

+ 22

Cross-Building Strategies in the Age of C++ Package Managers

LUIS CARO CAMPOS



20
22



Cross-Building Strategies in the Age of C++ Package Managers

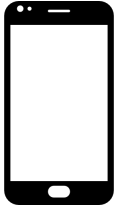
Luis Caro Campos



CONAN
C/C++ Package Manager

ARM ubiquity

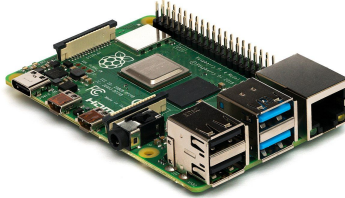
- Porting apps and libraries to run on ARM-based systems is very popular these days



Smartphones



NVIDIA Jetson, Raspberry Pi



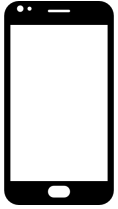
Apple Silicon



Windows/ARM64

ARM ubiquity

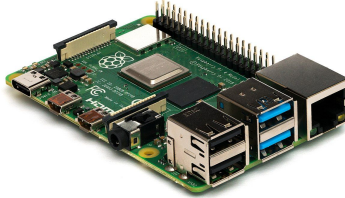
- Porting apps and libraries to run on ARM-based systems is very popular these days



Smartphones



NVIDIA Jetson, Raspberry Pi



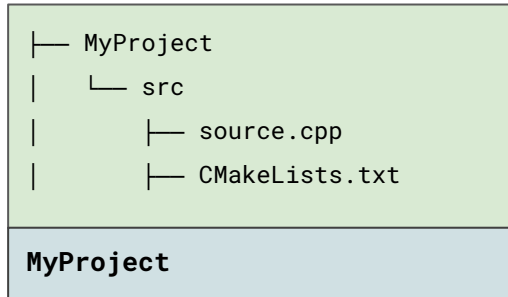
Apple Silicon



Windows/ARM64

- But most development still happens on x86_64 machines

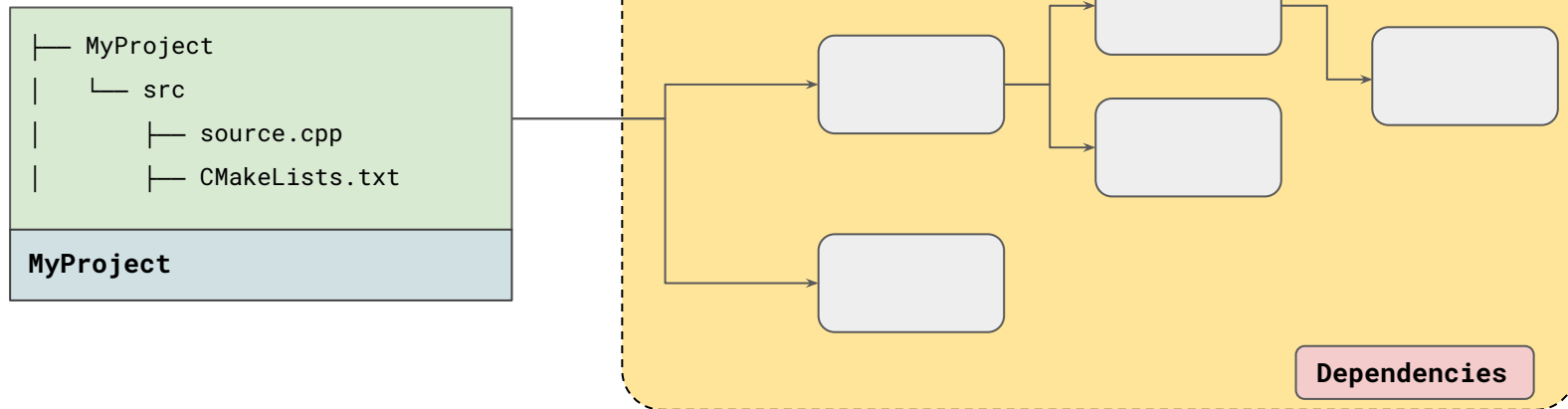
Starting point



Project already builds for:

- Linux x86_64
- Windows x86_64
- macOS x86_64

Starting point



Project already builds for:

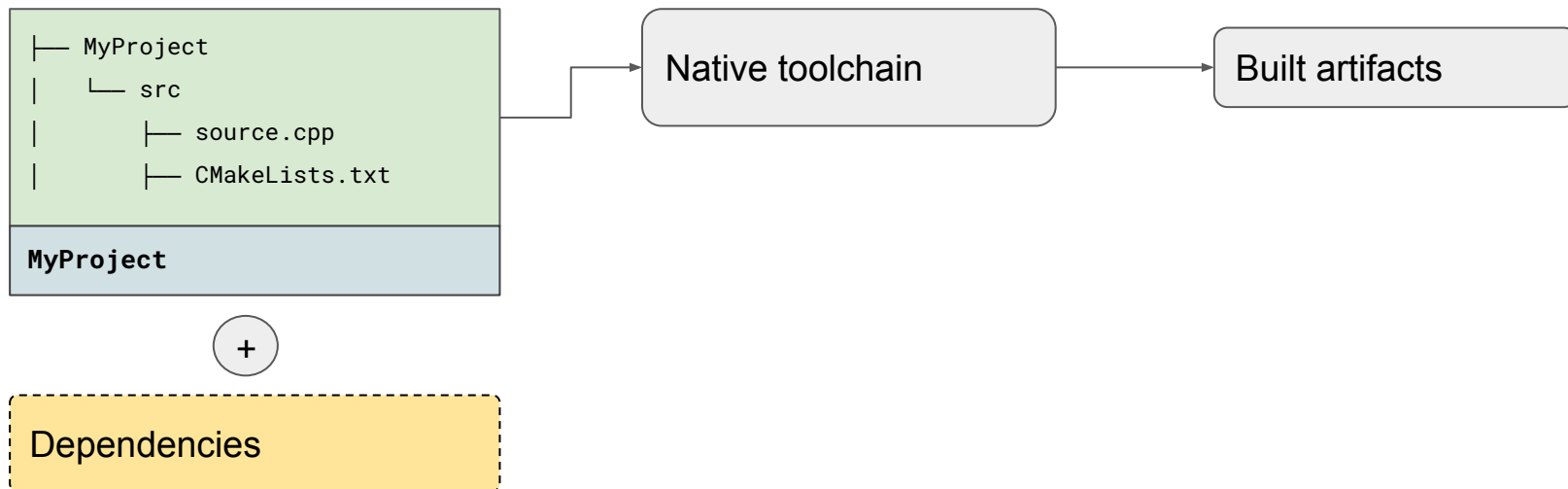
- Linux x86_64
- Windows x86_64
- macOS x86_64

Want to target:

- Linux 64-bit ARMv8 (AArch64)

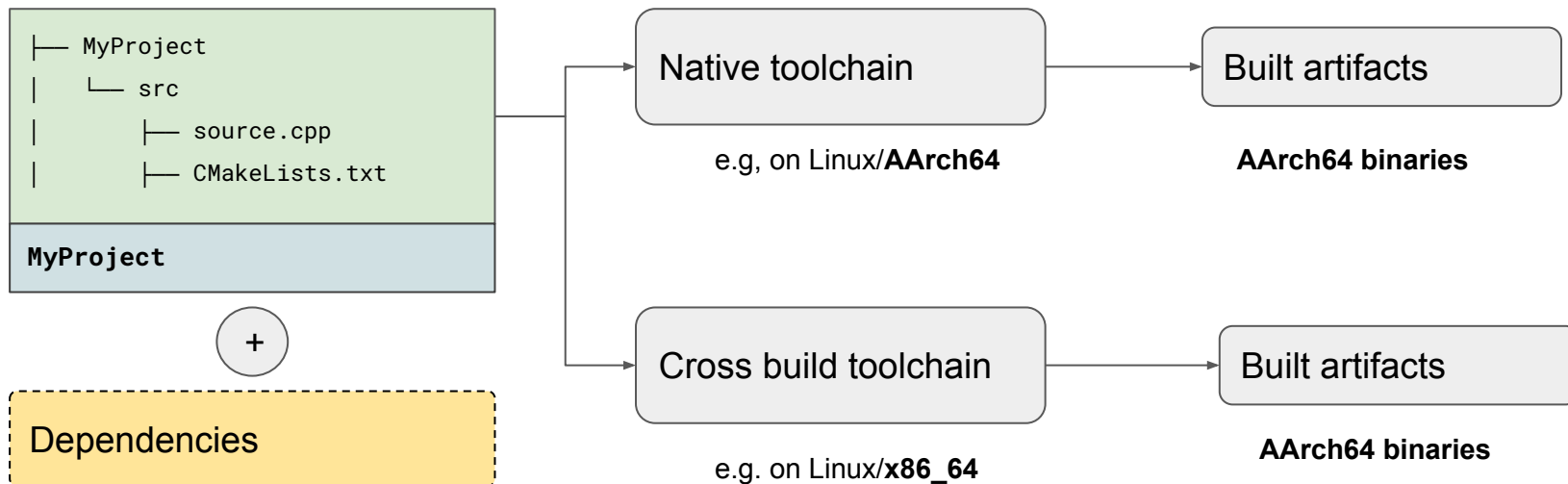
How do we approach this?

- If we have the option, we can build **natively** on the target device
 - E.g. **Raspberry Pi 4 + Ubuntu 22.04**



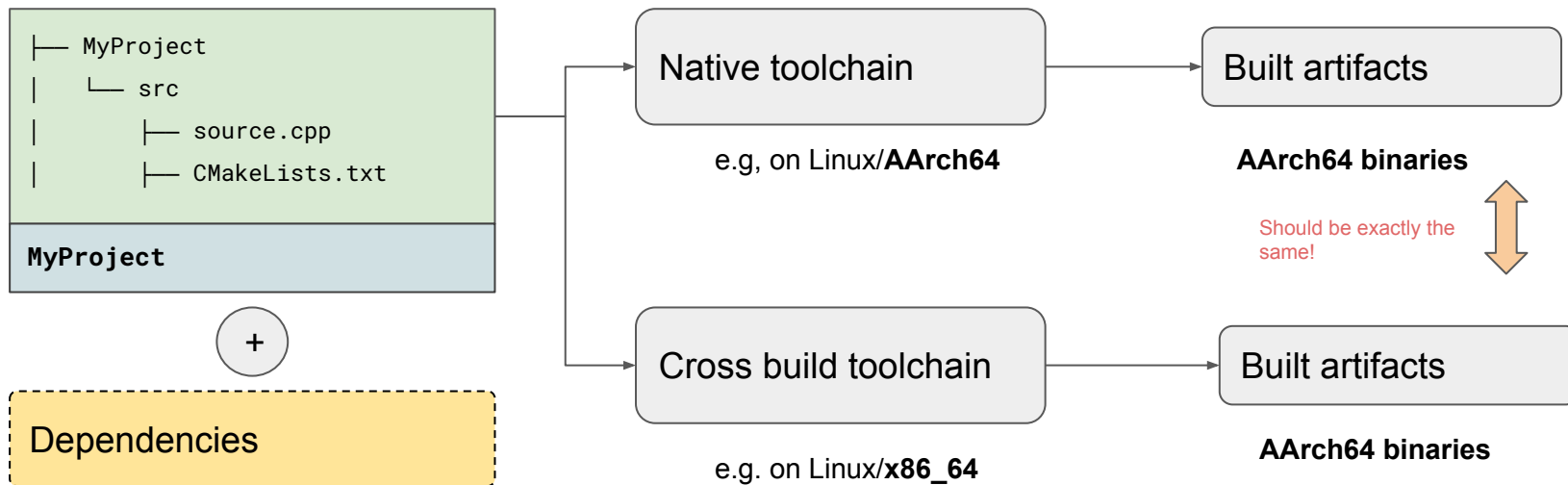
How do we approach this? (cont'd)

- Another possibility is cross building

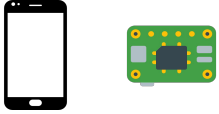


How do we approach this? (cont'd)

- Another possibility is cross building



Why cross build?



No native toolchain

- On some devices, a native toolchain is **not provided**, e.g.: mobile devices, or some embedded devices without a compiler or terminal access



Compile times

- On low-power devices compile times can be significantly slower than on a typical desktop-grade machine



IDE

- Some vendors provide specialised software to develop on, which may require us to port our existing project's build system to a new tool. If we can invoke the tools in the toolchain directly, we may choose to retain our existing build system.

Why cross build?



Hardware availability

- We may want to get a head-start on porting our application to a new platform
- Supply-chain constraints: we may have to wait for months for a device



Continuous integration

- We may already have a project fully ported - and would like to test that building works on CI and generate binaries for the target platform
- Configuring a target device as a build agent could be challenging (or too slow)

Cross-building challenges

- For the simplest cases, most build systems will accept a different compiler and “just work” - being completely agnostic to the platform the compiler is targetting

Cross-building challenges

- For the simplest cases, most build systems will accept a different compiler and “just work” - being completely agnostic to the platform the compiler is targetting
- In other cases...

Current Cross Compile Tutorial for RPI4???

Tue Apr 13, 2021 6:55 pm

I'm about to give up after 3 days of compiler hell..



Start with two rpi to get distcc working. Leave it in "basic" mode with zeroconf. Next, replace "gcc" with a dummy compiler stub (eg: gcc920,g++920) then get it working with that. That done, build your arm-cross within your VM and hack on those stubs within it and fiddle with the VM distcc because by default it may not find it (my gcc920 etc is not in distcc "PATH"). Correctly configured the rpi will automatically find the VM if it's running. Multiple VM's can be implemented simply by cloning the VM and changing its hostname.

Cross-building challenges (cont'd)

Cross compiling in jetson nano

Home >  Autonomous Machines  Jetson & Embedded Systems  Jetson TX2  tensorrt

2061561301



I want to build the cross compiling environment in the ubuntu linux system. I build the tensorRT cross compiling environment. When i watch the NVIDIA TE say(web: [Sample Support Guide :: NVIDIA Deep Learning TensorRT Document](#))

Cross-building challenges (cont'd)

🔒 Cross compiling in jetson nano

Home > ■ Autonomous Machines ■ Jetson & Embedded Systems ■ Jetson TX2 tensorrt

2061561301



I want to build the cross compiling environment in the ubuntu linux system. I build the tensorRT cross compiling environment. When i watch the NVIDIA TE say(web: [Sample Support Guide :: NVIDIA Deep Learning TensorRT Document](#))



AastaLLL  Moderator

Hi,


Since TensorRT depends on much more packages than a CUDA app. We usually recommend users compile it on the Jetson directly.



Dependencies

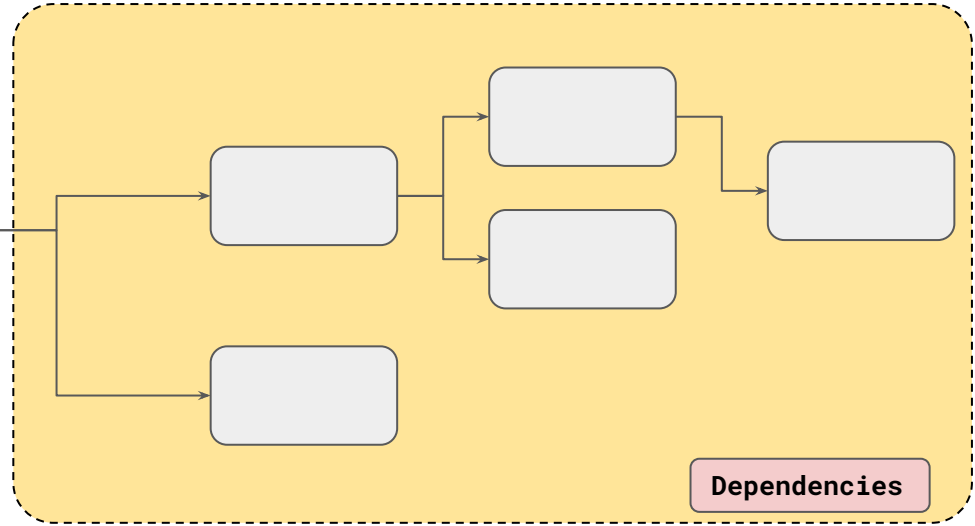
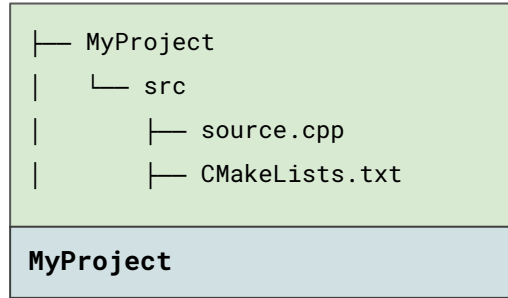
ISO C++ Developer Survey 2022 - “Developer frustrations”

2022 Annual C++ Developer Survey "Lite"

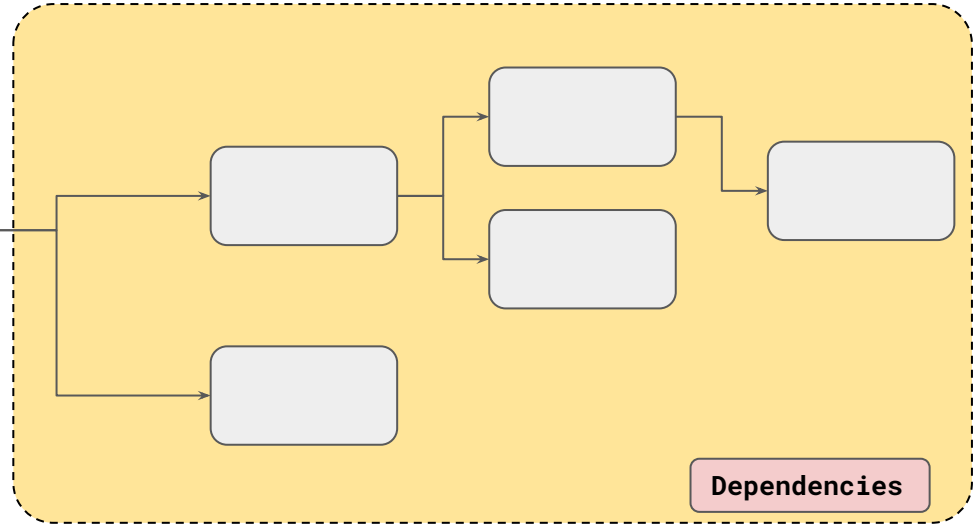
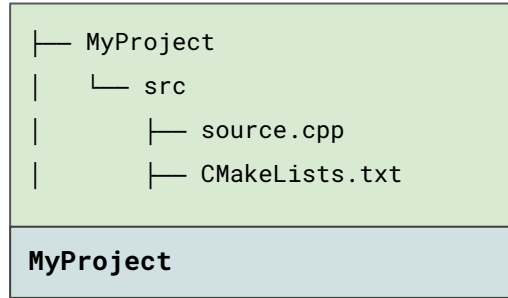


	MAJOR PAIN POINT	MINOR PAIN POINT	NOT A SIGNIFICANT ISSUE FOR ME	TOTAL	WEIGHTED AVERAGE
Managing libraries my application depends on	47.63% 563	34.77% 411	17.60% 208	1,182	2.30
Build times	43.94% 515	38.65% 453	17.41% 204	1,172	2.27
Setting up a continuous integration pipeline from scratch (automated builds, tests, ...)	33.73% 394	40.75% 476	25.51% 298	1,168	2.08
Setting up a development environment from scratch (compiler, build system, IDE, ...)	27.83% 329	42.98% 508	29.19% 345	1,182	1.99
Concurrency safety: Races, deadlocks, performance bottlenecks	25.04% 293	46.67% 546	28.29% 331	1,170	1.97
Managing CMake projects	29.34% 343	38.15% 446	32.51% 380	1,169	1.97
Debugging issues in my code	17.85% 209	54.57% 639	27.58% 323	1,171	1.90

Getting started



Getting started



Cross-build toolchain

- Compiler
- Linker, assembler, ...
- C Library
- ...

Cross-building terminology

GNU convention:

- Build - The machine we are building *on*
- Host - The machine we are building *for*
- Target - In the context of building a *compiler*, the machine that compiler produces code *for* - (not the topic of this talk)

Cross-building terminology

GNU convention:

- Build - The machine we are building *on*
- Host - The machine we are building *for*
- Target - In the context of building a *compiler*, the machine that compiler produces code *for* - (not the topic of this talk)

Different tools use different conventions:

- **Conan:** same as GNU convention (build and host)
- **vcpkg:**
 - “Host” - the machine we are building on
 - “Target” - the machine we are building for
- **CMake:**
 - ``CMAKE_HOST_XXX``: variables relevant to the machine CMake is running **on**.

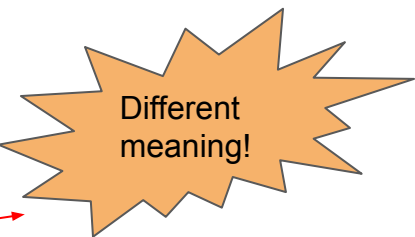
Cross-building terminology

GNU convention:

- Build - The machine we are building *on*
- Host - The machine we are building *for*
- Target - In the context of building a *compiler*, the machine that compiler produces code *for* - (not the topic of this talk)

Different tools use different conventions:

- Conan: same as GNU convention (build and host)
- Vcpkg:
 - "Host" - the machine we are building *on*
 - "Target" - the machine we are building *for*
- CMake:
 - `CMAKE_HOST`_XXX: variables relevant to the machine CMake is running *on*.



Our goal

We may already be using a C++ package manager to handle our dependencies:



vcpkg

```
conan install ..  
cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
```

```
cmake -B build -S .  
-DCMAKE_TOOLCHAIN_FILE=~/.vcpkg/scripts/buildsystems/vcpkg.cmake
```

Our goal

We may already be using a C++ package manager to handle our dependencies:



```
conan install ..  
cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
```

vcpkg

```
cmake -B build -S .  
-DCMAKE_TOOLCHAIN_FILE=~/.vcpkg/scripts/buildsystems/vcpkg.cmake
```



```
conan install .. --profile:host linux-aarch64 --profile:build default  
cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
```

vcpkg

```
cmake -B build -S .  
-DCMAKE_TOOLCHAIN_FILE=~/.vcpkg/scripts/buildsystems/vcpkg.cmake \  
-DVCPKG_TARGET_TRIPLET=arm64-linux
```

..._TRIPLET=arm64-linux

Building dependencies: the compiler

We need to tell the build systems to use the compiler in our toolchain.

- Agnostic build systems (simplest case) - only need to know which compiler to use, e.g.
 - **GCC on Linux**
 - `CC=aarch64-linux-gnu-gcc`
 - `CXX=aarch64-linux-gnu-g++`
 - **Apple's clang** is already a full-fledged cross-compiler:
 - `-arch arm64`
 - `-arch x86_64`
 - **On Windows with MSVC:**
 - `vcvarsall.bat x64_arm64` (Command line)
 - `cmake -G "Visual Studio 16 2019" -A ARM64`

Building dependencies: the compiler (cont'd)

- Very common case - the build system needs to be **aware** that we are in a cross-building scenario:
 - GNU Build system (Autotools) need to be passed the “triplets” during the configure stage
 - `--build: x86_64-pc-linux-gnu`
 - `--host: aarch64-pc-linux-gnu`
 - CMake:
 - `CMAKE_SYSTEM_NAME` is set manually
 - `CMAKE_SYSTEM_PROCESSOR` (optionally set)
- In these cases the build system may perform tasks differently when it is aware that is cross-building

Strategy: Leverage the package manager to specify the compiler

- Both Conan and vcpkg allow passing custom options to underlying build systems (CMake, Makefiles, Meson, ...)
 - Conan: Custom configuration inside **Profiles**
 - vcpkg: special **variables** in custom **triplet** files

```
[settings]
arch=armv8
build_type=Release
compiler=apple-clang
compiler.cppstd=17
compiler.libcxx=libc++
compiler.version=13
os=Macos
```

Conan Build profile:
macos-arm-native

Strategy: Leverage the package manager to specify the compiler

- Both Conan and vcpkg allow passing custom options to underlying build systems (CMake, Makefiles, Meson, ...)
 - Conan: Custom configuration inside **Profile**
 - vcpkg: special **variables** in custom **triples**

```
[settings]
arch=armv8
build_type=Release
compiler=apple-clang
compiler.cppstd=17
compiler.libcxx=libc++
compiler.version=13
os=Macos
```

Conan Build profile:
macos-arm-native

```
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.cppstd=17
compiler.libcxx=libc++
compiler.version=13
os=Macos
```

```
[conf]
tools.cmake.cmaketoolchain:system_name="Darwin"
tools.cmake.cmaketoolchain:system_processor="x86_64"

tools.build:cflags=["-arch x86_64"]
tools.build:cflags=["-arch x86_64"]
tools.build:sharedlinkflags=["-arch x86_64"]
tools.build:exelinkflags=["-arch x86_64"]

tools.apple.sdk_path=/path/to/MacOSX.sdk
```

Conan Host profile: **macos-x86_64-cross**

Building ZLIB, macOS example

```
conan create . --version=1.2.12 --profile:host=macos-arm-native --profile:build=macos-arm-native -o "zlib*:shared=True"
```

 Native build

Building ZLIB, macOS example

```
conan create . --version=1.2.12 --profile:host=macos-arm-native --profile:build=macos-arm-native -o "zlib*:shared=True"
```

```
conan create . --version=1.2.12 --profile:host=macos-x86_64-cross --profile:build=macos-arm-native -o "zlib*:shared=True"
```

 Cross build

Building ZLIB, macOS example

```
conan create . --version=1.2.12 --profile:host=macos-arm-native --profile:build=macos-arm-native -o "zlib*:shared=True"
```

```
conan create . --version=1.2.12 --profile:host=macos-x86_64-cross --profile:build=macos-arm-native -o "zlib*:shared=True"
```

```
(venv-conan2) % conan list packages zlib/1.2.12#latest
Local Cache:
zlib/1.2.12#b76db676bd992afa93dd18a675323942:24612164eb0760405fcd237df0102e554ed1cb2f
  settings:
    arch=x86_64
    build_type=Release
    compiler=apple-clang
    compiler.version=13
    os=Macos
  options:
    shared=True
zlib/1.2.12#b76db676bd992afa93dd18a675323942:a3c9d80d887539fac38b81ff8cd4585fe42027e0
  settings:
    arch=armv8
    build_type=Release
    compiler=apple-clang
    compiler.version=13
    os=Macos
  options:
    shared=True
```

Building ZLIB, macOS example

```
conan create . --version=1.2.12 --profile:host=macos-arm-native --profile:build=macos-arm-native -o "zlib*:shared=True"
```

```
conan create . --version=1.2.12 --profile:host=macos-x86_64-cross --profile:build=macos-arm-native -o "zlib*:shared=True"
```

```
(venv-conan2) % conan list packages zlib/1.2.12#latest
Local Cache:
zlib/1.2.12#b76db676bd992afa93dd18a675323942:24612164eb0760405fcd237df0102e554ed1cb2f
  settings:
    arch=x86_64
    build_type=Release
    compiler=apple-clang
    compiler.version=13
    os=Macos
  options:
    shared=True
zlib/1.2.12#b76db676bd992afa93dd18a675323942:a3c9d80d887539fac38b81ff8cd4585fe42027e0
  settings:
    arch=armv8
    build_type=Release
    compiler=apple-clang
    compiler.version=13
    os=Macos
  options:
    shared=True
```

```
% lipo -detailed_info libz.dylib
input file libz.dylib is not a fat file
Non-fat file: libz.dylib is architecture:
x86_64
```

```
% lipo -detailed_info libz.dylib
input file libz.dylib is not a fat file
Non-fat file: libz.dylib is architecture:
arm64
```

Strategy 2: Package your compiler toolchain up

- There are cases where we are given a pre-existing compiler toolchain.
For example Linux:

```
[settings]
arch=x86_64
build_type=Release
compiler=gcc
compiler.cppstd=gnu17
compiler.libcxx=libstdc++11
compiler.version=11
os=Linux
```

Conan Build profile:
linux-x86_64-native

Strategy 2: Package your compiler toolchain up

- There are cases where we are given a pre-existing compiler toolchain.
For example Linux:

```
[settings]
arch=x86_64
build_type=Release
compiler=gcc
compiler.cppstd=gnu17
compiler.libcxx=libstdc++11
compiler.version=11
os=Linux
```

Conan Build profile:
linux-x86_64-native

```
[settings]
arch=armv8
build_type=Release
compiler=gcc
compiler.cppstd=gnu17
compiler.libcxx=libstdc++11
compiler.version=10
os=Linux
```

```
[tool_requires]
gcc-rpi4-aarch64/10.3
```

Conan Host profile:
linux-aarch64-**cross**

Strategy: Package your compiler toolchain up (cont'd)

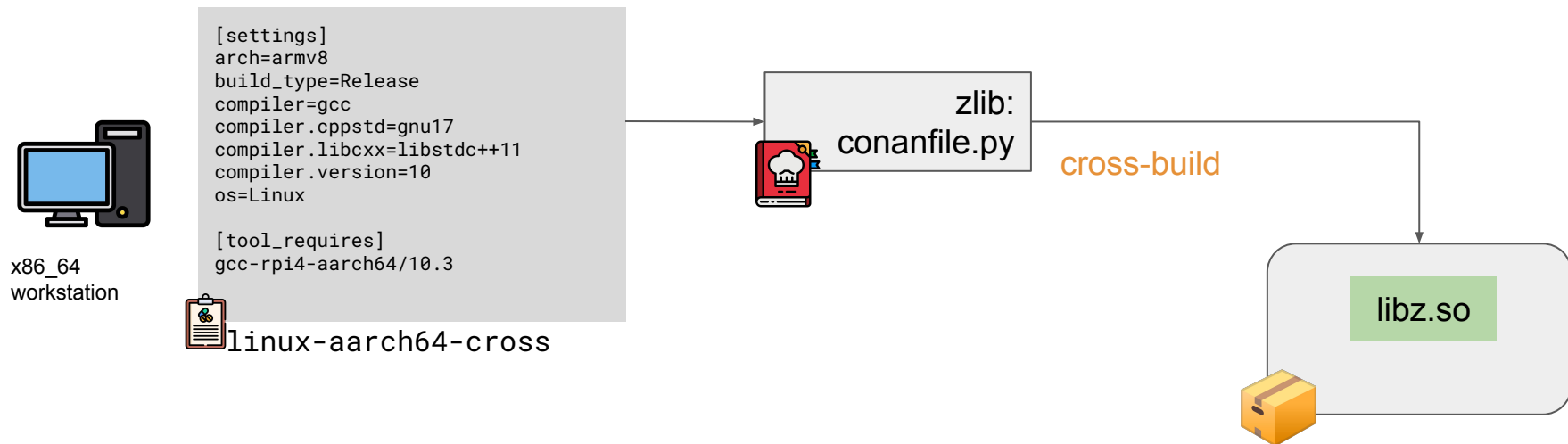
```
def package_info(self):
    self.buildenv_info.append_path("PATH", os.path.join(self.package_folder, "bin"))
    vars = {
        "CC": "aarch64-linux-gnu-gcc",
        "CXX": "aarch64-linux-gnu-g++",
        "LD": "aarch64-linux-gnu-ld",
        "AS": "aarch64-linux-gnu-as",
        "AR": "aarch64-linux-gnu-ar",
        "RANLIB": "aarch64-linux-gnu-ranlib"
    }

    for env_var, value in vars.items():
        app_path = os.path.join(self.package_folder, "bin", f"{value}")
        self.buildenv_info.define(env_var, app_path)

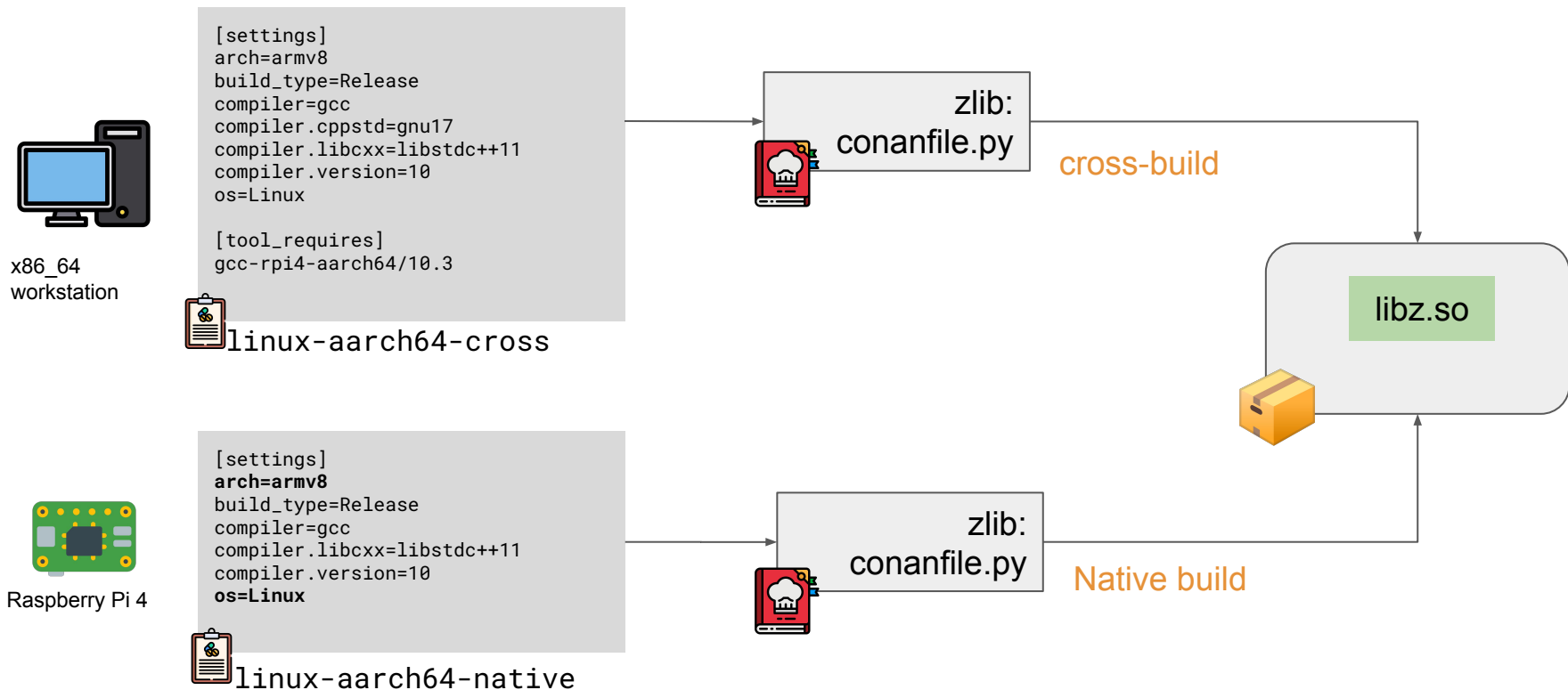
    # Cause the variables needed to be consumed by the CMakeToolchain generator downstream
    self.conf_info.define("tools.cmake.cmaketoolchain:system_name", "Linux")
    self.conf_info.define("tools.cmake.cmaketoolchain:system_processor", "aarch64")
```

The package defines the variables understood by different build systems
It will contain a pre-existing binary package

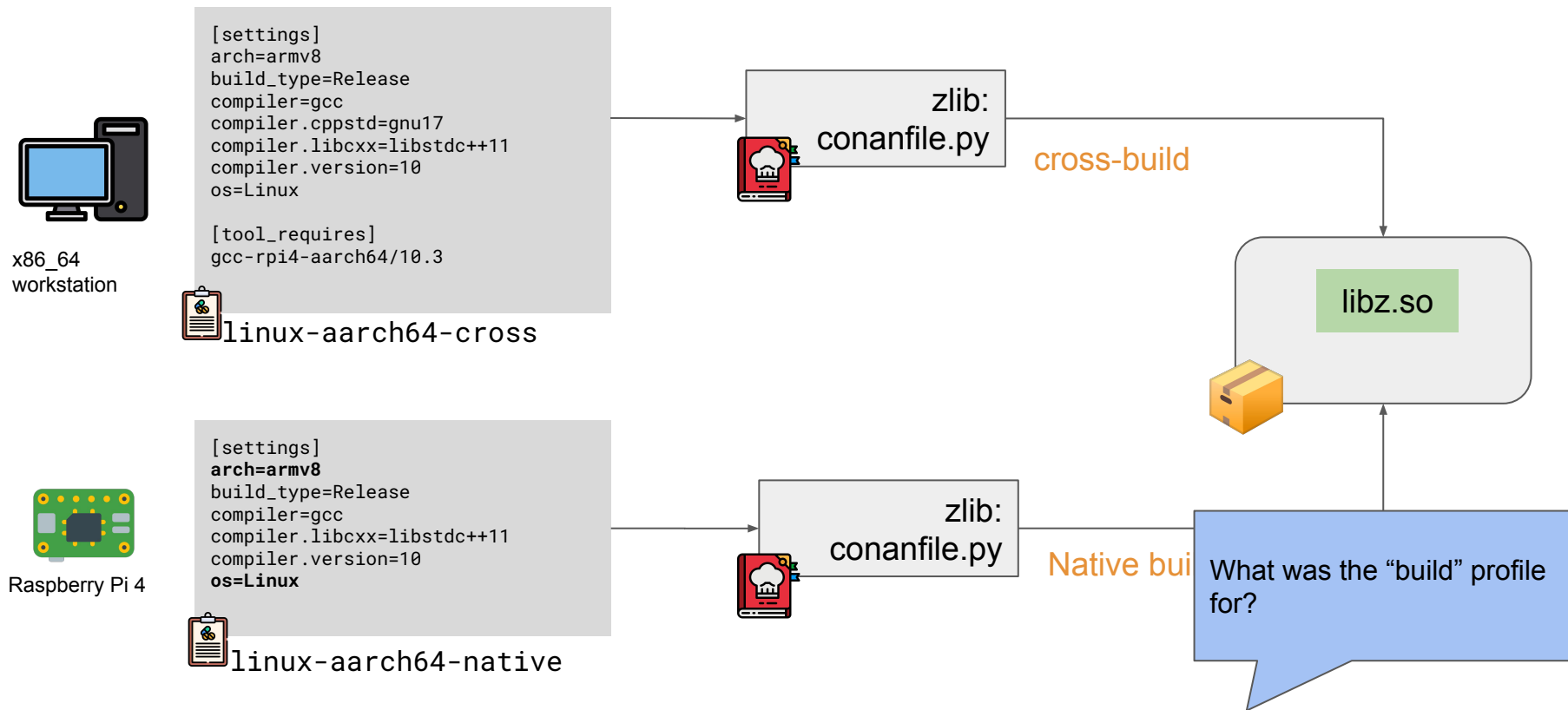
Library-only packages



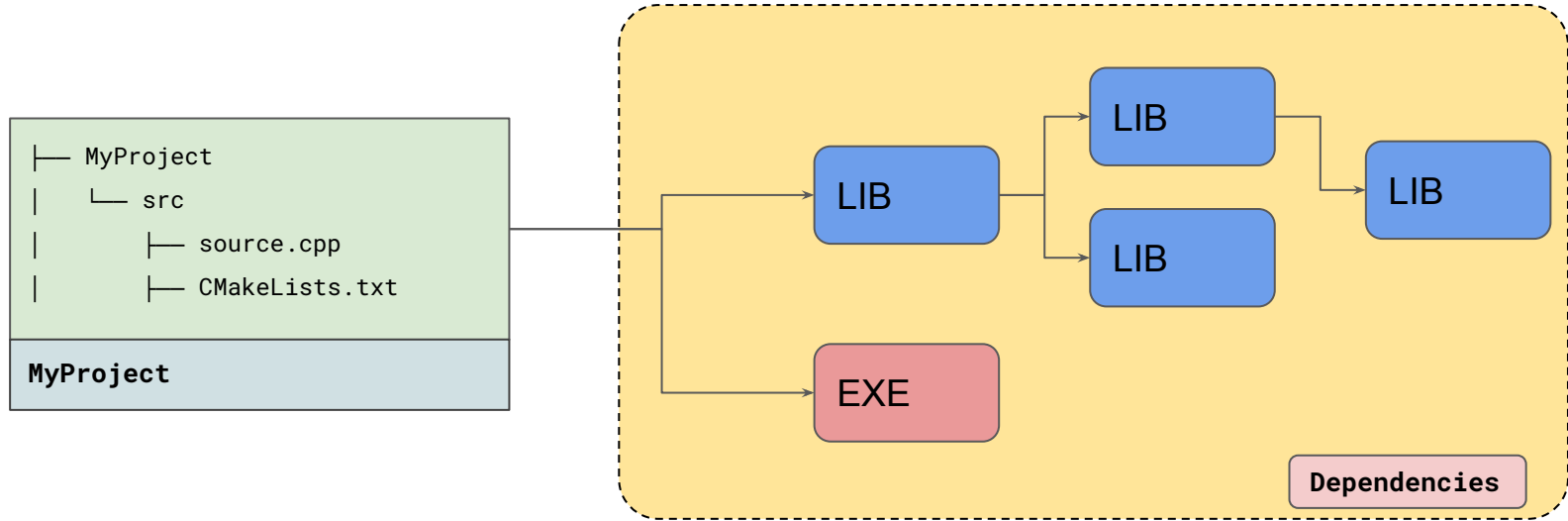
Library-only packages



Library-only packages

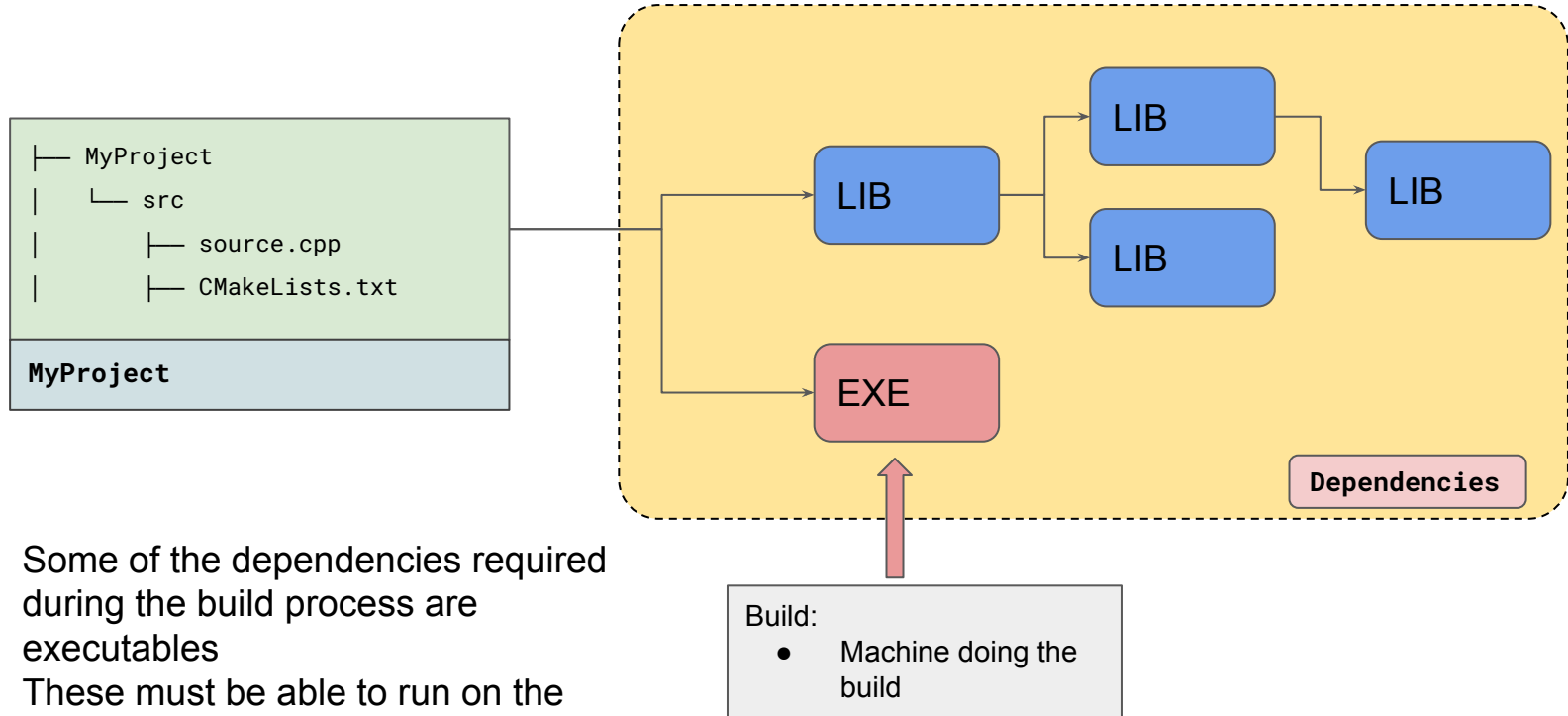


Executable dependencies



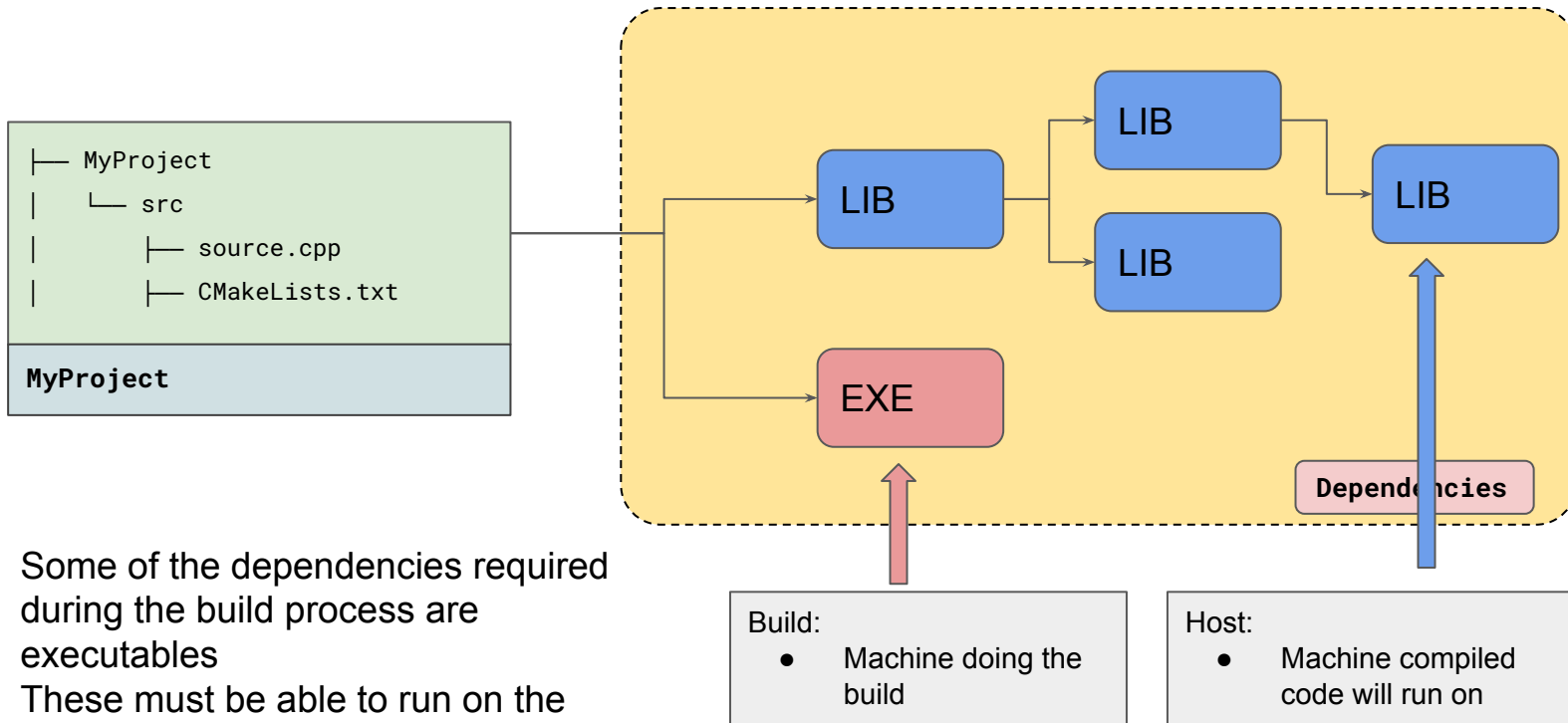
- Some of the dependencies required during the build process are executables
- These must be able to run on the machine performing the build

Executable dependencies



- Some of the dependencies required during the build process are executables
- These must be able to run on the machine performing the build

Executable dependencies



- Some of the dependencies required during the build process are executables
- These must be able to run on the machine performing the build

Strategy: requirements for build-time executables

```
class MyPkg(ConanFile):  
    requires = "opencv/3.4.17", "zlib/1.2.12"  
    tool_requires = "tool_foo/1.9", "tool_bar/8.7"
```



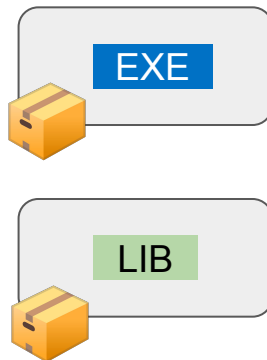
```
conan install . \  
--profile:host=macos-x86_64-cross \  
--profile:build=macos-arm-native
```

```
{  
  "name": "contoso-http-library",  
  "version-string": "1.0.0",  
  "dependencies": [  
    "contoso-core-library",  
    {  
      "name": "contoso-code-generator",  
      "host": true  
    },  
    {  
      "name": "contoso-build-system",  
      "host": true  
    }  
  ]  
}
```

vcpkg

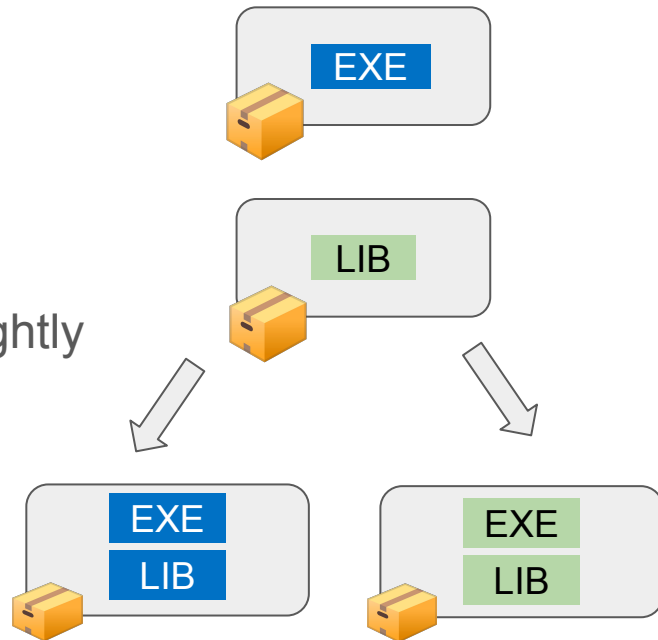
Library and executable combos

- So far we have seen packages with:
 - Only executables (“build context”)
 - Only libraries (“host context”)
- But there’s situations where executables are tightly coupled with libraries, e.g.
 - Protobuf compiler and libprotobf
 - Qt tools (moc, uic, ...) and Qt libraries

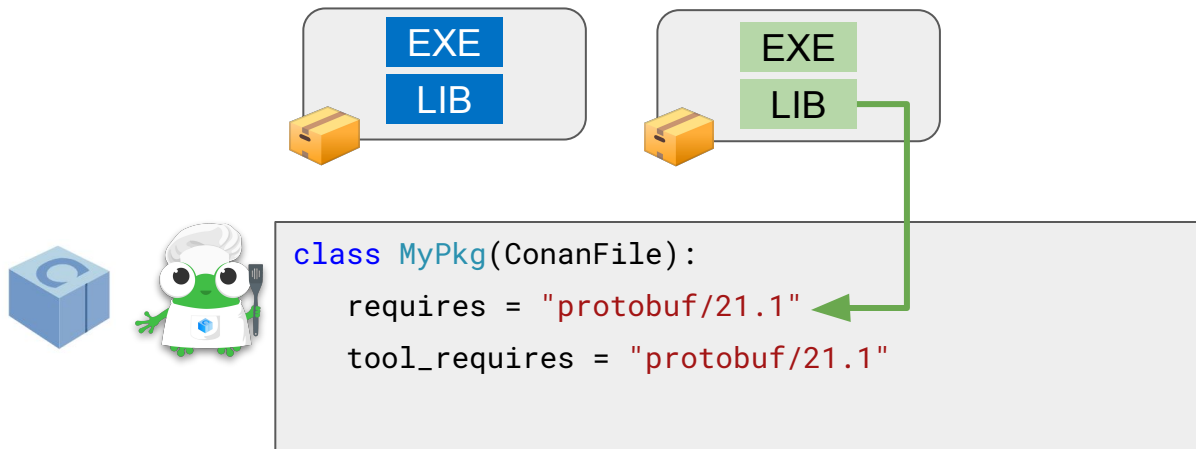


Library and executable combos

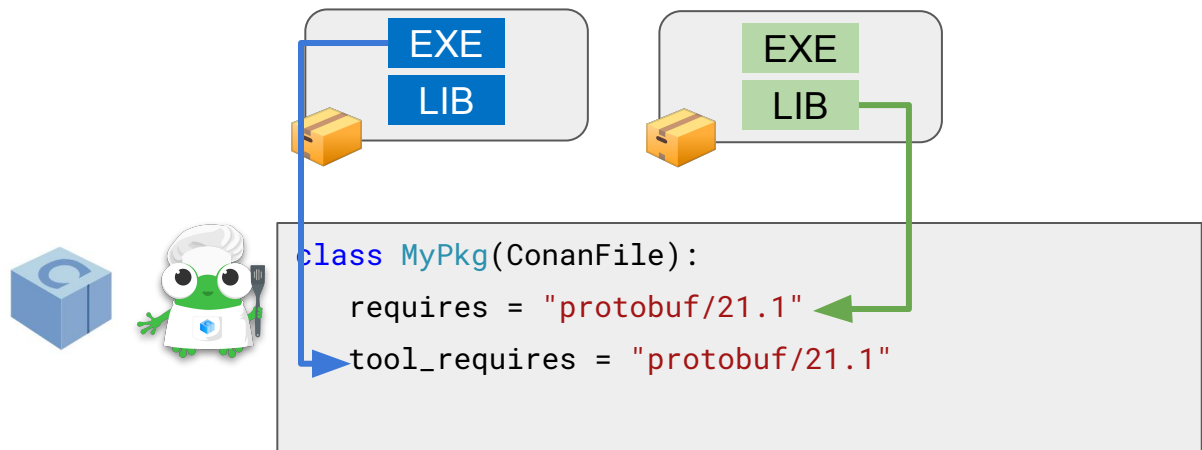
- So far we have seen packages with:
 - Only executables (“build context”)
 - Only libraries (“host context”)
- But there’s situations where executables are tightly coupled with libraries, e.g.
 - Protobuf compiler and libprotobf
 - Qt tools (moc, uic, ...) and Qt libraries



Strategy: same requirement in both contexts



Strategy: same requirement in both contexts



```
conan install . --profile:build=macos-arm-native --profile:host=macos-x86_64-cross
```

```
----- Computing necessary packages -----
```

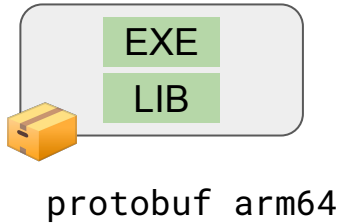
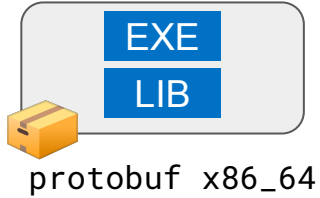
```
Requirements
```

```
    protobuf/21.1:61fd152983603f73eb14ba2d71fee3169a0f586a - Cache
```

```
Build requirements
```

```
    protobuf/21.1:7253508f17f424e6fa69e70030cbeac2c8dec086- Cache
```

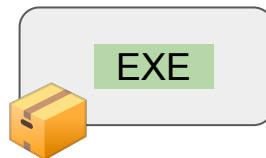
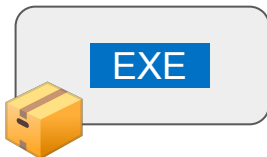
Strategy: split packages for executable and library



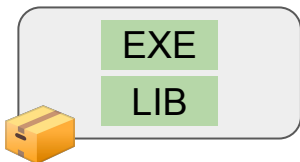
Strategy: split packages for executable and library



protobuf x86_64



protobuf-compiler

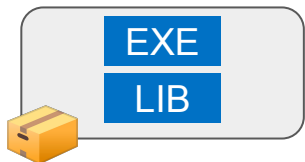


protobuf arm64

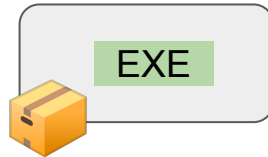
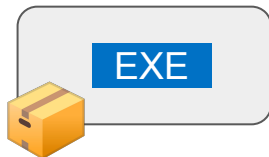
ARM

x86_64

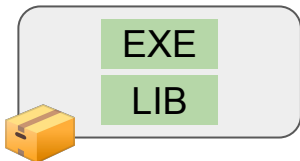
Strategy: split packages for executable and library



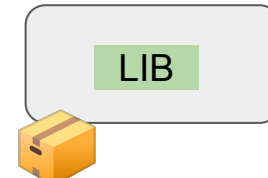
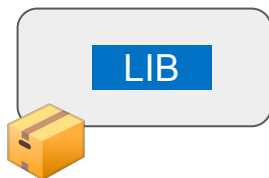
protobuf x86_64



protobuf-compiler



protobuf arm64

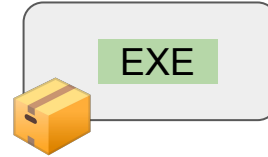


libprotobuf

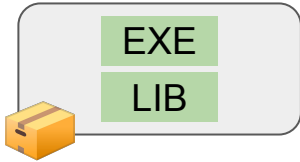
Strategy: split packages for executable and library



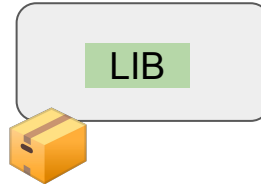
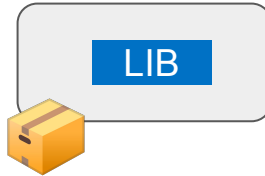
protobuf x86_64



protobuf-compiler



protobuf arm64



libprotobuf

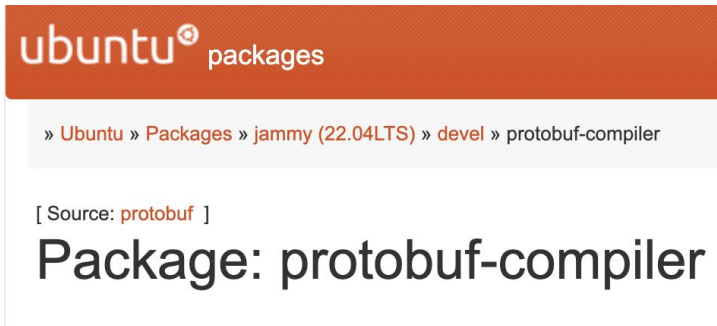


```
class MyPkg(ConanFile):  
    requires = "libprotobuf/21.1"  
    tool_requires = "protobuf-compiler/21.1"
```

ARM

x86_64

Split packages for executables and libraries (cont'd)

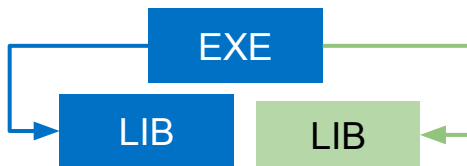


Package: libprotobuf23 (3.12.4-1ubuntu7)

- Debian systems support **multiarch** since the i386 / amd64 days
- Cross building is possible since the build tools (compiler, linker, CMake...) are aware of the multiple per-architecture library directories

Cross-arch tooling

- A special **corner case** of “library and executable combo”
- Where an executable (running on the build machine) may use libraries for the host architecture. Examples:
 - The linker (part of the compiler toolchain) runs on **x86_64**, but it takes compiled **arm64** libraries as input
 - The CUDA compiler (nvcc) - running in the build context, needs the libraries from the host context as inputs
 - Qt tools (moc, uic, ...)

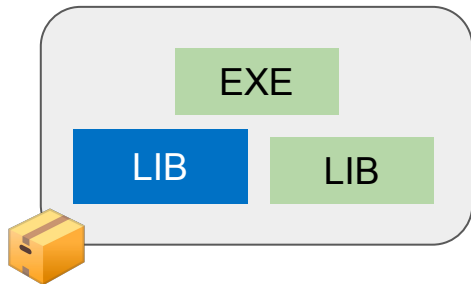
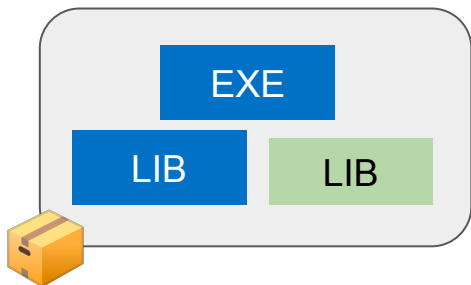


ARM

x86_64

Cross-arch tooling (cont'd)

How do we package this? A potential solution



build_context_activated

When you have a **build-require**, by default, the config files (*xxx-config.cmake*) files are not generated. But you can activate it using the **build_context_activated** attribute:

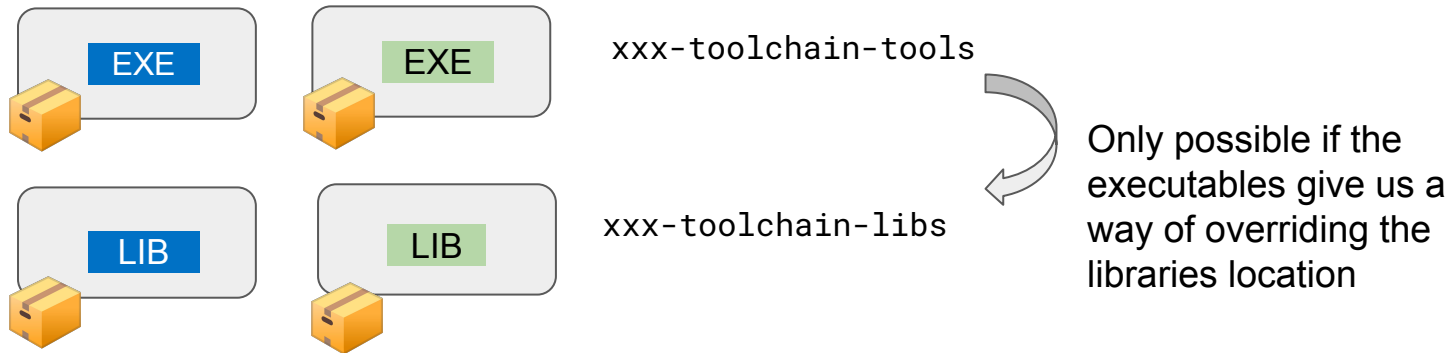
```
tool_requires = ["my_tool/0.0.1"]

def generate(self):
    cmake = CMakeDeps(self)
    # generate the config files for the tool require
    cmake.build_context_activated = ["my_tool"]
    cmake.generate()
```



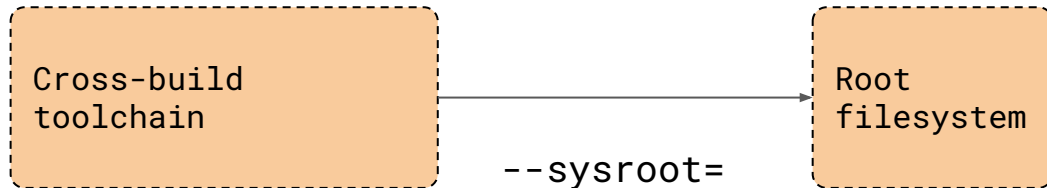
Downside: we need to conditionally set which set of libraries to use depending on the context

Cross-arch tooling (cont'd) - An alternative



```
class MyPkg(ConanFile):  
    requires = "xxx-toolchain-libs/11.4"  
    tool_requires = "xxx-toolchain-tools/11.4"
```

“Sysroot” and system libraries on Linux

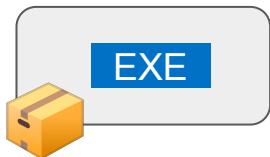


- For some target devices, we may be provided with a reference **root filesystem**.
- The toolchain may contain only C/C++ headers and runtime
- The root filesystem contains all “OS-level” system libraries
- On some systems, all are part of the same toolchain: e.g macOS, QNX

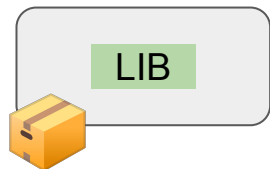
“Sysroot” on embedded Linux - caveats

- On embedded Linux, a number of arbitrary libraries can be contained in the reference root filesystem - these are dependencies too!
 - **Potential for problems** if we independently package libraries that may also have copies inside the root filesystem
- Passing `--sysroot` to the compiler and linker will re-root the search for implicitly linked libraries in the direction of the sysroot
 - We need to make sure that binutils and glibc of the toolchain are compatible with the versions in the reference root filesystem (and the device itself)
- Defining `CMAKE_SYSROOT` also causes CMake to re-root its search logic
 - This may cause issues if the root filesystem has libraries that we also have in the package manager - and requires some care!
 - To avoid issues, if a library is part of the system (and is present in the root filesystem), do not consume it from the C++ package manager.

Strategy: package the root filesystem



Compiler toolchain



Root filesystem

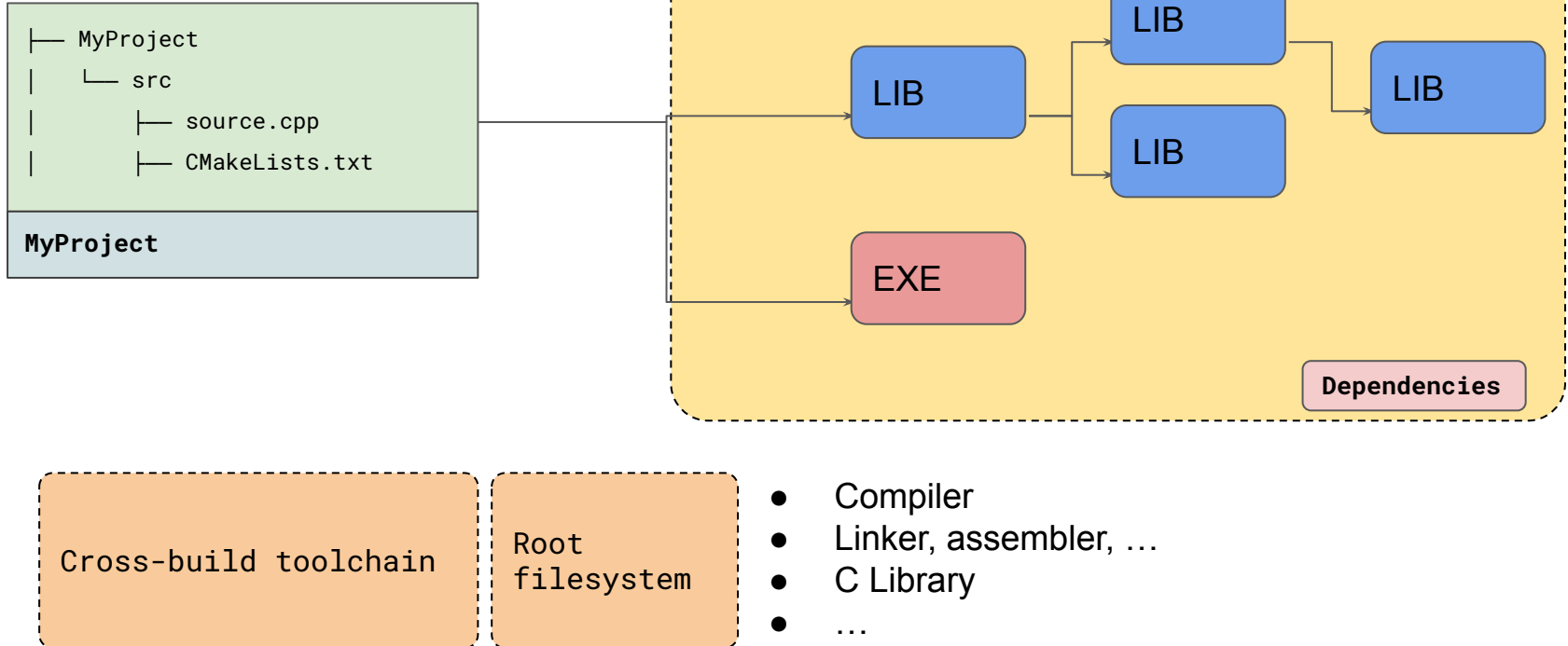
```
[settings]
arch=armv8
build_type=Release
compiler=gcc
compiler.cppstd=gnu17
compiler.libcxx=libstdc++11
compiler.version=10
os=Linux
```

```
[tool_requires]
gcc-rpi4-aarch64/10.3
rpi4-sysroot/22.04
```

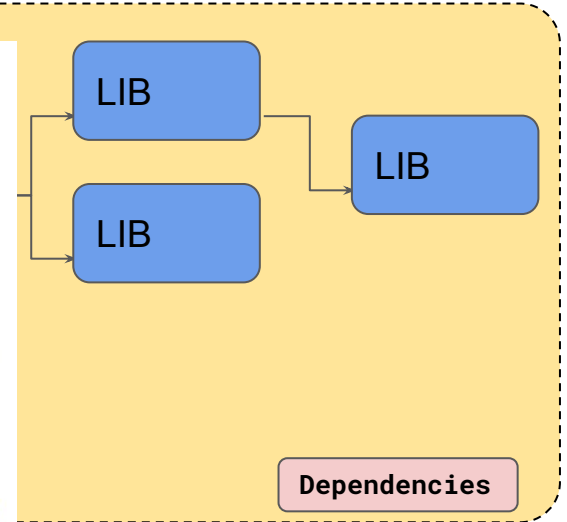
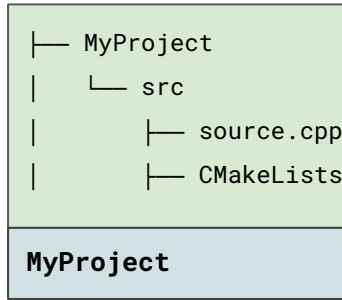
Conan Host profile:
linux-aarch64-cross

```
def package_info(self):
    self.conf_info.define("tools.build:cflags", f"--sysroot={self.package_folder}")
    self.conf_info.define("tools.build:cxxflags", f"--sysroot={self.package_folder}")
    self.conf_info.define("tools.build:sharedlinkflags", f"--sysroot={self.package_folder}")
    self.conf_info.define("tools.build:exelinkflags", f"--sysroot={self.package_folder}")
```

Ready to build our own code!



Ready to build our own code!



Cross-build toolchain

ROOT
filesystem

- Linker, assembler, ...
- C Library
- ...

Closing remarks

- We can leverage the power of modern C++ package managers to handle our project's dependencies when cross-building
 - Retaining our existing workflows
 - If feasible, the same project can be built natively or cross-built from a different machine offloading the heavy lifting to the package manager
- Cross-building is a dependency problem: compiler, linker and build system need to find libraries in different places than the default paths
 - This is already how Conan and vcpkg work
- Dependencies are easy to handle if they are library-only or executable only
- There are more complexities around executables that are tightly coupled with libraries
 - But can be overcome, and this can be a one-time effort

Thank you

Any questions?