

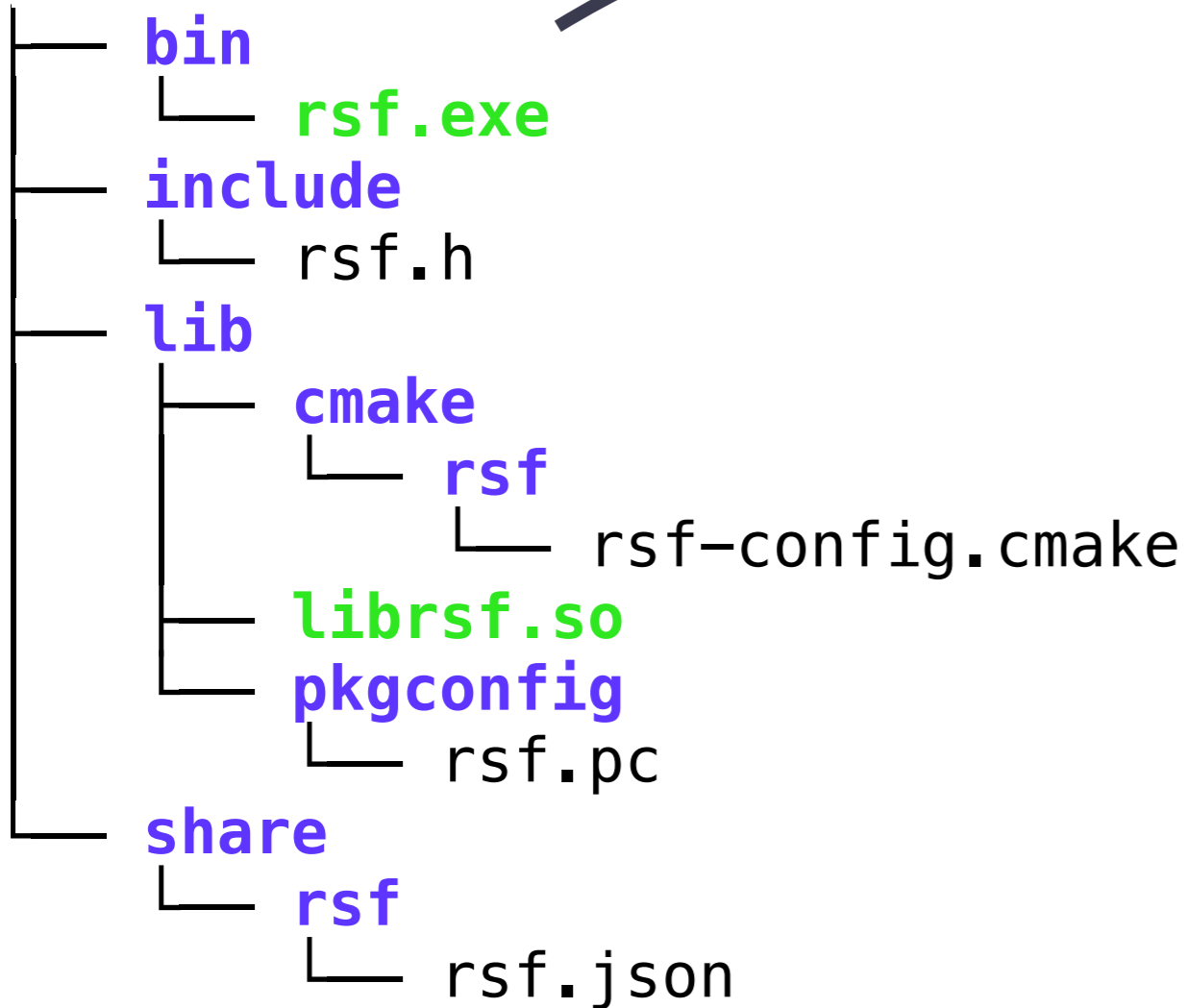
# Helping C/C++ Packages be Relocatable

WARWICK  
THE UNIVERSITY OF WARWICK

Ben Morgan

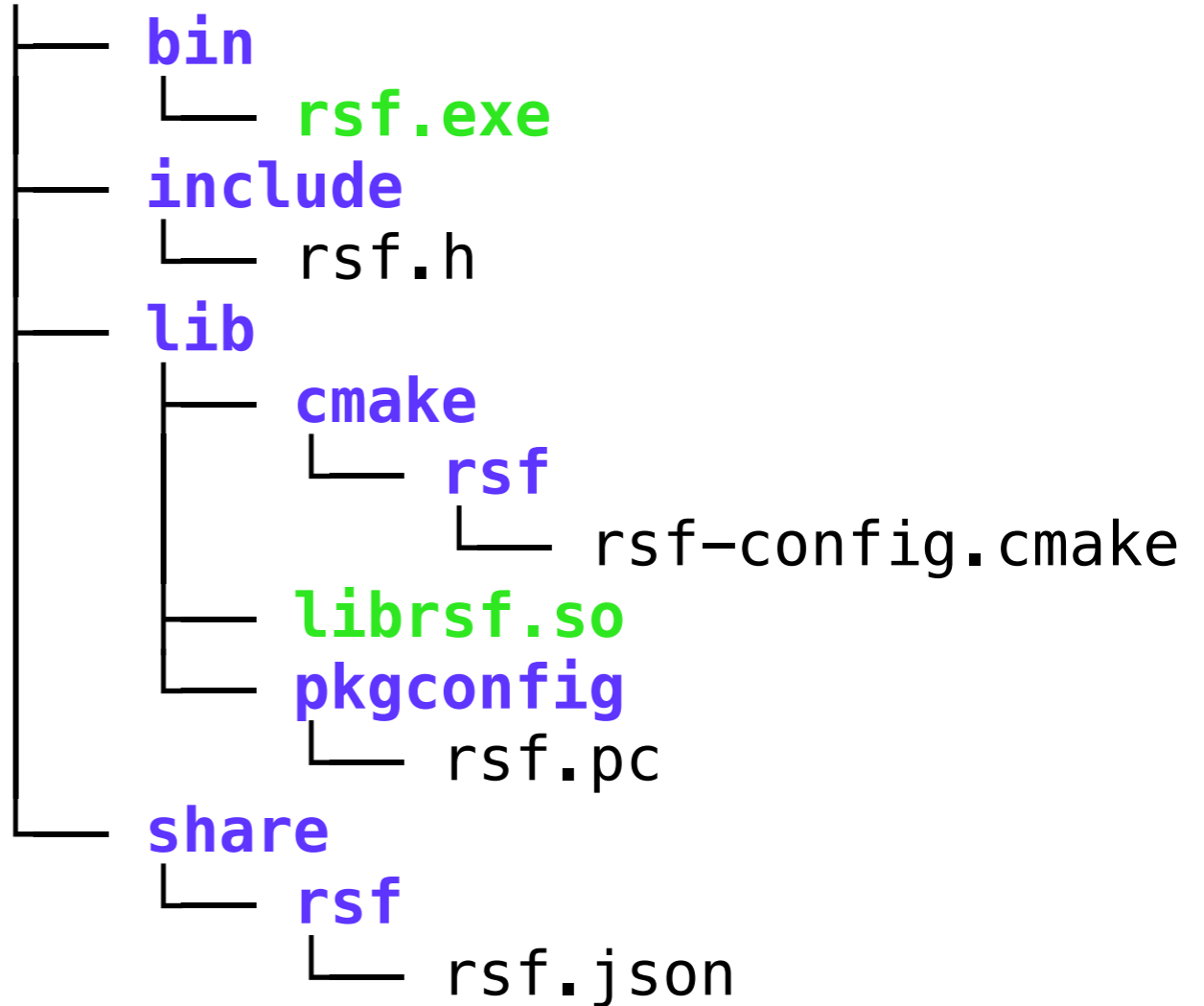
Copy/Link/Move

**/A/Prefix**



```
$ /A/Prefix/bin/rsf.exe  
rsf.json = {"foo" : "bar"}
```

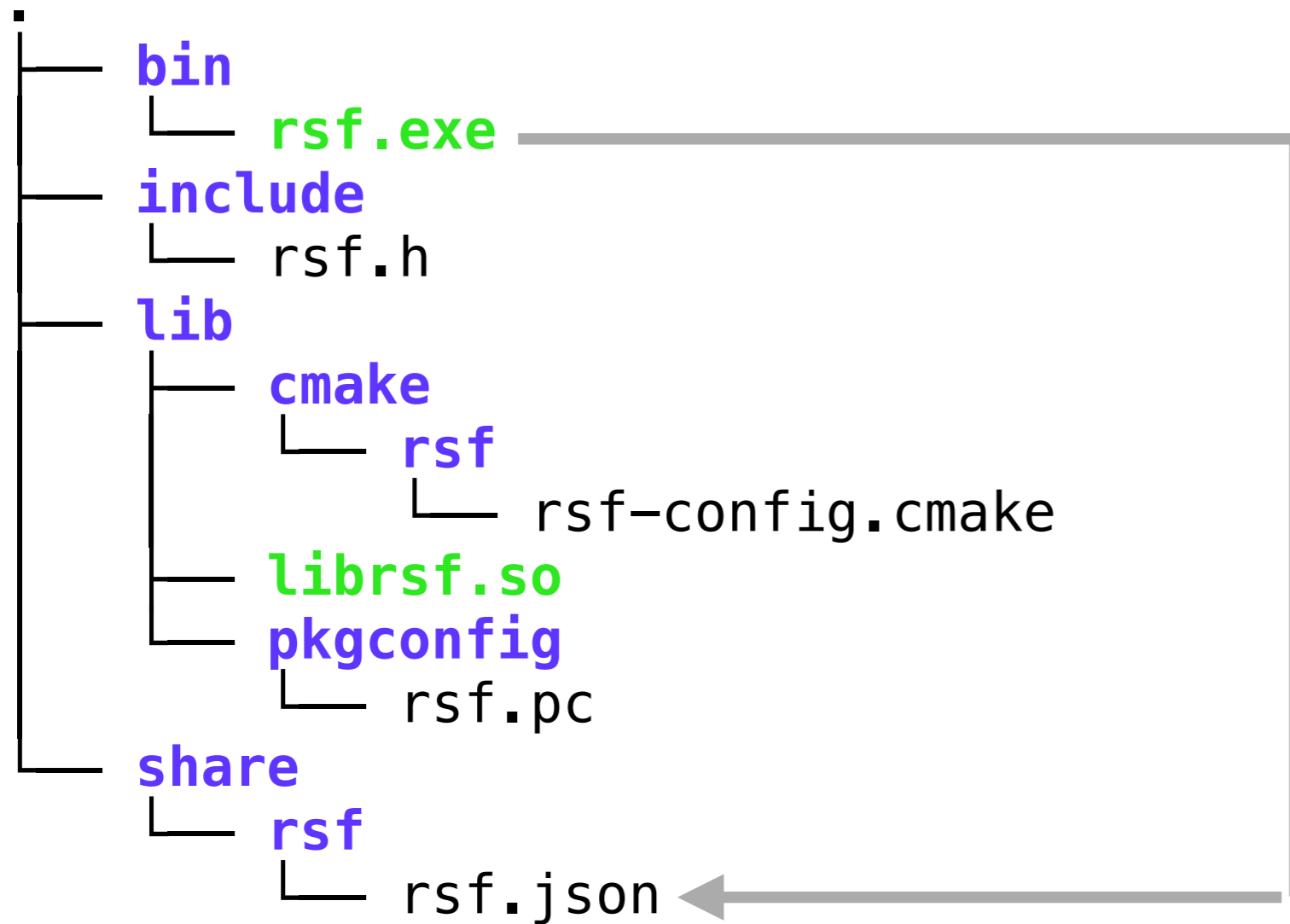
**/Another/Dir**



```
$ /Another/Dir/bin/rsf.exe  
rsf.json = {"foo" : "bar"}
```

# What is Relocatability?

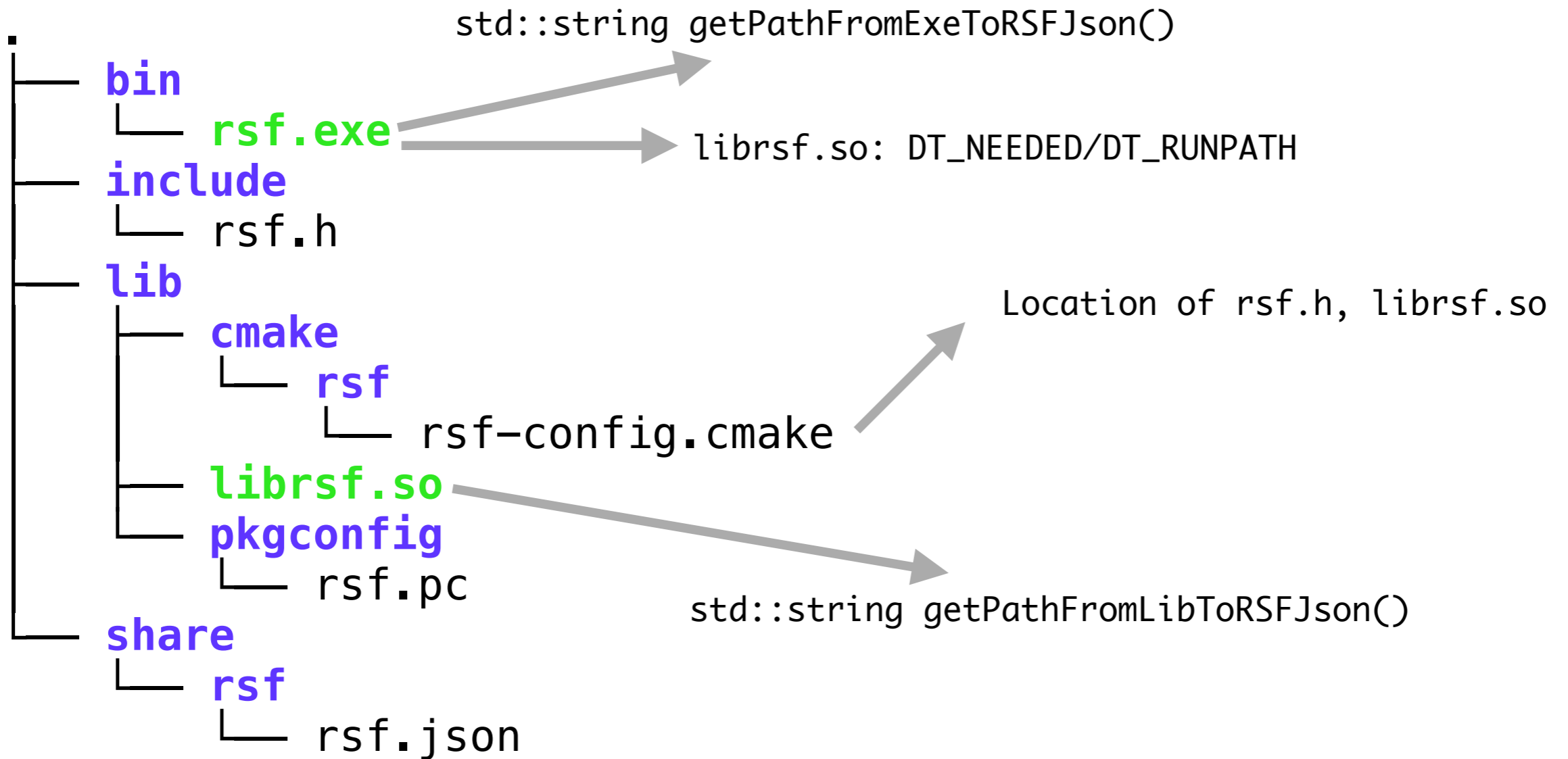
Package can be moved across filesystem and be used with no changes to itself or environment



Path to rsf.json  
 =  
 Dir holding rsf.exe  
 +  
 ../share/rsf

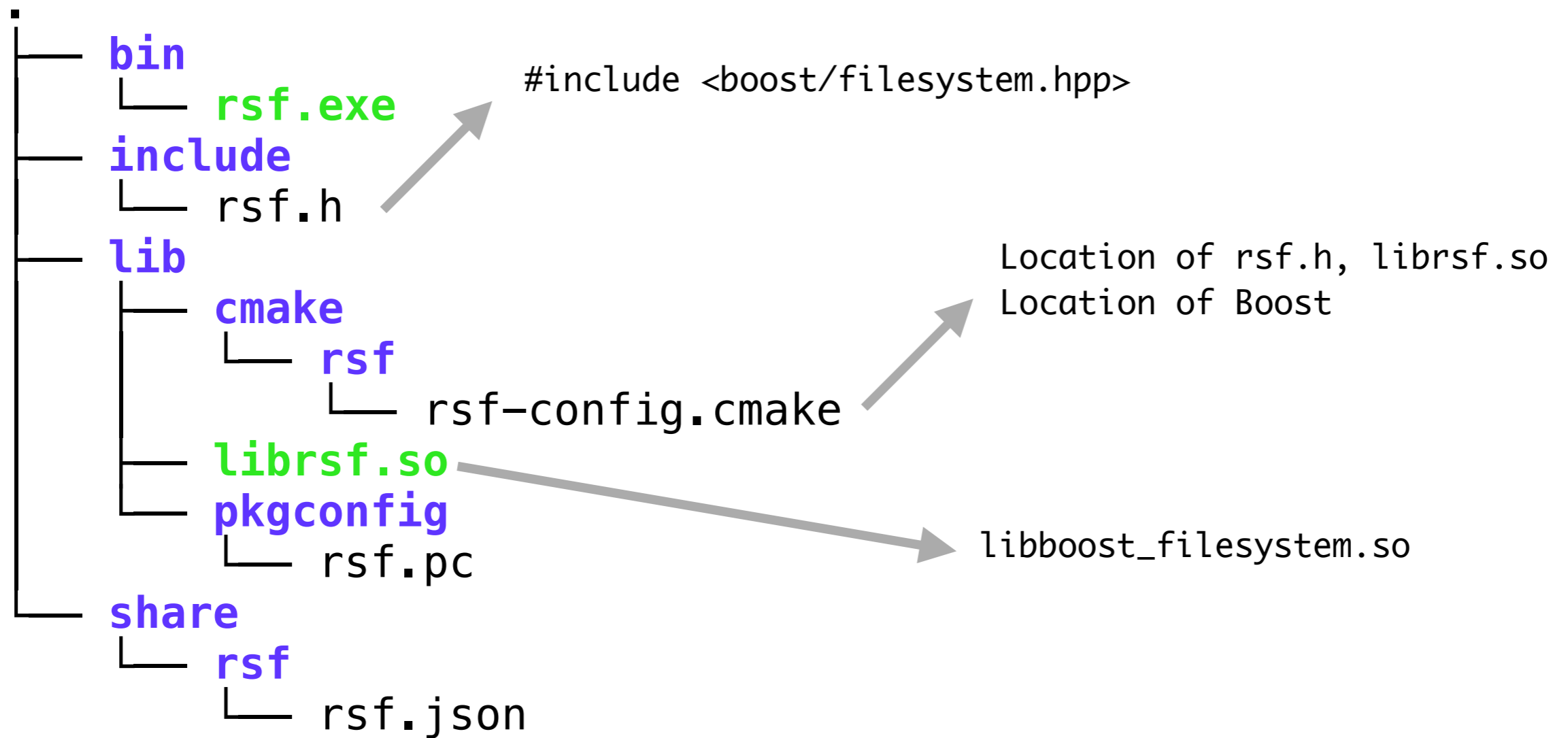
## Relocatability ~ Self-Location + Relative Paths

A file needs to know where it is, and the relative path from that to what it needs.



# Self-Location Needs

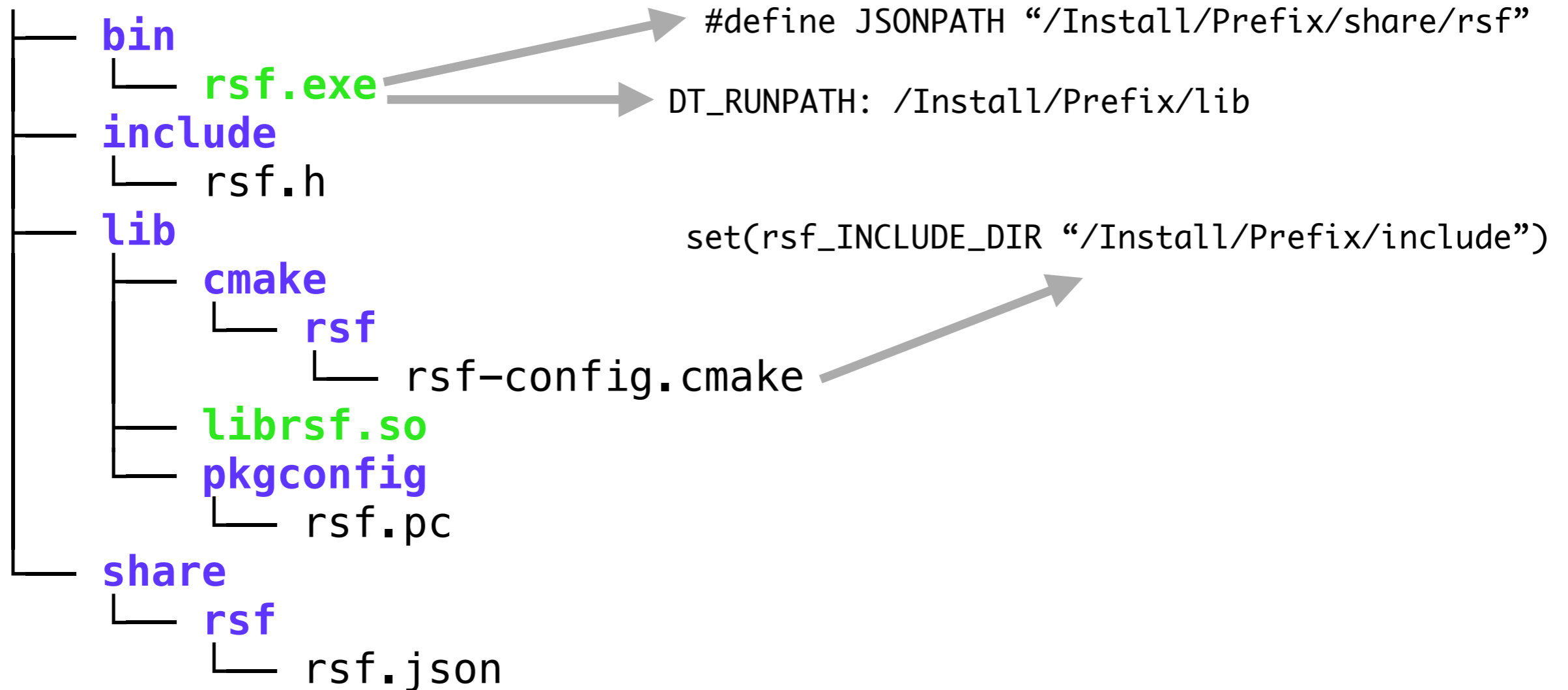
“File Introspection” covers both scripts, binaries, and executable format



## What About Dependencies?

Non-trivial packages use/  
expose others. Cannot rely on  
relative paths anymore...

# /Install/Prefix



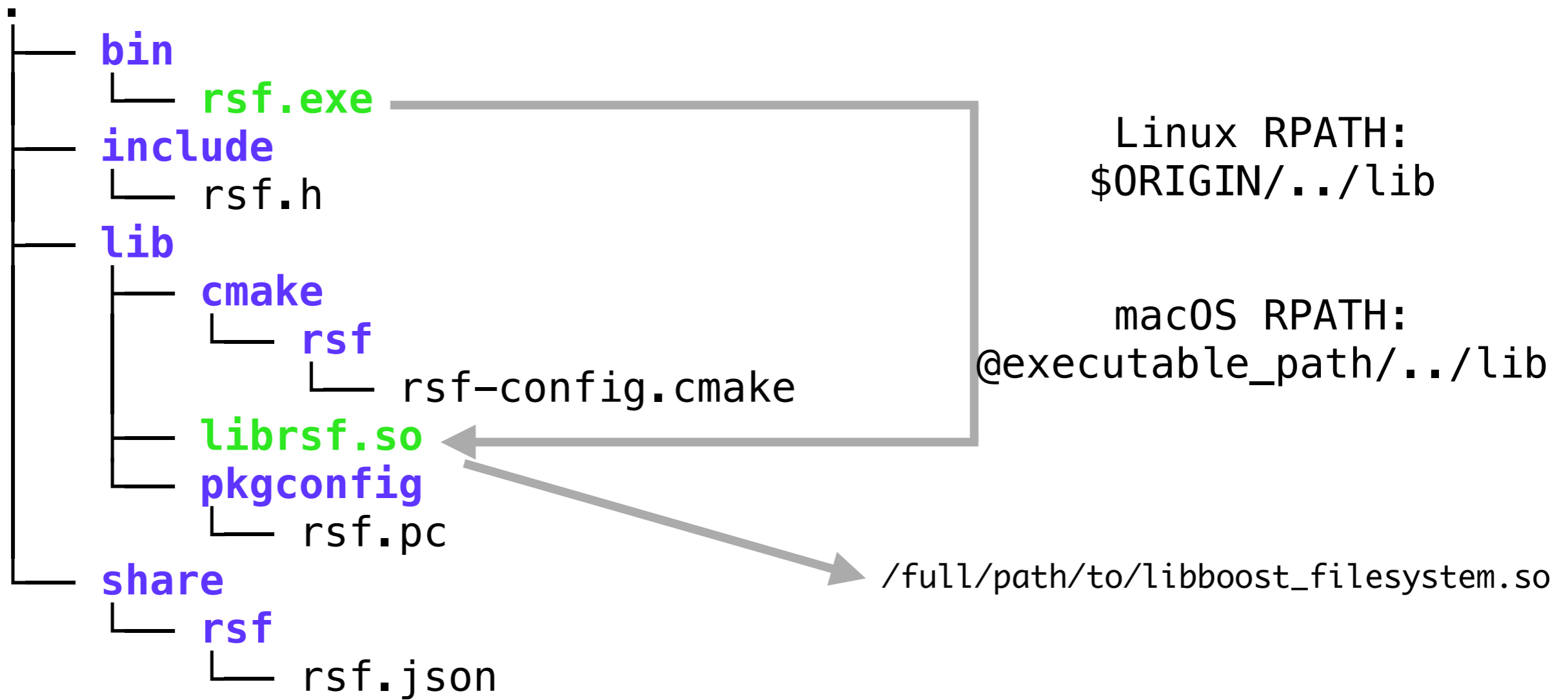
## Breaking Relocatability

Typically by hardcoding absolute paths in binaries, scripts and RPATHS.

# Requirements for Relocatability

---

- How can a C/C++ program/library find its filesystem location at runtime?
- How can scripts for CMake/PkgConfig find their location when used?
- How to handle external dependencies?
- How to fix existing non-relocatable packages?
- **We have, or know, solutions to some already!**

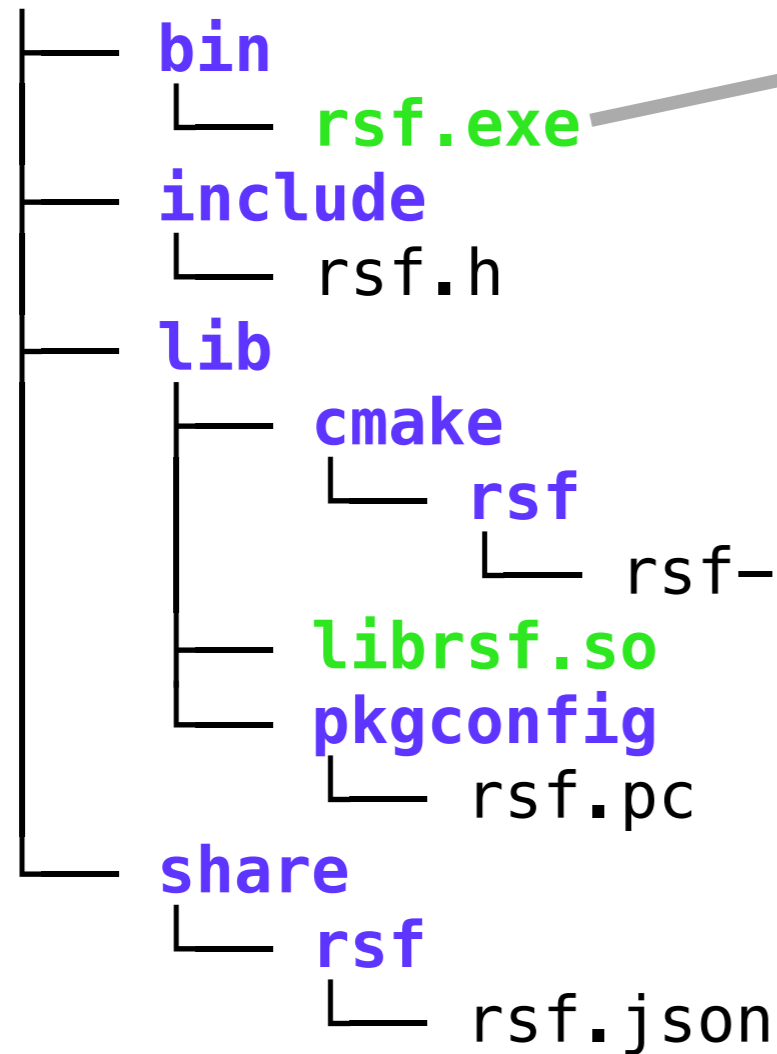


# RPATH/RUNPATH et al.

Internal: \$ORIGIN etc  
 External: Full RUN/RPATH  
 Packaging: patchelf update



# /New/Prefix



#define JSONPATH "/Old/Prefix/share/rsf"

#define JSONPATH "/New/Prefix/share/rsf"

set(rsf\_INCLUDE\_DIR "/Old/Prefix/include")

set(rsf\_INCLUDE\_DIR "/New/Prefix/include")

## Absolute Path Patching

Spack can patch paths in text files. Conda can also patch paths in binaries if New < Old

# Enabling Relocatability in our Packages

---

- How can a C/C++ program/library find its filesystem location at runtime?
  - ***Caveat: very basic usage, and use case dependent!***
- How can scripts for CMake/PkgConfig find their location when used, how to refind external libraries?
  - ***Caveat: "Modern" CMake has a nominally recommended way to do this, but may want a balance between config management (CMAKE\_PREFIX\_PATH) and full paths in a development environment.***

# argv[0]?

---

- Not guaranteed to provide a full or relative path to the program
- Can be arbitrary, or name of first found in PATH
- Might also be symlink
- Can use a series of checks to resolve these, assuming no program/system changes to PWD, PATH or links

# Binreloc for Programs

---

- Set of C functions that locate executable via:
- /proc on Linux/BSD
- `_NSGetExecutablePath()` on macOS
- Windows in principal by `GetModuleFileName()`
- ***Reliably returns abs path, but may not resolve softlinks on all platforms!***

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

int main(int argc, char* argv[]) {
    // Initialize binreloc system and error check
    BrInitError err;
    int brOK = br_init(&err);
    if (brOK != 1) {
        return 1;
    }

    // Find ourself...
    char* myPath = br_find_exe("");
    std::clog << "br_find_exe = " << myPath << std::endl;
    free(myPath);

    // What the system called us as...
    std::clog << "argv[0] = " << argv[0] << std::endl;

    // Find the directory we are in
    char* myDir = br_find_exe_dir("");
    std::clog << "br_find_exe_dir = " << myDir << std::endl;
    free(myDir);

    return 0;
}
```

# Binreloc for Libraries

---

- Binreloc also works for libraries, unlike `argv[0]` in the general case.
- Uses `/proc/self/maps` or `DL_info` to locate file that supplied a known symbol.
- ***However, only works for dynamic libraries!***

```
#include <boost/filesystem.hpp>
namespace fs = boost::filesystem;

namespace {
    void initBinReloc() {
        BrInitError err;
        int initOK = br_init_lib(&err);
        if(initOK != 1) {
            throw std::runtime_error("binreloc init failed");
        }
    }
}

namespace rsf {
    std::string getLibraryDir() {
        initBinReloc();
        char* libDir = br_find_exe_dir("");
        fs::path rawLibDir{libDir};
        free(libDir);
        auto canonicalLibDir = fs::canonical(rawLibDir);
        return canonicalLibDir.string();
    }

    std::string getResourceRootDir() {
        fs::path basePath{getLibraryDir()};
        basePath /= "../share/rsf";
        auto absPath = fs::canonical(basePath);
        return absPath.string();
    }
}
```

# Aside on Frameworks/Other Languages...

---

- ... which make this task a lot easier!
- Qt5: `QCoreApplication::applicationDirPath()`
- Poco: `Poco::Util::Application()`
- Python has `__file__`, `setuptools`
- Rust has `std::env::current_exe`
- Go has `os.Executable`

# Summary: Self-Location in C/C++ is Tricky...

---

- No perfect solution, and dependent on whether you are writing a program or a library
- Simple processing and checks of `argv[0]` may be sufficient for simple programs
- Binreloc potentially more reliable, though softlinks may/may not be resolved on all platforms.
- Binreloc only *known* solution for *dynamic* libraries

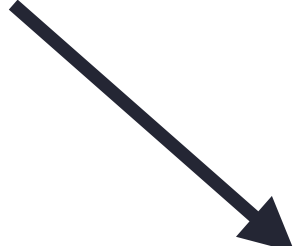
# Relocating CMake/PkgConfig Files

---

- The files installed to help clients of a package link to it from their build scripts
  - `find_package(Foo ...)`
  - `pkg-config --cflags --libs Foo`
- These can have hard coded paths as well, e.g.
  - `set(Foo_INCLUDE_DIR "/abs/path/to/Foo/include")`
  - `prefix=/abs/path/to/Foo`




# Directory of this file when pkg-config runs



```
prefix=${pcfiledir}/../..  
libdir=${prefix}/lib  
includedir=${prefix}/include
```

*Relative paths calculated  
by build system*



```
Name: @PROJECT_NAME@  
Version: @PROJECT_VERSION@  
Description: "Description of @PROJECT_NAME@"
```

```
Requires: boost-filesystem == @Boost_VERSION_STRING@
```

```
Libs: -L${libdir} -lResourceful  
Cflags: -I${includedir}
```

*Use pkg-config to locate  
paths to dependencies*



## Relocatable PkgConfig files

Handing off dependency  
location to pkg-config implies  
a correct PKG\_CONFIG\_PATH

# Relocating CMake -Config Files

---

- Couple of important rules here:
  - *If you're the package author, write `FooConfig.cmake`, NOT `FindFoo.cmake`! `FindFoo.cmake` only for third-party packages that don't/won't supply `FooConfig.cmake`*
  - *Even if you don't build with CMake you can still create a `FooConfig.cmake` for your package (and likewise `pkg-config` in all cases)*
  - *Don't use `find_{path, library}` in `FooConfig.cmake`. These are only for use in `FindFoo.cmake`, a `FooConfig.cmake` already knows where everything is*

```
add_library(rsf SHARED rsf.h rsf.cc)

target_include_directories(rsf PUBLIC
  $<BUILD_INTERFACE:${PROJECT_SOURCE_DIR}>
  $<INSTALL_INTERFACE:${CMAKE_INSTALL_INCLUDEDIR}>)

target_link_libraries(rsf PUBLIC Boost::filesystem)

add_executable(rsf.exe rsf.exe.cc)
target_link_libraries(rsf.exe PRIVATE rsf)

install(TARGETS rsf rsf.exe
  EXPORT rsfTargets
  RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
  LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR})
```

## Modern CMake for Relocatability

Attach build/use requirements to Targets, using **relative** paths for use/install locations

```

include(CMakePackageConfigHelpers)

# Version compatibility file
write_basic_package_version_file(rsfConfigVersion.cmake
    COMPATIBILITY SameMajorVersion)

# Generate the -Config file
configure_package_config_file(rsfConfig.cmake.in rsfConfig.cmake
    INSTALL_DESTINATION "${CMAKE_INSTALL_LIBDIR}/cmake/rsf")

# Install it
install(FILES
    ${CMAKE_CURRENT_BINARY_DIR}/rsfConfigVersion.cmake
    ${CMAKE_CURRENT_BINARY_DIR}/rsfConfig.cmake
    DESTINATION "${CMAKE_INSTALL_LIBDIR}/cmake/rsf")

# An exports file that will define imported targets for use by clients
install(EXPORT rsfTargets
    DESTINATION "${CMAKE_INSTALL_LIBDIR}/cmake/rsf"
    # Namespace is useful because it leads to strong guarantees on
    # existence and location of what we end up linking to
    NAMESPACE rsf::)

```

## Using Helper Module for Package Config file

By exporting the targets, CMake generates **imported** targets for use by clients

```
# CMake will expand this to with macros to assist in
# self-locating paths relative to this file
@PACKAGE_INIT@

# Refind needed boost dependency (the Boost::filesystem target)
find_dependency(Boost @Boost_MAJOR_VERSION@.@Boost_MINOR_VERSION@)

# Include the exported targets file
# CMAKE_CURRENT_LIST_DIR is our self-location variable
include("${CMAKE_CURRENT_LIST_DIR}/rsfTargets.cmake")
```

## The Template rsfConfig.cmake

Not a lot! See the CMake Documentation for more details

```

# Compute the installation prefix relative to this file.
get_filename_component(_IMPORT_PREFIX "${CMAKE_CURRENT_LIST_FILE}" PATH)
get_filename_component(_IMPORT_PREFIX "${_IMPORT_PREFIX}" PATH)
get_filename_component(_IMPORT_PREFIX "${_IMPORT_PREFIX}" PATH)
get_filename_component(_IMPORT_PREFIX "${_IMPORT_PREFIX}" PATH)
if(_IMPORT_PREFIX STREQUAL "/")
    set(_IMPORT_PREFIX "")
endif()

# Create imported target rsf::rsf
add_library(rsf::rsf SHARED IMPORTED)

set_target_properties(rsf::rsf PROPERTIES
    INTERFACE_INCLUDE_DIRECTORIES "${_IMPORT_PREFIX}/include"
    INTERFACE_LINK_LIBRARIES "Boost::filesystem"
)

# Create imported target rsf::rsf.exe
add_executable(rsf::rsf.exe IMPORTED)

# Load information for each installed configuration.
get_filename_component(_DIR "${CMAKE_CURRENT_LIST_FILE}" PATH)
file(GLOB CONFIG_FILES "${_DIR}/rsfTargets-*.cmake")
foreach(f ${CONFIG_FILES})
    include(${f})
endforeach()

```

# Contents of rsfTargets.cmake

Recreates targets, using all relative paths. Dependencies via targets...

```

# Import target "rsf::rsf" for configuration ""
set_property(TARGET rsf::rsf APPEND PROPERTY IMPORTED_CONFIGURATIONS NOCONFIG)
set_target_properties(rsf::rsf PROPERTIES
    IMPORTED_LOCATION_NOCONFIG "${_IMPORT_PREFIX}/lib/librsf.dylib"
    IMPORTED_SONAME_NOCONFIG "@rpath/librsf.dylib"
)

list(APPEND _IMPORT_CHECK_TARGETS rsf::rsf )
list(APPEND _IMPORT_CHECK_FILES_FOR_rsfc::rsf "${_IMPORT_PREFIX}/lib/librsf.dylib" )

# Import target "rsf::rsf.exe" for configuration ""
set_property(TARGET rsf::rsf.exe APPEND PROPERTY IMPORTED_CONFIGURATIONS NOCONFIG)
set_target_properties(rsf::rsf.exe PROPERTIES
    IMPORTED_LOCATION_NOCONFIG "${_IMPORT_PREFIX}/bin/rsf.exe"
)

list(APPEND _IMPORT_CHECK_TARGETS rsf::rsf.exe )
list(APPEND _IMPORT_CHECK_FILES_FOR_rsfc::rsf.exe "${_IMPORT_PREFIX}/bin/rsf.exe" )

```

## Locating the binaries

One locations file per build mode. Sets target location and checks files exist.

# Modern CMake and Relocatability

---

- Use modern target-based approach to define build/use requirements for your programs and libraries
- Always use relative (to `CMAKE_INSTALL_PREFIX`) install paths to enable automatic generation of use time paths based on location of the `-Config` file.
- Export use-time targets rather than set `_LIBRARIES` like variables
- Only link to external dependencies via Imported Targets, refind them at use time in your `-Config` file.



# Configuration/Environment Management

---

- There's an important caveat with the techniques presented for CMake/PkgConfig
- They both hand off responsibility for locating external dependencies to the tools, and hence rely on a correctly set `CMAKE_PREFIX_PATH` and `PKG_CONFIG_PATH`
- Most package managers and development environments seem to know this, but it's something to keep in mind.

# Summary

---

- Creating fully relocatable packages requires both application coding and build scripting
- **Core requirement is scripts and binaries being able to locate themselves on the filesystem at runtime**
- In C/C++, `argv[0]` and/or `Binreloc` can enable this, with some traps and pitfalls
- For CMake/PkgConfig, self-location is straightforward, external dependencies handled through indirection

## **Discussion**

*Example Code: <https://github.com/drbenmorgan/>*  
*Resourceful*