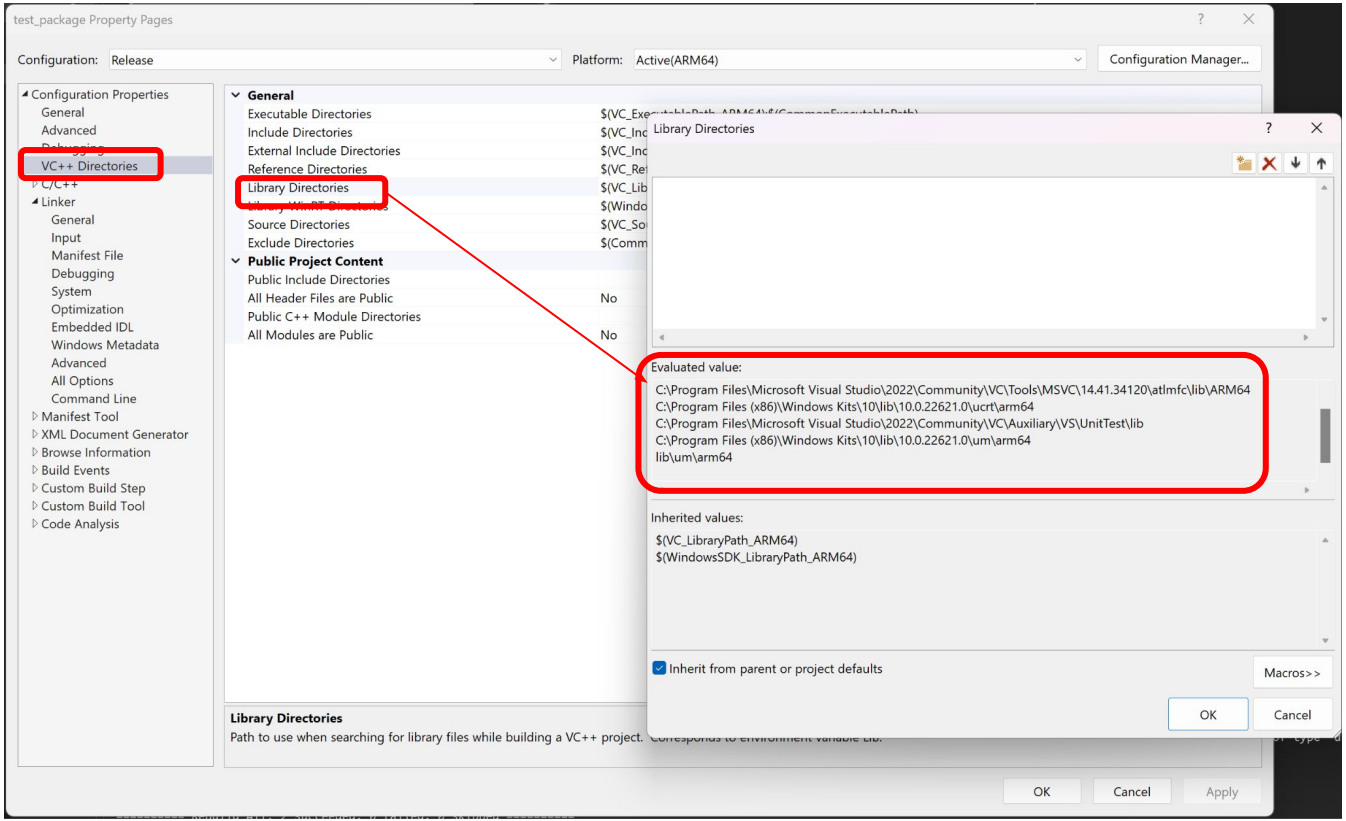
- Link.exe can be invoked by the compiler (cl.exe)
  - but typically build systems (CMake, MSBuild) will invoke link.exe directly
- Linker inputs:
  - Library names, e.g. `foo.lib`
  - File paths, e.g. `C:/path/to/foo.lib`
- Linker search path:
  - Content of **`LIB`** environment variable
  - **`/LIBPATH` (prepend paths)**

`Link.exe` default search path

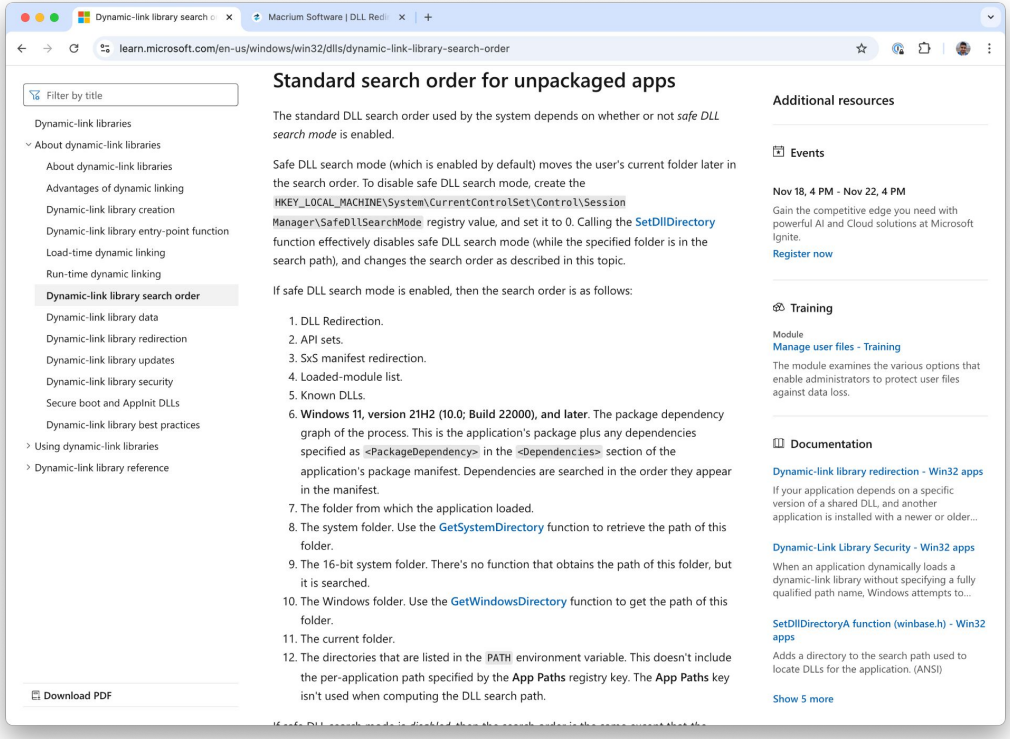
```
> echo %LIB%
LIB=C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.41.34120\ATLMFC\lib\ARM64;
    C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.41.34120\lib\ARM64;
    C:\Program Files (x86)\Windows Kits\10\lib\10.0.22621.0\ucrt\arm64;
    C:\Program Files (x86)\Windows Kits\10\\lib\10.0.22621.0\\um\arm64
```

- Set by Visual Studio Tools Command Prompt (“vcvars”)
- Depends on specific toolset and Windows SDK
- Pass **`/VERBOSE:LIB`** to print search procedure

`Link.exe` default search path



The DLL search path at runtime



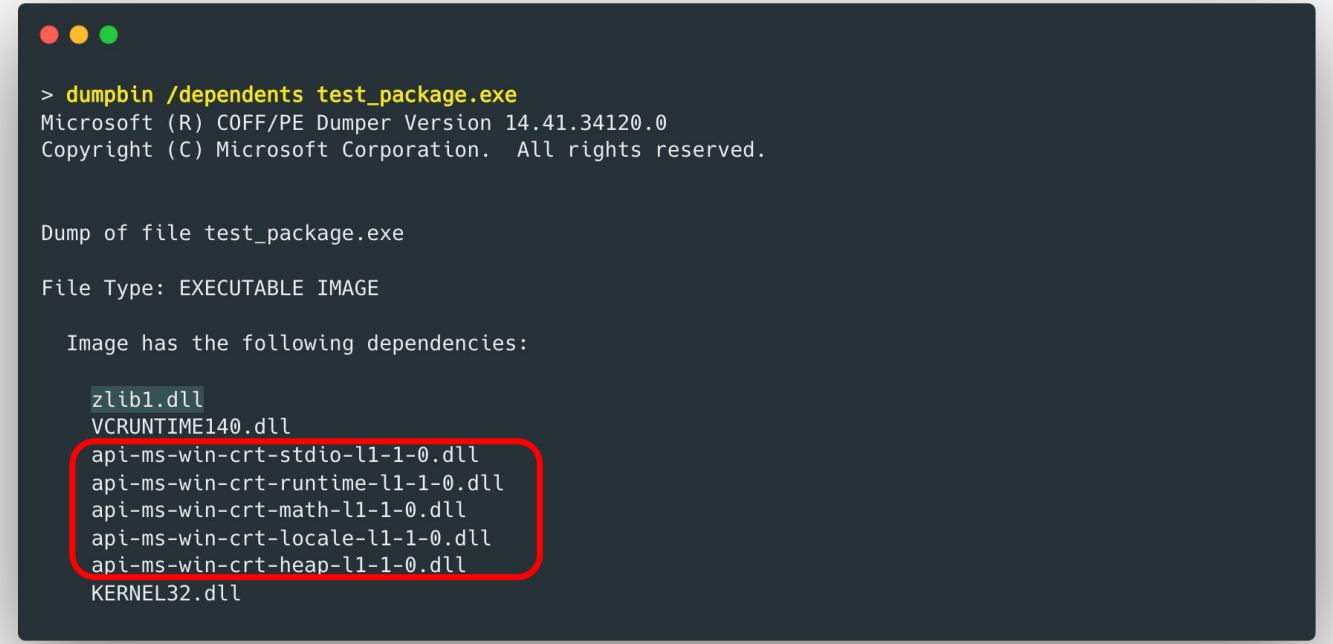
- Typically:
  - Next to the .exe file
  - System directory
  - Entries in PATH environment variable

Dynamic-link library search order

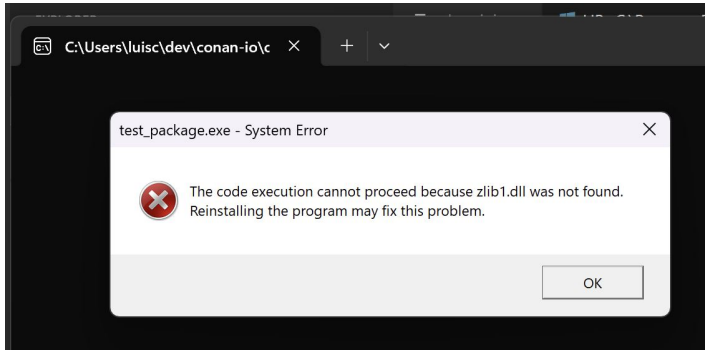
Microsoft Learn

https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order

Listing dependencies



DLL not found



Launched from File explorer

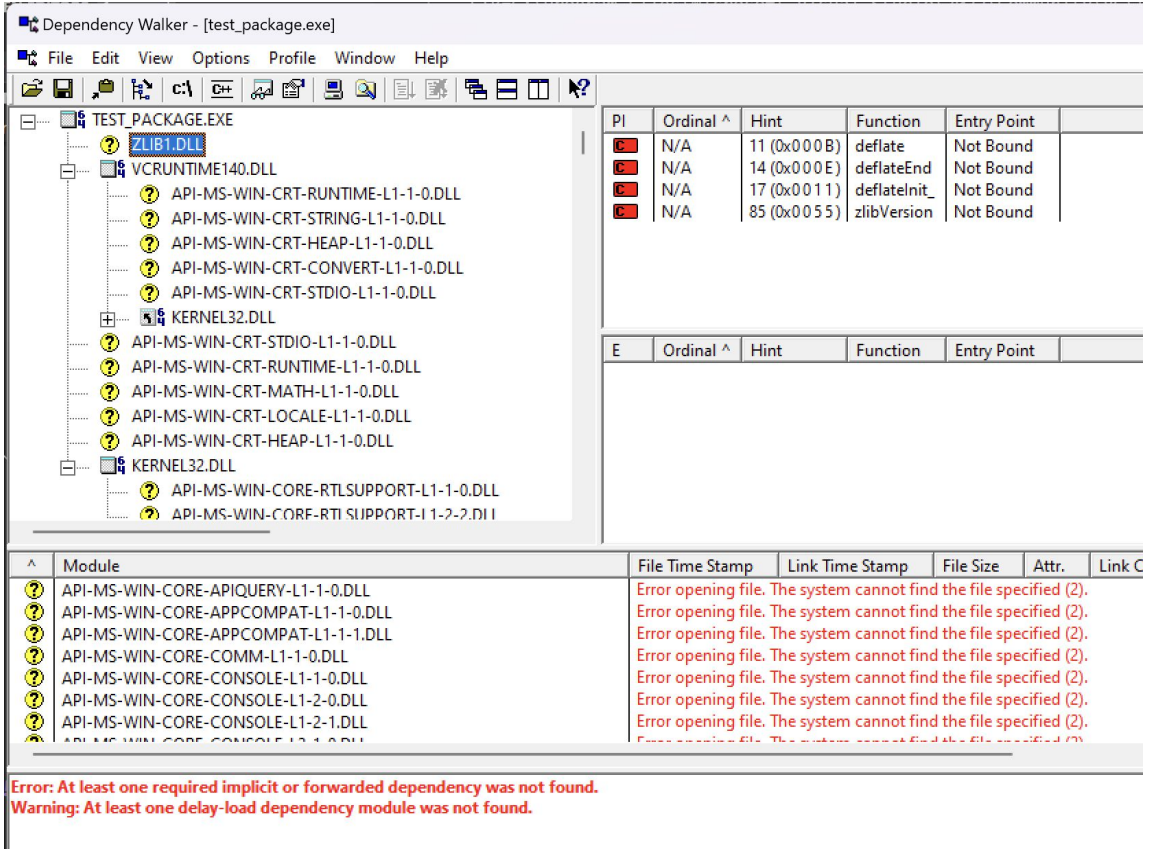
Launched from command prompt

```
C:\Users\luisc\dev>test_package.exe

C:\Users\luisc\dev>echo %errorlevel%
-1073741515
```

DLL troubleshooting

Our options are limited:

- Dependency walker
  - “depends.exe”
- Windows debugger



Build Systems

Build systems: Why locate libraries?

The linker already tells us if it can't find a library.

We could just do `-lopencv_core` and let the linker fail if the library can't be found.

But we may see things like the following in our build scripts:

- `find_package(OpenCV)`
- `PKG_CHECK_MODULES([OPENCV], [opencv >= 2.0])`

Why locate libraries? (cont’d)

- Error out early
  - Saves time!
- Additional checks
  - Is it the right version?
  - Is it the right architecture? Configured properly?
- Portability/cross platform for abstraction: filenames are different!
- Correct compiler and linker flags:
  - Compiler definitions
  - Paths to non-default locations: Include paths, linker paths (-L, -rpath-link)

Locating libraries - build system approaches

Try and link a “probe” executable during the configuration phase 😕

⇨ OK for “system” libraries with well known names
👎 brings runtime loader into the “build” mix

“Find” the library file ahead of time

⇨ 👎 requires knowing library (file) names ahead of time
👎 need to replicate the linker search path (varies across systems/distros)

Package descriptor files to derive compiler/linker flags

⇨ Provides better abstraction, only need to know a generic “package” name

Requires a convention to search for these files… and their names!

Package descriptor files

pkg-config .pc files

CMake [package]-config.cmake: imported targets

Common Package Specification

Package descriptor files

Package descriptor files and where to find them

Coming to a C++ conference near you in 2025

CMake: Build time vs runtime

- CMake help on platforms that support RPATH:
  - macOS, Linux and other ELF platforms

"Build tree"

`cmake --build`

CMake passes `-rpath` to linker - for paths it knows about!

All linked binaries have absolute paths - this is why we can run binaries from local builds

"Install tree"

`cmake --install`

CMake patches the RPATHs, either removing them, or replacing them with the developers choice (relative RPATH or path to install location)

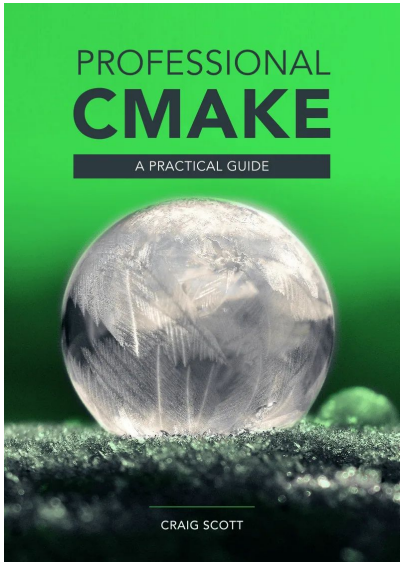
CMake: Build time vs runtime - Windows

On Windows we’re not so lucky.

- When launching executables from the build folder, we need DLLs
  - Either in the same folder as the executables
  - Or in the PATH environment variable
- Problems:
  - DLLs of dependencies in non-default locations
  - Projects that place DLLs and executables in different subfolders
  - Won’t be able to run, debug, run tests

CMake: Build time vs runtime - Windows

```
# In this example, Algo is assumed to be a shared library defined elsewhere in
# the project and whose binary will be in a different directory to test_Algo
add_executable(test_Algo ...)
add_test(NAME CheckAlgo COMMAND test_Algo)

if(WIN32)
   # Ensure the required DLLs can be found at runtime
   set(algoDir "$<SHELL_PATH:$<TARGET_FILE_DIR:Algo>>")
   set(otherDllDir "C:\\path\\to\\another\\dll")
   set_property(TEST CheckAlgo APPEND PROPERTY
       ENVIRONMENT_MODIFICATION
           PATH=path_list_prepend:${algoDir}
           PATH=path_list_prepend:${otherDllDir}
   )
endif()
```



Snippet from “Professional CMake: A Practical Guide” by Craig Scott

CMake: Build time vs runtime - Windows

```
find_package(foo CONFIG REQUIRED) # package generated by install(EXPORT)

add_executable(exe main.c)
target_link_libraries(exe PRIVATE foo::foo foo::bar)
add_custom_command(TARGET exe POST_BUILD
 COMMAND ${CMAKE_COMMAND} -E copy -t $<TARGET_FILE_DIR:exe>
$<TARGET_RUNTIME_DLLS:exe>
 COMMAND_EXPAND_LISTS
)
```

CMake documentation for generator expressions

CMake: Build time vs runtime - Windows

Package managers to the rescue!

Conan CMake integrations:

- Command line: the `conanrun.bat` (PATH env var)
- Visual Studio generators (requires CMake 3.27 or higher)
  - Conan sets CMAKE_VS_DEBUGGER_ENVIRONMENT
- Optional:
  - Copy all DLLs to build folder - generate() method during conan install
  - `CONAN_RUNTIME_LIB_DIRS` - defined in toolchain

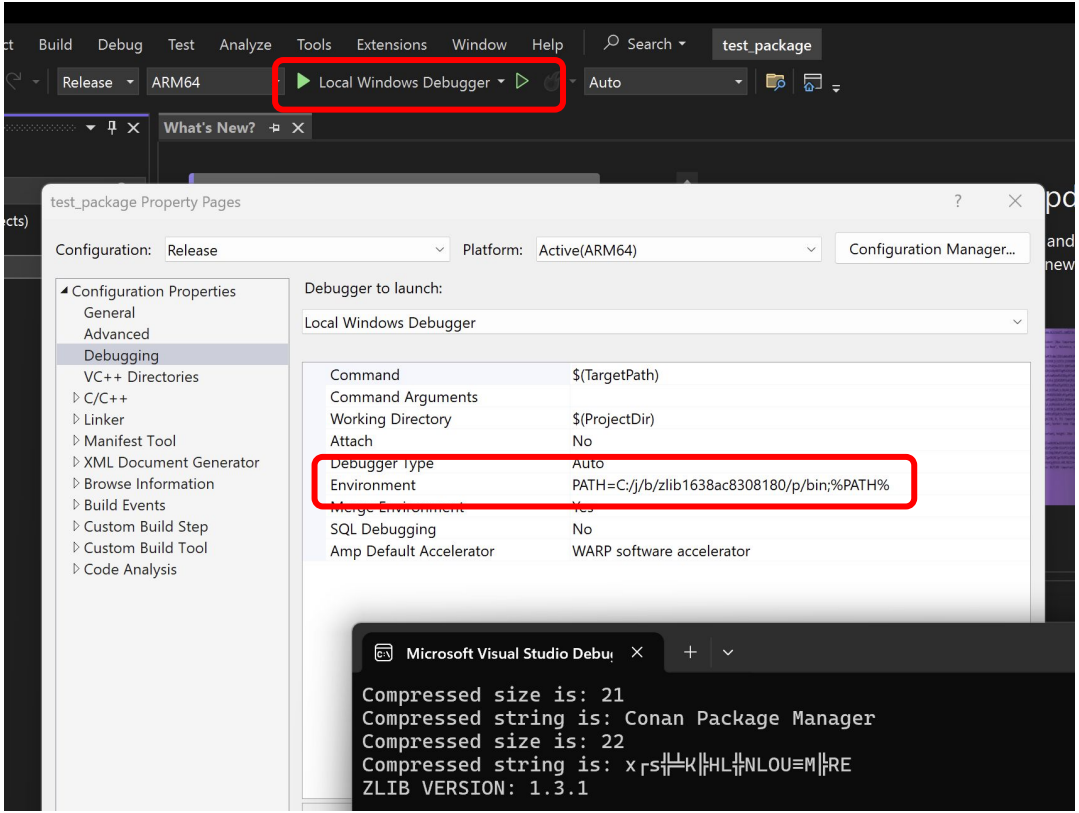
vcpkg:

- The add_executable built-in function is overridden to copy DLL dependencies right next to the executable

CMake: Build time vs runtime - Windows


```
C:\Users\dev\my_project>conan install . -o "*:shared=True"

C:\Users\dev\my_project>cmake --preset conan-default

C:\Users\dev\my_project>cmake --build --preset conan-release

C:\Users\dev\my_project>build\generators\conanrun.bat

C:\Users\dev\my_project>build\Release\test_package.exe

Compressed size is: 21
Compressed string is: Conan Package Manager
Compressed size is: 22
Compressed string is: x▯s▯▯K▯HL▯NLOU▯M▯RE
ZLIB VERSION: 1.3.1
```

CMake: Build time vs runtime - Windows



Creating relocatable bundles

Installers, packages, bundles

- Goal: build on one machine, run in another
- Application must find all its shared library dependencies when launched

Who provides the shared library dependencies	Where the application lives once installed
System package manager	Fixed location
Bundled with the application	Arbitrary location (chosen at install time)

Installers, packages, bundles

- Goal: build on one machine, run in another
- Application must find all its shared library dependencies when launched

Who provides the shared library dependencies	Where the application lives once installed	
System package manager	Fixed location	Typical Linux
Bundled with the application	Arbitrary location (chosen at install time)	macOS, Windows…

Linux system package managers

- Linux users like these: JustWorks™
- External shared library dependencies are provided by other packages
  - We just need to package our executable files (and first-party libraries)
  - And record dependencies on other packages
- But has some downsides:
  - Dependencies available and versions are decided for us
  - Software vendors have to provide different packages for each distro (and each version!)
  - Some choose to host their own servers, users need to set them up first
  - Linux only!

Bundling dependencies

- Bundle the shared libraries needed by our executable(s)
- Give the runtime loader a helping hand:
  - ELF platforms, Apple platforms: Relative RPATH ($ORIGIN, @executable_path)
  - Place DLLs next to executable on Windows

First we need to determine what to bundle!

System runtime libraries

First-party shared libraries

Third-party shared libraries

We typically don’t bundle these (or can’t)

Bundle formats

- Could be a folder with executable and libraries
- Or specific formats:
  - macOS App bundles (.app)
  - AppImage on Linux
  - Android APK

Gathering shared libraries for bundles

Linux: we can rely on the linker loader to gives us a list (e.g. using `ldd`)

- Based on the output, determine if the library can/should be copied:
  - E.g. exclude system libraries from `/usr/lib`

Some examples:

- `linuxdeployqt`
  - (No longer maintained as of 2024)

Downsides:

- Mostly Linux and platforms that have ldd
- Uses the actual loader, won't work for cross-built binaries

Gathering shared libraries for bundles

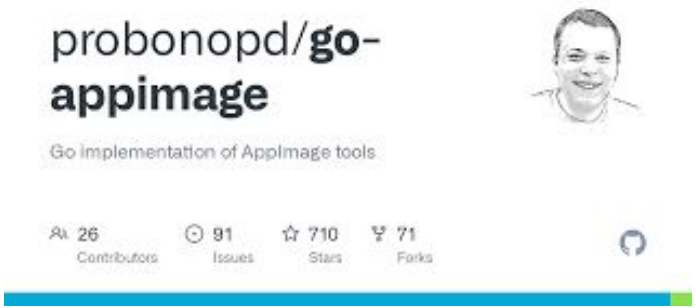
README.md | appdirtool.go

src > appimagetool > appdirtool.go > getDirsFromSoConf

```
1114
1115  func findLibrary(filename string) (string, error) {
1116
1117      // Look for libraries in commonly used default locations
1118      locs := []string{"/usr/lib64", "/lib64", "/usr/lib", "/lib",
1119          "/usr/lib/x86_64-linux-gnu/libfakeroot",
1120          "/usr/local/lib",
1121          "/usr/local/lib/x86_64-linux-gnu",
1122          "/lib/x86_64-linux-gnu",
1123          "/usr/lib/x86_64-linux-gnu",
1124          "/lib32",
1125          "/usr/lib32"}
1126      for _, loc := range locs {
1127          libraryLocations = helpers.AppendIfMissing(libraryLocations, filepath.Clean(loc))
1128      }
1129
1130      // Additionally, look for libraries in the same locations in which glibc ld.so looks for libraries
1131      if helpers.Exists("/etc/ld.so.conf") {
1132          locs := getDirsFromSoConf("/etc/ld.so.conf")
1133          for _, loc := range locs {
1134              libraryLocations = helpers.AppendIfMissing(libraryLocations, filepath.Clean(loc))
1135          }
1136      }
1137
1138      // Also look for libraries in in LD_LIBRARY_PATH
1139      ldpstr := os.Getenv("LD_LIBRARY_PATH")
1140      ldps := strings.Split(ldpstr, ":")
1141      for _, ldp := range ldps {
1142          if ldp != "" {
1143              libraryLocations = helpers.AppendIfMissing(libraryLocations, filepath.Clean(ldp))
1144          }
1145      }
1146
```

Alternatively: replicate the search logic of the runtime loader

Example from `go-appimage`:



Gathering shared libraries for bundles

```
install(TARGETS my_executable
  RUNTIME_DEPENDENCY_SET my_app_deps
)
install(RUNTIME_DEPENDENCY_SET my_app_deps
   PRE_EXCLUDE_REGEXES
       [[api-ms-win-.*]]
       [[ext-ms-.*]]
       [[kernel32\.dll]]
       [[libc\.so\..*]] [[libgcc_s\.so\..*]] [[libm\.so\..*]] [[libstdc\+\+\.so\..*]]
   POST_EXCLUDE_REGEXES
       [[.*/system32/.*\.dll]]
       [[^/lib.*]]
       [[^/usr/lib.*]]
)
```

Docker

- A container is a process that is isolated from the rest of the system
- Which includes the filesystem
- When an application requests files via system calls:
  - The ELF interpreter, e.g. /lib/ld-linux-aarch64.so.1
  - Runtime dependencies
- The kernel creates an illusion around the process

If everything else fails…

… just link everything statically

Questions?

Shared Libraries and Where to Find Them

CONAN
C/C++ Package Manager

JFrog

Luis Caro Campos
R&D Team Lead, JFrog