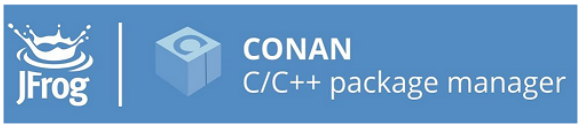


Conan Documentation

Release 2.0.0-beta

The Conan team

Sep 12, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

Conan is a dependency and package manager for C and C++ languages. It is free and open-source, works in all platforms (Windows, Linux, OSX, FreeBSD, Solaris, etc.), and can be used to develop for all targets including embedded, mobile (iOS, Android), and bare metal. It also integrates with all build systems like CMake, Visual Studio (MSBuild), Makefiles, SCons, etc., including proprietary ones.

It is specifically designed and optimized for accelerating the development and Continuous Integration of C and C++ projects. With full binary management, it can create and reuse any number of different binaries (for different configurations like architectures, compiler versions, etc.) for any number of different versions of a package, using exactly the same process in all platforms. As it is decentralized, it is easy to run your own server to host your own packages and binaries privately, without needing to share them. The free JFrog Artifactory Community Edition (CE) is the recommended Conan server to host your own packages privately under your control.

Conan is mature and stable, with a strong commitment to forward compatibility (non-breaking policy), and has a complete team dedicated full time to its improvement and support. It is backed and used by a great community, from open source contributors and package creators in ConanCenter to thousands of teams and companies using it.

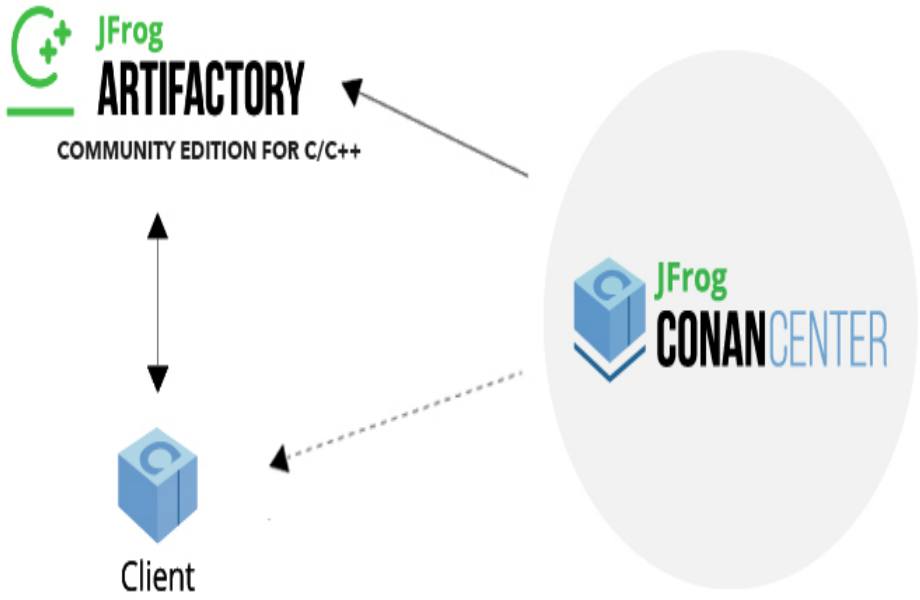
1.1 Open Source

Conan is Free and Open Source, with a permissive MIT license. Check out the source code and issue tracking (for questions and support, reporting bugs and suggesting feature requests and improvements) at https://github.com/conan-io/conan

1.2 Decentralized package manager

Conan is a decentralized package manager with a client-server architecture. This means that clients can fetch packages from, as well as upload packages to, different servers ("remotes"), similar to the "git" push-pull model to/from git remotes.

At a high level, the servers are just storing packages. They do not build nor create the packages. The packages are created by the client, and if binaries are built from sources, that compilation is also done by the client application.



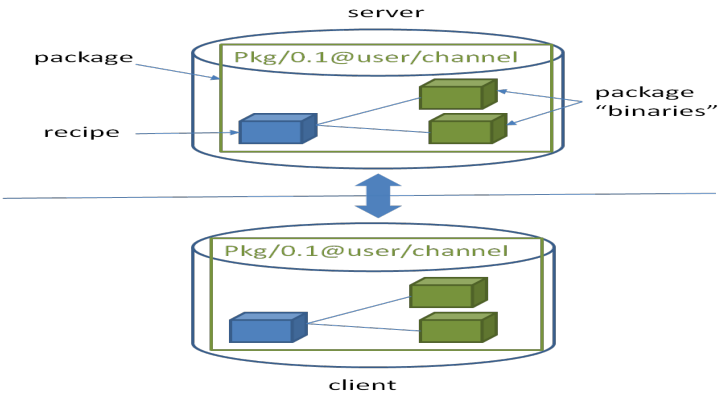
The different applications in the image above are:

- The Conan client: this is a console/terminal command-line application, containing the heavy logic for package creation and consumption. Conan client has a local cache for package storage, and so it allows you to fully create and test packages offline. You can also work offline as long as no new packages are needed from remote servers.
- JFrog Artifactory Community Edition (CE) is the recommended Conan server to host your own packages privately under your control. It is a free community edition of JFrog Artifactory for Conan packages, including a WebUI, multiple auth protocols (LDAP), Virtual and Remote repositories to create advanced topologies, a Rest API, and generic repositories to host any artifact.
- The conan_server is a small server distributed together with the Conan client. It is a simple open-source implementation and provides basic functionality, but no WebUI or other advanced features.
- ConanCenter is a central public repository where the community contributes packages for popular open-source libraries like Boost, Zlib, OpenSSL, Poco, etc.

1.3 Binary management

One of the most powerful features of Conan is that it can create and manage pre-compiled binaries for any possible platform and configuration. By using pre-compiled binaries and avoiding repeated builds from source, it saves significant time for developers and Continuous Integration servers, while also improving the reproducibility and traceability of artifacts.

A package is defined by a "conanfile.py". This is a file that defines the package's dependencies, sources, how to build the binaries from sources, etc. One package "conanfile.py" recipe can generate any arbitrary number of binaries, one for each different platform and configuration: operating system, architecture, compiler, build type, etc. These binaries can be created and uploaded to a server with the same commands in all platforms, having a single source of truth for all packages and not requiring a different solution for every different operating system.



Installation of packages from servers is also very efficient. Only the necessary binaries for the current platform and configuration are downloaded, not all of them. If the compatible binary is not available, the package can be built from sources in the client too.

1.4 All platforms, all build systems and compilers

Conan works on Windows, Linux (Ubuntu, Debian, RedHat, ArchLinux, Raspbian), OSX, FreeBSD, and SunOS, and, as it is portable, it might work in any other platform that can run Python. It can target any existing platform: ranging from bare metal to desktop, mobile, embedded, servers, and cross-building.

Conan works with any build system too. There are built-in integrations to support the most popular ones like CMake, Visual Studio (MSBuild), Autotools and Makefiles, Meson, SCons, etc., but it is not a requirement to use any of them. It is not even necessary that all packages use the same build system: each package can use their own build system, and depend on other packages using different build systems. It is also possible to integrate with any build system, including proprietary ones.

Likewise, Conan can manage any compiler and any version. There are default definitions for the most popular ones: gcc, cl.exe, clang, apple-clang, intel, with different configurations of versions, runtimes, C++ standard library, etc. This model is also extensible to any custom configuration.

1.5 Stable

From Conan 2.0 and onwards, there is a commitment to stability, with the goal of not breaking user space while evolving the tool and the platform. This means:

- Moving forward to following minor versions 2.1, 2.2, ..., 2.X should never break existing recipes, packages or command line flows
- If something is breaking, it will be considered a regression and reverted
- Bug fixes will not be considered breaking, recipes and packages relying on the incorrect behavior of such bugs will be considered already broken.
- Only documented features are considered part of the public interface of Conan. Private implementation details, and everything not included in the documentation is subject to change.
- The compatibility is always considered forward. New APIs, tools, methods, helpers can be added in following 2.X versions. Recipes and packages created with these features will be backwards incompatible with earlier Conan versions.
- Only the latest released patch (major.minor.patch) of every minor version is supported and stable.

There are some things that are not included in this commitment:

- Public repositories, like **ConanCenter**, assume the use of the latest version of the Conan client, and using an older version may result in failure of packages and recipes created with a newer version of the client. It is recommended to use your own private repository to store your own copy of the packages for production, or as a secondary alternative, to use some locking mechanism to avoid possible disruption from packages in ConanCenter that are updated and require latest Conan version.
- Configuration and automatic tools detection, like the detection of the default profile (`conan profile detect`) can and will change at any time. Users are encouraged to define their configurations in their own profiles files for repeatability. New versions of Conan might detect different default profiles.
- Builtin default implementation of extension points as plugins or hooks can also change with every release. Users can provide their own ones for stability.
- Output of packages templates with `conan new` can update at any time to use latest features.
- The output streams stdout, stderr, i.e. the terminal output can change at any time. Do not parse the terminal output for automation.
- Anything that is explicitily labeled as `experimental`, `alpha`, `beta` in the documentation, or in the Conan cli output.
- Other tools and repositories outside of the Conan client

Conan needs Python>=3.6 to run. Conan will deprecate support for Python versions 1 year after those versions have been declared End Of Life (EOL).

If you have any question regarding Conan updates, stability, or any clarification about this definition of stability, please report in the documentation issue tracker: https://github.com/conan-io/docs.

1.6 Community

Conan is being used in production by thousands of companies like TomTom, Audi, RTI, Continental, Plex, Electrolux and Mercedes-Benz and many thousands of developers around the world.

But an essential part of Conan is that many of those users will contribute back, creating an amazing and helpful community:

- The https://github.com/conan-io/conan project has around 6K stars in Github and counts with contributions of almost 300 different users (this is just the client tool).
- Many other users contribute recipes for ConanCenter via the https://github.com/conan-io/conan-center-index repo, creating packages for popular Open Source libraries, contributing many thousands of Pull Requests per year.
- More than two thousands Conan users hang around the CppLang Slack #conan channel, and help responding to questions, discussing problems and approaches, making it one of the most active channels in the whole CppLang slack.
- There is a Conan channel in #include<cpp> discord.

Have any questions? Please check out our *FAQ section* or .

CHAPTER

TWO

WHAT'S NEW IN CONAN 2.0

Conan 2.0 comes with many exciting improvements based on the lessons learned in the last years with Conan 1.X. Also, a lot of effort has been made to backport necessary things to Conan 1.X to make the upgrade easier: Recipes using latest 1.X integrations will be compatible with Conan 2.0, and binaries for both versions will not collide and be able to live in the same server repositories.

2.1 Conan 2.0 migration guide

If you are using Conan 1.X, please read the Conan 2.0 Migration guide , to start preparing your package recipes to 2.0 and be aware of some changes while you still work in Conan 1.X. That guide summarizes the above mentioned backports to make the upgrade easier.

2.2 New graph model

Conan 2.0 defines new requirement traits (headers, libs, build, run, test, package_id_mode, options, transitive_headers, transitive_libs) and package types (static, shared, application, header-only) to better represent the relations that happen with C and C++ binaries, like executables or shared libraries linking static libraries or shared libraries.

Read more:

- https://www.youtube.com/watch?v=kKGglzm5ous
- https://github.com/conan-io/tribe/blob/main/design/026-requirements_traits.md
- https://github.com/conan-io/tribe/blob/main/design/027-package_types.md

2.3 New public Python API

A new modular Python API is made available, public and documented. This is a real API, with building blocks that are already used to build the Conan built-in commands, but that will allow further extensions. There are subapis for different functional groups, like `api.list`, `api.search`, `api.remove`, `api.profile`, `api.graph`, `api.upload`, `api.remotes`, etc. that will allow to implement advanced user flows, functionality and automation.

Read more:

- *Python API reference*

2.4 New build system integrations

Introduced in latest Conan 1.X, Conan 2.0 will use modern build system integrations like `CMakeDeps` and `CMakeToolchain` that are fully transparent CMake integration (i.e. the consuming `CMakeLists.txt` doesn't

need to be aware at all about Conan). These integrations can also achieve a better IDE integration, for example via CMakePresets.json.

Read more:

- *Tools reference*

2.5 New custom user commands

Conan 2.0 allows extending Conan with custom user commands, written in python that can be called as `conan xxxx`. These commands can be shared and installed with `conan config install`, and have layers of commands and subcommands. The custom user comands use the new 2.0 public Python API to implement their functionality.

2.6 New CLI

Conan 2.0 has redesigned the CLI for better consistency, removing ambiguities, and improving the user experience. The new CLI also sends all the information, warning, and error messages to stderr, while keeping the final result in stdout, allowing multiple output formats like `--format=html` or `--format=json` and using redirects to create files `--format=json > myfile.json`. The information provided by the CLI will be more structured and thorough so that it can be used more easily for automation, especially in CI/CD systems.

Read more:

- *Commands reference*

2.7 New deployers

Conan 2.0 implements "deployers", which can be called in the command line as `conan install ... --deploy=mydeploy`, typically to perform copy operations from the Conan cache to user folders. Such deployers can be built-in ("full_deploy" and "direct_deploy" are provided so far), or user-defined, which can be shared and managed with `conan config install`. Deployers run before generators, and they can change the target folders. For example, if the `--deploy=full_deploy` deployer runs before `CMakeDeps`, the files generated by `CMakeDeps` will point to the local copy in the user folder done by the `full_deploy` deployer, and not to the Conan cache.

Deployers can be multi-configuration. Running `conan install . --deploy=full_deploy` repeatedly for different profiles, can achieve a fully self-contained project, including all the artifacts, binaries, and build files that is completely independent of Conan and no longer require Conan at all to build.

2.8 New package_id

Conan 2.0 defines a new, dynamic `package_id` that is a great improvement over the limitations of Conan 1.X. This `package_id` will take into account the package types and types of requirements to implement a more meaningful strategy, depending on the scenario. For example, it is well known that when an application `myapp` is linking a static library `mylib`, any change in the binary of the static library `mylib` requires re-building the application `myapp`. So Conan will default to a mode like `full_mode` that will generate a new `myapp` `package_id`, for every change in the `mylib` recipe or binary. While a dependency between a static library `mylib_a` that is used by‘‘mylib_b‘‘ in general does not imply that a change in `mylib_b` always needs a rebuild of `mylib_a`, and that relationship can default to a `minor_mode` mode. In Conan 2.0, the one doing modifications to `mylib_a` can better express whether the consumer `mylib_b` needs to rebuild or not, based on the version bump (patch version bump will not trigger a rebuild while a minor version bump will trigger it)

Furthermore the default versioning scheme in Conan has been generalized to any number of digits and letters, as opposed to the official "semver" that uses just 3 fields.

2.9 compatibility.py

Conan 2.0 features a new extension mechanism to define binary compatibility at a global level. A `compatibility.py` file in the Conan cache will be used to define which fallbacks of binaries should be used in case there is some missing binary for a given package. Conan will provide a default one to account for `cppstd` compatibility, and executables compatibility, but this extension is fully configurable by the user (and can also be shared and managed with `conan config install`)

2.10 New lockfiles

Lockfiles in Conan 2.0 have been greatly simplified and made way more flexible. Lockfiles are now modeled as lists of sorted references, which allow one single lockfile being used for multiple configurations, merging lockfiles, applying partially defined lockfiles, being strict or non-strict, adding user defined constraints to lockfiles, and much more.

Read more:

- *Tutorial introduction to lockfiles*
- https://github.com/conan-io/tribe/blob/main/design/034-new_lockfiles.md
- *Tutorial about versioning and lockfiles*

2.11 New configuration and environment management

The new configuration system called `[conf]` in profiles and command line, and introduced experimentally in Conan 1.X, is now the major mechanism to configure and control Conan behavior. The idea is that the configuration system is used to transmit information from Conan (a Conan profile) to Conan (A Conan recipe, or a Conan build system integration like `CMakeToolchain`). This new configuration system can define strings, boolean, lists, being cleaner, more structured and powerful than using environment variables. A better, more explicit environment management, also introduced in Conan 1.X is now the way to pass information from Conan (profiles) to tools (like compilers, build systems).

Read more:

- *Reference of enviroment tools*

2.12 Multi-revision cache

The Conan cache has been completely redesigned to allow storing more than one revision at a time. It has also shortened the paths, using hashes, removing the need to use `short_paths` in Windows. Note that the cache is still not concurrent, so parallel jobs or tasks should use independent caches.

2.13 New extensions plugins

Several extension points, named "plugins" have been added, to provide advanced and typically orthogonal functionality to what the Conan recipes implement. These plugins can be shared, managed and installed via `conan config install`

2.13.1 Profile checker

A new `profile.py` extension point is provided that can be used to perform operations on the profile after it has been processed. A default implementation that checks that the given compiler version is capable of supporting the given compiler `cppstd` is provided, but this is fully customizable by the user.

2.13.2 Command wrapper

A new `cmd_wrapper.py` extension provides a way to wrap any `conanfile.py` command (i.e., anything that runs inside `self.run()` in a recipe), in a new command. This functionality can be useful for wrapping build commands in build optimization tools as IncrediBuild or compile caches.

2.13.3 Package signing

A new `sign.py` extension has been added to implement signing and verifying of packages. As the awareness about the importance of software supply chain security grows, it is becoming more important the capability of being able to sign and verify software packages. This extension point will soon get a plugin implementation based on Sigstore.

2.14 Package immutability optimizations

The thorough use of `revisions` (already introduced in Conan 1.X as opt-in in https://docs.conan.io/en/latest/versioning/revisions.html) in Conan 2.0, together with the declaration of artifacts **immutability** allows for improved processes, downloading, installing and updated dependencies as well as uploading dependencies.

The `revisions` allow accurate traceability of artifacts, and thus allows better update flows. For example, it will be easier to get different binaries for different configurations from different repositories, as long as they were created from the same recipe revision.

The package transfers, uploads, downloads, will also be more efficient, based on `revisions`. As long as a given revision exists on the server or in the cache, Conan will not transfer artifacts at all for that package.

CHAPTER

THREE

INSTALL

Conan can be installed in many Operating Systems. It has been extensively used and tested in Windows, Linux (different distros), OSX, and is also actively used in FreeBSD and Solaris SunOS. There are also several additional operating systems on which it has been reported to work.

There are three ways to install Conan:

1. The preferred and **strongly recommended way to install Conan** is from PyPI, the Python Package Index, using the `pip` command.
2. There are other available installers for different systems, which might come with a bundled python interpreter, so that you don't have to install python first. Note that some of **these installers might have some limitations**, especially those created with pyinstaller (such as Windows exe & Linux deb).
3. Running Conan from sources.

3.1 Install with pip (recommended)

To install latest Conan 2.0 pre-release version using `pip`, you need a Python >= 3.6 distribution installed on your machine. Modern Python distros come with pip pre-installed. However, if necessary you can install pip by following the instructions in pip docs.

Install Conan:

```
$ pip install conan --pre
```

Important: Please READ carefully

- Make sure that your **pip** installation matches your **Python (>= 3.6)** version.
- In **Linux**, you may need **sudo** permissions to install Conan globally.
- We strongly recommend using **virtualenvs** (virtualenvwrapper works great) for everything related to Python. (check https://virtualenvwrapper.readthedocs.io/en/stable/, or https://pypi.org/project/virtualenvwrapper-win/ in Windows) With Python 3, the built-in module `venv` can also be used instead (check https://docs.python.org/3/library/venv.html). If not using a **virtualenv** it is possible that conan dependencies will conflict with previously existing dependencies, especially if you are using Python for other purposes.
- In **OSX**, especially the latest versions that may have **System Integrity Protection**, pip may fail. Try using virtualenvs, or install with another user `$ pip install --user conan`.
- Some Linux distros, such as Linux Mint, require a restart (shell restart, or logout/system if not enough) after installation, so Conan is found in the path.

3.1.1 Known installation issues with pip

- When Conan is installed with **`pip install --user <username>`**, usually a new directory is created for it. However, the directory is not appended automatically to the *PATH* and the **`conan`** commands do not work. This can usually be solved restarting the session of the terminal or running the following command:

```
$ source ~/.profile
```

3.2 Install from source

You can run Conan directly from source code. First, you need to install Python and pip.

Clone (or download and unzip) the git repository and install it.

Conan 2 is still in alpha stage, so you must check the *develop2* branch of the repository:

```
# clone folder name matters, to avoid imports issues
$ git clone https://github.com/conan-io/conan.git conan_src
$ cd conan_src
$ git fetch --all
$ git checkout -b develop2 origin/develop2
$ python -m pip install -e .
```

And test your `conan` installation:

```
$ conan
```

You should see the Conan commands help.

3.3 Update

If installed via `pip`, Conan 2.0 pre-release version can be easily updated:

```
$ pip install conan --pre --upgrade  # Might need sudo or --user
```

The default `<userhome>/.conan/settings.yml` file, containing the definition of compiler versions, etc., will be upgraded if Conan does not detect local changes, otherwise it will create a `settings.yml.new` with the new settings. If you want to regenerate the settings, you can remove the `settings.yml` file manually and it will be created with the new information the first time it is required.

The upgrade shouldn't affect the installed packages or cache information. If the cache becomes inconsistent somehow, you may want to remove its content by deleting it (`<userhome>/.conan`).

CHAPTER

FOUR

TUTORIAL

The purpose of this section is to guide you through the most important Conan features with practical examples. From using libraries already packaged by Conan, to how to package your libraries and store them in a remote server alongside all the precompiled binaries.

Important: This tutorial is part of the Conan 2.0 documentation. Conan 2.0 is still in alpha state. Some details, like the repositories and libraries used for the tutorial, will change as we update the current 1.X Conan packages to be compatible with Conan 2.0.

4.1 Consuming packages

This section shows how to build your projects using Conan to manage your dependencies. We will begin with a basic example of a C project that uses CMake and depends on the **zlib** library. This project will use a *conanfile.txt* file to declare its dependencies.

We will also cover how you can not only use 'regular' libraries with Conan but also manage tools you may need to use while building: like CMake, msys2, MinGW, etc.

Then, we will explain different Conan concepts like settings and options and how you can use them to build your projects for different configurations like Debug, Release, with static or shared libraries, etc.

Also, we will explain how to transition from the *conanfile.txt* file we used in the first example to a more powerful *conanfile.py*.

After that, we will introduce the concept of Conan build and host profiles and explain how you can use them to cross-compile your application to different platforms.

Then, in the "Introduction to versioning" we will learn about using different versions, defining requirements with version ranges, the concept of revisions and a brief introduction to lockfiles to achieve reproducibility of the dependency graph.

4.1.1 Build a simple CMake project using Conan

Let's get started with an example: We are going to create a string compressor application that uses one of the most popular C++ libraries: Zlib.

Important: In this example, we will retrieve the Zlib Conan package from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

We'll use CMake as build system in this case but keep in mind that Conan **works with any build system** and is not limited to using CMake. You can check more examples with other build systems in the *Read More section*.

Please, first clone the sources to recreate this project, you can find them in the examples2.0 repository in GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/consuming_packages/simple_cmake_project
```

We start from a very simple C language project with this structure:

```
.
├── CMakeLists.txt
└── src
    └── main.c
```

This project contains a basic *CMakeLists.txt* including the **zlib** dependency and the source code for the string compressor program in *main.c*.

Let's have a look at the *main.c* file:

Listing 1: **main.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <zlib.h>

int main(void) {
    char buffer_in [256] = {"Conan is a MIT-licensed, Open Source package manager for␣
↪C and C++ development "
                            "for C and C++ development, allowing development teams to␣
↪easily and efficiently "
                            "manage their packages and dependencies across platforms␣
↪and build systems."};
    char buffer_out [256] = {0};

    z_stream defstream;
    defstream.zalloc = Z_NULL;
    defstream.zfree = Z_NULL;
    defstream.opaque = Z_NULL;
    defstream.avail_in = (uInt) strlen(buffer_in);
    defstream.next_in = (Bytef *) buffer_in;
    defstream.avail_out = (uInt) sizeof(buffer_out);
    defstream.next_out = (Bytef *) buffer_out;

    deflateInit(&defstream, Z_BEST_COMPRESSION);
    deflate(&defstream, Z_FINISH);
    deflateEnd(&defstream);

    printf("Uncompressed size is: %lu\n", strlen(buffer_in));
    printf("Compressed size is: %lu\n", strlen(buffer_out));

    printf("ZLIB VERSION: %s\n", zlibVersion());

    return EXIT_SUCCESS;
}
```

Also, the contents of *CMakeLists.txt* are:

Listing 2: **CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.15)
project(compressor C)

find_package(ZLIB REQUIRED)

add_executable(${PROJECT_NAME} src/main.c)
target_link_libraries(${PROJECT_NAME} ZLIB::ZLIB)
```

Our application relies on the **Zlib** library. Conan, by default, tries to install libraries from a remote server called ConanCenter. You can search there for libraries and also check the available versions. In our case, after checking the available versions for Zlib we choose to use the latest available version: **zlib/1.2.11**.

The easiest way to install the **Zlib** library and find it from our project with Conan is using a *conanfile.txt* file. Let's create one with the following content:

Listing 3: **conanfile.txt**

```
[requires]
zlib/1.2.11

[generators]
CMakeDeps
CMakeToolchain
```

As you can see we added two sections to this file with a syntax similar to an *INI* file.

- **[requires]** section is where we declare the libraries we want to use in the project, in this case, **zlib/1.2.11**.
- **[generators]** section tells Conan to generate the files that the compilers or build systems will use to find the dependencies and build the project. In this case, as our project is based in *CMake*, we will use *CMakeDeps* to generate information about where the **Zlib** library files are installed and *CMakeToolchain* to pass build information to *CMake* using a *CMake* toolchain file.

Besides the *conanfile.txt*, we need a **Conan profile** to build our project. Conan profiles allow users to define a configuration set for things like the compiler, build configuration, architecture, shared or static libraries, etc. Conan, by default, will not try to detect a profile automatically, so we need to create one. To let Conan try to guess the profile, based on the current operating system and installed tools, please run:

```
conan profile detect --force
```

This will detect the operating system, build architecture and compiler settings based on the environment. It will also set the build configuration as *Release* by default. The generated profile will be stored in the Conan home folder with name *default* and will be used by Conan in all commands by default unless another profile is specified via the command line. After executing the command you should see some output similar to this but for your configuration:

```
$ conan profile detect --force
CC and CXX: /usr/bin/gcc, /usr/bin/g++
Found gcc 10
gcc>=5, using the major as version
gcc C++ standard library: libstdc++11
Detected profile:
[settings]
os=Linux
arch=x86_64
compiler=gcc
```

(continues on next page)

(continued from previous page)

```
compiler.version=10
compiler.libcxx=libstdc++11
compiler.cppstd=gnu14
build_type=Release
[options]
[tool_requires]
[env]
...
```

We will use Conan to install **Zlib** and generate the files that CMake needs to find this library and build our project. We will generate those files in the folder *build*. To do that, run:

```
$ conan install . --output-folder=build --build=missing
```

You will get something similar to this as the output of that command:

```
$ conan install . --output-folder=build --build=missing
...
-------- Computing dependency graph ----------
zlib/1.2.11: Not found in local cache, looking in remotes...
zlib/1.2.11: Checking remote: conanv2
zlib/1.2.11: Trying with 'conanv2'...
Downloading conanmanifest.txt
Downloading conanfile.py
Downloading conan_export.tgz
Decompressing conan_export.tgz
zlib/1.2.11: Downloaded recipe revision f1fadf0d3b196dc0332750354ad8ab7b
Graph root
    conanfile.txt: /home/conan/examples2/tutorial/consuming_packages/simple_cmake_
↪project/conanfile.txt
Requirements
    zlib/1.2.11#f1fadf0d3b196dc0332750354ad8ab7b - Downloaded (conanv2)

-------- Computing necessary packages ----------
Requirements
    zlib/1.2.11
↪#f1fadf0d3b196dc0332750354ad8ab7b:cdc9a35e010a17fc90bb845108cf86cfcbce64bf
↪#dd7bf2a1ab4eb5d1943598c09b616121 - Download (conanv2)

-------- Installing packages ----------

Installing (downloading, building) binaries...
zlib/1.2.11: Retrieving package cdc9a35e010a17fc90bb845108cf86cfcbce64bf from remote
↪'conanv2'
Downloading conanmanifest.txt
Downloading conaninfo.txt
Downloading conan_package.tgz
Decompressing conan_package.tgz
zlib/1.2.11: Package installed cdc9a35e010a17fc90bb845108cf86cfcbce64bf
zlib/1.2.11: Downloaded package revision dd7bf2a1ab4eb5d1943598c09b616121

-------- Finalizing install (deploy, generators) ----------
conanfile.txt: Generator 'CMakeToolchain' calling 'generate()'
conanfile.txt: Generator 'CMakeDeps' calling 'generate()'
conanfile.txt: Aggregating env generators
```

As you can see in the output, there are a couple of things that happened:

- Conan installed the *Zlib* library from the remote server we configured at the beginning of the tutorial. This server stores both the Conan recipes, which are the files that define how libraries must be built, and the binaries that can be reused so we don't have to build from sources every time.
- Conan generated several files under the **build** folder. Those files were generated by both the `CMakeToolchain` and `CMakeDeps` generators we set in the **conanfile.txt**. `CMakeDeps` generates files so that CMake finds the Zlib library we have just downloaded. On the other side, `CMakeToolchain` generates a toolchain file for CMake so that we can transparently build our project with CMake using the same settings that we detected for our default profile.

Now we are ready to build and run our **compressor** app:

Listing 4: Windows

```
$ cd build
# assuming Visual Studio 15 2017 is your VS version and that it matches your default␣
↪profile
$ cmake .. -G "Visual Studio 15 2017" -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build . --config Release
...
[100%] Built target compressor
$ Release\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

Listing 5: Linux, macOS

```
$ cd build
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
...
[100%] Built target compressor
$ ./compressor
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

Read more

- *Getting started with Meson*
- Getting started with Autotools
- ...

4.1.2 Using build tools as Conan packages

Important: In this example, we will retrieve the CMake Conan package from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

In the previous example, we built our CMake project and used Conan to install and locate the **Zlib** library. Conan used the CMake version found in the system path to build this example. But, what happens if you don't have CMake installed in your build environment or want to build your project with a specific CMake version different from the

one you have already installed system-wide? In this case, you can declare this dependency in Conan using a type of requirement named `tool_requires`. Let's see an example of how to add a `tool_requires` to our project, and use a different CMake version to build it.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository in GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/consuming_packages/tool_requires
```

The structure of the project is the same as the one of the previous example:

```
.
├── conanfile.txt
├── CMakeLists.txt
└── src
    └── main.c
```

The main difference is the addition of the **[tool_requires]** section in the **conanfile.txt** file. In this section, we declare that we want to build our application using CMake **v3.19.8**.

Listing 6: **conanfile.txt**

```
[requires]
zlib/1.2.11

[tool_requires]
cmake/3.19.8

[generators]
CMakeDeps
CMakeToolchain
```

We also added a message to the *CMakeLists.txt* to output the CMake version:

Listing 7: **CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.15)
project(compressor C)

find_package(ZLIB REQUIRED)

message("Building with CMake version: ${CMAKE_VERSION}")

add_executable(${PROJECT_NAME} src/main.c)
target_link_libraries(${PROJECT_NAME} ZLIB::ZLIB)
```

Now, as in the previous example, we will use Conan to install **Zlib** and **CMake 3.19.8** and generate the files to find both of them. We will generate those files the folder *build*. To do that, just run:

```
$ conan install . --output-folder=build --build=missing
```

You can check the output:

```
-------- Computing dependency graph ----------
cmake/3.19.8: Not found in local cache, looking in remotes...
cmake/3.19.8: Checking remote: conanv2
cmake/3.19.8: Trying with 'conanv2'...
Downloading conanmanifest.txt
```

(continues on next page)

(continued from previous page)

```
Downloading conanfile.py
cmake/3.19.8: Downloaded recipe revision 3e3d8f3a848b2a60afafbe7a0955085a
Graph root
    conanfile.txt: /Users/carlosz/Documents/developer/conan/examples2/tutorial/
↪consuming_packages/tool_requires/conanfile.txt
Requirements
    zlib/1.2.11#f1fadf0d3b196dc0332750354ad8ab7b - Cache
Build requirements
    cmake/3.19.8#3e3d8f3a848b2a60afafbe7a0955085a - Downloaded (conanv2)

-------- Computing necessary packages ----------
Requirements
    zlib/1.2.11
↪#f1fadf0d3b196dc0332750354ad8ab7b:2a823fda5c9d8b4f682cb27c30caf4124c5726c8
↪#48bc7191ec1ee467f1e951033d7d41b2 - Cache
Build requirements
    cmake/3.19.8
↪#3e3d8f3a848b2a60afafbe7a0955085a:f2f48d9745706caf77ea883a5855538256e7f2d4
↪#6c519070f013da19afd56b52c465b596 - Download (conanv2)

-------- Installing packages ----------

Installing (downloading, building) binaries...
cmake/3.19.8: Retrieving package f2f48d9745706caf77ea883a5855538256e7f2d4 from remote
↪'conanv2'
Downloading conanmanifest.txt
Downloading conaninfo.txt
Downloading conan_package.tgz
Decompressing conan_package.tgz
cmake/3.19.8: Package installed f2f48d9745706caf77ea883a5855538256e7f2d4
cmake/3.19.8: Downloaded package revision 6c519070f013da19afd56b52c465b596
zlib/1.2.11: Already installed!

-------- Finalizing install (deploy, generators) ----------
conanfile.txt: Generator 'CMakeToolchain' calling 'generate()'
conanfile.txt: Generator 'CMakeDeps' calling 'generate()'
conanfile.txt: Aggregating env generators
```

Now, if you check the folder you will see that Conan generated a new file called `conanbuild.sh/bat`. This is the result of automatically invoking a `VirtualBuildEnv` generator when we declared the `tool_requires` in the **conanfile.txt**. This file sets some environment variables like a new `PATH` that we can use to inject to our environment the location of CMake v3.19.8.

Activate the virtual environment, and run `cmake --version` to check that you have installed the new CMake version in the path.

Listing 8: Windows

```
$ cd build
$ conanbuild.bat
```

Listing 9: Linux, macOS

```
$ cd build
$ source conanbuild.sh
Capturing current environment in deactivate_conanbuildenv-release-x86_64.sh
Configuring environment variables
```

Run `cmake` and check the version:

```
$ cmake --version
cmake version 3.19.8
...
```

As you can see, after activating the environment, the CMake v3.19.8 binary folder was added to the path and is the currently active version now. Now you can build your project as you previously did, but this time Conan will use CMake 3.19.8 to build it:

Listing 10: Windows

```
# assuming Visual Studio 15 2017 is your VS version and that it matches your default␣
↪profile
$ cmake .. -G "Visual Studio 15 2017" -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build . --config Release
...
Building with CMake version: 3.19.8
...
[100%] Built target compressor
$ Release\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

Listing 11: Linux, macOS

```
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
...
Building with CMake version: 3.19.8
...
[100%] Built target compressor
$ ./compressor
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

Note that when we activated the environment, a new file named `deactivate_conanbuild.sh/bat` was created in the same folder. If you source this file you can restore the environment as it was before.

Listing 12: Windows

```
$ deactivate_conanbuild.bat
```

Listing 13: Linux, macOS

```
$ source deactivate_conanbuild.sh
Restoring environment
```

Run `cmake` and check the version, it will be the version that was installed previous to the environment activation:

```
$ cmake --version
cmake version 3.22.0
...
```

Read more

- Using MinGW as tool_requires
- Using tool_requires in profiles
- Using conf to set a toolchain from a tool requires
- Creating recipes for tool_requires: packaging build tools

4.1.3 Building for multiple configurations: Release, Debug, Static and Shared

Important: In this example, we will retrieve the CMake Conan package from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository in GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/consuming_packages/different_configurations
```

So far, we built a simple CMake project that depended on the **zlib** library and learned about `tool_requires`, a special type or requirements for build-tools like CMake. In both cases, we did not specify anywhere that we wanted to build the application in *Release* or *Debug* mode, or if we wanted to link against *static* or *shared* libraries. That is because Conan, if not instructed otherwise, will use a default configuration declared in the 'default profile'. This default profile was created in the first example when we run the **`conan profile detect`** command. Conan stores this file in the **/profiles** folder, located in the Conan user home. You can check the contents of your default profile by running the **`conan config home`** command to get the location of the Conan user home and then showing the contents of the default profile in the **/profiles** folder:

```
$ conan config home
Current Conan home: /Users/tutorial_user/.conan2

# output the file contents
$ cat /Users/tutorial_user/.conan2/profiles/default
[settings]
os=Macos
arch=x86_64
compiler=apple-clang
compiler.version=13.0
compiler.libcxx=libc++
compiler.cppstd=gnu98
build_type=Release
[options]
[tool_requires]
[env]
```

As you can see, the profile has different sections. The `[settings]` section is the one that has information about things like the operating system, architecture, compiler, and build configuration. When you call a Conan command setting the `--profile` argument, Conan will take all the information from the profile and apply it to the packages you want to build or install. If you don't specify that argument it's equivalent to call it with `--profile=default`. These two commands will behave the same:

```
$ conan install . --build=missing
$ conan install . --build=missing --profile=default
```

You can store different profiles and use them to build for different settings. For example, to use a `build_type=Debug`, or adding a `tool_requires` to all the packages you build with that profile. One example of a *debug* profile could be:

Listing 14: <conan home>/profiles/debug

```
[settings]
os=Macos
arch=x86_64
compiler=apple-clang
compiler.version=13.0
compiler.libcxx=libc++
compiler.cppstd=gnu98
build_type=Debug
```

Modifying settings: use Debug configuration for the application and its dependencies

Using profiles is not the only way to set the configuration you want to use. You can also override the profile settings in the Conan command using the `--settings` argument. For example, you can build the project from the previous examples in *Debug* configuration instead of *Release*.

Before building, please check that we modified the source code from the previous example to show the build configuration the sources were built with:

```
#include <stdlib.h>
...

int main(void) {
    ...
    #ifdef NDEBUG
    printf("Release configuration!\n");
    #else
    printf("Debug configuration!\n");
    #endif

    return EXIT_SUCCESS;
}
```

Now let's build our project for *Debug* configuration:

```
$ conan install . --output-folder=build --build=missing --settings=build_type=Debug
```

As we explained above, this is the equivalent of having *debug* profile and running these command using the `--profile=debug` argument instead of the `--settings=build_type=Debug` argument.

This **`conan install`** command will check if we already installed the required libraries (Zlib) in Debug configuration and install them otherwise. It will also set the build configuration in the `conan_toolchain.cmake` toolchain that the CMakeToolchain generator creates so that when we build the application it's built in *Debug* configuration. Now build your project as you did in the previous examples and check in the output how it was built in *Debug* configuration:

Listing 15: Windows

```
# assuming Visual Studio 15 2017 is your VS version and that it matches your default␣
↪profile
$ cd build
$ cmake .. -G "Visual Studio 15 2017" -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build . --config Debug
```

(continues on next page)

(continued from previous page)

```
$ Debug\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
Debug configuration!
```

Listing 16: Linux, macOS

```
$ cd build
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Debug
$ cmake --build .
$ ./compressor
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
Debug configuration!
```

Modifying options: linking the application dependencies as shared libraries

So far, we have been linking *Zlib* statically in our application. That's because in the Zlib's Conan package there's an attribute set to build in that mode by default. We can change from **static** to **shared** linking by setting the `shared` option to `True` using the `--options` argument. To do so, please run:

Listing 17: Windows

```
$ conan install . --output-folder=build --build=missing --options=zlib/1.2.
↪11:shared=True
```

Doing this, Conan will install the *Zlib* shared libraries, generate the files to build with them and, also the necessary files to locate those dynamic libraries when running the application. Let's build the application again after configuring it to link *Zlib* as a shared library:

Listing 18: Windows

```
$ cd build
# assuming Visual Studio 15 2017 is your VS version and that it matches your default␣
↪profile
$ cmake .. -G "Visual Studio 15 2017" -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build . --config Release
...
[100%] Built target compressor
```

Listing 19: Linux, Macos

```
$ cd build
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
...
[100%] Built target compressor
```

Now, if you try to run the compiled executable you will see an error because the executable can't find the shared libraries for *Zlib* that we just installed.

Listing 20: Windows

```
$ Release\compressor.exe
(on a pop-up window) The code execution cannot proceed because zlib1.dll was not␣
↪found. Reinstalling the program may fix this problem.
```

Listing 21: Linux, Macos

```
$ ./compressor
./compressor: error while loading shared libraries: libz.so.1: cannot open shared␣
↪object file: No such file or directory
```

This is because shared libraries (*.dll* in windows, *.dylib* in OSX and *.so* in Linux), are loaded at runtime. That means that the application executable needs to know where are the required shared libraries when it runs. On Windows, the dynamic linker will search in the same directory then in the *PATH* directories. On OSX, it will search in the directories declared in *DYLD_LIBRARY_PATH* as on Linux will use the *LD_LIBRARY_PATH*.

Conan provides a mechanism to define those variables and make it possible, for executables, to find and load these shared libraries. This mechanism is the `VirtualRunEnv` generator. If you check the output folder you will see that Conan generated a new file called `conanrun.sh/bat`. This is the result of automatically invoking that `VirtualRunEnv` generator when we activated the `shared` option when doing the **`conan install`**. This generated script will set the **PATH**, **LD_LIBRARY_PATH**, **DYLD_LIBRARY_PATH** and **DYLD_FRAMEWORK_PATH** environment variables so that executables can find the shared libraries.

Activate the virtual environment, and run the executables again:

Listing 22: Windows

```
$ conanrun.bat
$ Release\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
...
```

Listing 23: Linux, macOS

```
$ source conanrun.sh
$ ./compressor
Uncompressed size is: 233
Compressed size is: 147
...
```

Just as in the previous example with the `VirtualBuildEnv` generator, when we run the `conanrun.sh/bat` script a deactivation script called `deactivate_conanrun.sh/bat` is created to restore the environment. Source or run it to do so:

Listing 24: Windows

```
$ deactivate_conanrun.bat
```

Listing 25: Linux, macOS

```
$ source deactivate_conanrun.sh
```

Difference between settings and options

You may have noticed that for changing between *Debug* and *Release* configuration we used a Conan **setting**, but when we set *shared* mode for our executable we used a Conan **option**. Please, note the difference between **settings** and **options**:

- **settings** are typically a project-wide configuration defined by the client machine. Things like the operating system, compiler or build configuration that will be common to several Conan packages and would not make sense to define one default value for only one of them. For example, it doesn't make sense for a Conan package to declare "Visual Studio" as a default compiler because that is something defined by the end consumer, and unlikely to make sense if they are working in Linux.
- **options** are intended for package-specific configuration that can be set to a default value in the recipe. For example, one package can define that its default linkage is static, and this is the linkage that should be used if consumers don't specify otherwise.

Read more

- Installing configurations with conan config install
- VS Multi-config
- Example about how settings and options influence the package id
- Cross-compiling using –profile:build and –profile:host
- Using patterns for settings and options

4.1.4 Understanding the flexibility of using conanfile.py vs conanfile.txt

Important: In this example, we will retrieve Conan packages from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

In the previous examples, we declared our dependencies (*Zlib* and *CMake*) in a *conanfile.txt* file. Let's have a look at that file:

Listing 26: **conanfile.txt**

```
[requires]
zlib/1.2.11

[tool_requires]
cmake/3.19.8

[generators]
CMakeDeps
CMakeToolchain
```

Using a *conanfile.txt* to build your projects using Conan it's enough for simple cases, but if you need more flexibility you should use a *conanfile.py* file where you can use Python code to make things such as adding requirements dynam-

ically, changing options depending on other options or setting options for your requirements. Let's see an example on how to migrate to a *conanfile.py* and use some of those features.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository in GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/consuming_packages/conanfile_py
```

Check the contents of the folder and note that the contents are the same that in the previous examples but with a *conanfile.py* instead of a *conanfile.txt*.

```
.
├── CMakeLists.txt
├── conanfile.py
└── src
    └── main.c
```

Remember that in the previous examples the *conanfile.txt* had this information:

Listing 27: **conanfile.txt**

```
[requires]
zlib/1.2.11

[tool_requires]
cmake/3.19.8

[generators]
CMakeDeps
CMakeToolchain
```

We will translate that same information to a *conanfile.py*. This file is what is typically called a **"Conan recipe"**. It can be used for consuming packages, like in this case, and also to create packages. For our current case, it will define our requirements (both libraries and build tools) and logic to modify options and set how we want to consume those packages. In the case of using this file to create packages, it can define (among other things) how to download the package's source code, how to build the binaries from those sources, how to package the binaries, and information for future consumers on how to consume the package. We will explain how to use Conan recipes to create packages in the "Creating Packages" section later.

The equivalent of the *conanfile.txt* in form of Conan recipe could look like this:

Listing 28: **conanfile.py**

```
from conan import ConanFile


class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

    def requirements(self):
        self.requires("zlib/1.2.11")
        self.tool_requires("cmake/3.19.8")
```

To create the Conan recipe we declared a new class that inherits from the `ConanFile` class. This class has different class attributes and methods:

- **settings** this class attribute defines the project-wide variables, like the compiler, its version, or the OS itself that may change when we build our project. This is related to how Conan manages binary compatibility as these

values will affect the value of the **package ID** for Conan packages. We will explain how Conan uses this value to manage binary compatibility later.

- **generators** this class attribute specifies which Conan generators will be run when we call the **`conan install`** command. In this case, we added **CMakeToolchain** and **CMakeDeps** as in the *conanfile.txt*.
- **requirements()** in this method we can use the `self.requires()` and `self.tool_requires()` methods to declare all our dependencies (libraries and build tools).

You can check that running the same commands as in the previous examples will lead to the same results as before.

Listing 29: Windows

```
$ conan install . --output-folder=build --build=missing
$ cd build
$ conanbuild.bat
# assuming Visual Studio 15 2017 is your VS version and that it matches your default␣
↪profile
$ cmake .. -G "Visual Studio 15 2017" -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build . --config Release
...
Building with CMake version: 3.19.8
...
[100%] Built target compressor

$ Release\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
$ deactivate_conanbuild.bat
```

Listing 30: Linux, macOS

```
$ conan install . --output-folder build --build=missing
$ cd build
$ source conanbuild.sh
Capturing current environment in deactivate_conanbuildenv-release-x86_64.sh
Configuring environment variables
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
...
Building with CMake version: 3.19.8
...
[100%] Built target compressor

$ ./compressor
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
$ source deactivate_conanbuild.sh
```

So far we have achieved the same functionality we had using a *conanfile.txt*, let's see how we can take advantage of the capabilities of the *conanfile.py* to define the project structure we want to follow and also to add some logic using Conan settings and options.

Conditional requirements using a conanfile.py

You could add some logic to the *requirements()* method to add or remove requirements conditionally. Imagine, for example, that you want to add an additional dependency in Windows or that you want to use the system's CMake

installation instead of using the Conan *tool_requires*:

Listing 31: conanfile.py

```
from conan import ConanFile


class CompressorRecipe(ConanFile):
    # Binary configuration
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

    def requirements(self):
        self.requires("zlib/1.2.11")
        # Use the system's CMake for Windows
        # and add base64 dependency
        if self.settings.os == "Windows":
            self.requires("base64/0.4.0")
        else:
            self.tool_requires("cmake/3.19.8")
```

Use the layout() method

In the previous examples, every time we executed a *conan install* command we had to use the *–output-folder argument* to define where we wanted to create the files that Conan generates. Also, note that we used a different folder when building in Windows or in Linux/Macos depending if we were using a multi-config CMake generator or not. You can define this directly in the *conanfile.py* inside the *layout()* method and make it work for every platform without adding more changes:

Listing 32: **conanfile.py**

```
from conan import ConanFile


class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

    def requirements(self):
        self.requires("zlib/1.2.11")
        self.tool_requires("cmake/3.19.8")

    def layout(self):
        # We make the assumption that if the compiler is msvc the
        # CMake generator is multi-config
        if self.settings.get_safe("compiler") == "msvc":
            multi = True
        else:
            multi = False

        self.folders.build = "build" if multi else f"build/{str(self.settings.build_
↪type)}"
        self.folders.generators = "build"
```

As you can see, we defined two different attributes for the Conanfile in the *layout()* method:

- **self.folders.build** is the folder where the resulting binaries will be placed. The location depends on the type of CMake generator. For multi-config, they will be located in a dedicated folder inside the build folder, while for

single-config, they will be located directly in the build folder.

- **self.folders.generators** is the folder where all the auxiliary files generated by Conan (CMake toolchain and cmake dependencies files) will be placed.

Note that he definitions of the folders is different if it is a multi-config generator (like Visual Studio), or a single-config generator (like Unix Makefiles). In the first case, the folder is the same irrespective of the build type, and the build system will manage the different build types inside that folder. But single-config generators like Unix Makefiles, must use a different folder for each different configuration (as a different build_type Release/Debug). In this case we added a simple logic to consider multi-config if the compiler name is *msvc*.

Check that running the same commands as in the previous examples without the *–output-folder* argument will lead to the same results as before:

Listing 33: Windows

```
$ conan install . --build=missing
$ cd build
$ conanbuild.bat
# assuming Visual Studio 15 2017 is your VS version and that it matches your default␣
↪profile
$ cmake .. -G "Visual Studio 15 2017" -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build . --config Release
...
Building with CMake version: 3.19.8
...
[100%] Built target compressor

$ Release\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
$ deactivate_conanbuild.bat
```

Listing 34: Linux, macOS

```
$ conan install . --build=missing
$ cd build
$ source conanbuild.sh
Capturing current environment in deactivate_conanbuildenv-release-x86_64.sh
Configuring environment variables
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
...
Building with CMake version: 3.19.8
...
[100%] Built target compressor

$ ./Release/compressor
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
$ source deactivate_conanbuild.sh
```

There's no need to always write this logic in the *conanfile.py*. There are some pre-defined layouts you can import and directly use in your recipe. For example, for the CMake case, there's a *cmake_layout()* already defined in Conan:

Listing 35: **conanfile.py**

```
from conan import ConanFile
from conan.tools.cmake import cmake_layout


class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

    def requirements(self):
        self.requires("zlib/1.2.11")
        self.tool_requires("cmake/3.19.8")

    def layout(self):
        cmake_layout(self)
```

Use the validate() method to raise an error for non-supported configurations

The *validate()* method is evaluated when Conan loads the *conanfile.py* and you can use it to perform checks of the input settings. If, for example, your project does not support *armv8* architecture on Macos you can raise the *ConanInvalidConfiguration* exception to make Conan return with a special error code. This will indicate that the configuration used for settings or options is not supported.

Use the objects `self.info.settings` and `self.info.options` to read the configuration, otherwise, the "compatible" packages (method `compatibility()` and plugin `compatibility.py` won't be able to verify if the potentially compatible configurations are valid or not.

Listing 36: conanfile.py

```
...
from conan.errors import ConanInvalidConfiguration

class CompressorRecipe(ConanFile):
    ...

    def validate(self):
        if self.info.settings.os == "Macos" and self.info.settings.arch == "armv8":
            raise ConanInvalidConfiguration("ARM v8 not supported")
```

Read more

- *Using "cmake_layout" + "CMakeToolchain" + "CMakePresets feature" to build your project*.
- Importing resource files in the generate() method
- Layouts advanced use
- Conditional generators in configure()

4.1.5 How to cross-compile your applications using Conan: host and build contexts

Important: In this example, we will retrieve Conan packages from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configura-

tion (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/consuming_packages/cross_building
```

In the previous examples, we learned how to use a *conanfile.py* or *conanfile.txt* to build an application that compresses strings using the *Zlib* and *CMake* Conan packages. Also, we explained that you can set information like the operating system, compiler or build configuration in a file called the Conan profile. You can use that profile as an argument (**`--profile`**) to invoke the **`conan install`**. We also explained that not specifying that profile is equivalent to using the **`--profile=default`** argument.

For all those examples, we used the same platform for building and running the application. But, what if you want to build the application on your machine running Ubuntu Linux and then run it on another platform like a Raspberry Pi? Conan can model that case using two different profiles, one for the machine that **builds** the application (Ubuntu Linux) and another for the machine that **runs** the application (Raspberry Pi). We will explain this "two profiles" approach in the next section.

Conan two profiles model: build and host profiles

Even if you specify only one **`--profile`** argument when invoking Conan, Conan will internally use two profiles. One for the machine that **builds** the binaries (called the **build** profile) and another for the machine that **runs** those binaries (called the **host** profile). Calling this command:

```
$ conan install . --build=missing --profile=someprofile
```

Is equivalent to:

```
$ conan install . --build=missing --profile:host=someprofile --profile:build=default
```

As you can see we used two new arguments:

- `profile:host`: This is the profile that defines the platform where the built binaries will run. For our string compressor application this profile would be the one applied for the *Zlib* library that will run in a **Raspberry Pi**.
- `profile:build`: This is the profile that defines the platform where the binaries will be built. For our string compressor application, this profile would be the one used by the *CMake* tool that will compile it on the **Ubuntu Linux** machine.

Note that when you just use one argument for the profile `--profile` is equivalent to `--profile:host`. If you don't specify the `--profile:build` argument, Conan will use the *default* profile internally.

So, if we want to build the compressor application in the Ubuntu Linux machine but run it in a Raspberry Pi, we should use two different profiles. For the **build** machine we could use the default profile, that in our case looks like this:

Listing 37: <conan home>/profiles/default

```
[settings]
os=Linux
arch=x86_64
build_type=Release
compiler=gcc
compiler.cppstd=gnu14
compiler.libcxx=libstdc++11
compiler.version=9
```

And the profile for the Raspberry Pi that is the **host** machine:

Listing 38: <local folder>/profiles/raspberry

```
[settings]
os=Linux
arch=armv7hf
compiler=gcc
build_type=Release
compiler.cppstd=gnu14
compiler.libcxx=libstdc++11
compiler.version=9
[buildenv]
CC=arm-linux-gnueabihf-gcc-9
CXX=arm-linux-gnueabihf-g++-9
LD=arm-linux-gnueabihf-ld
```

Important: Please, take into account that in order to build this example successfully, you should have installed a toolchain that includes the compiler and all the tools to build the application for the proper architecture. In this case the host machine is a Raspberry Pi 3 with *armv7hf* architecture operating system and we have the *arm-linux-gnueabihf* toolchain installed in the Ubuntu machine.

If you have a look at the *raspberry* profile, there is a section named `[buildenv]`. This section is used to set the environment variables that are needed to build the application. In this case we declare the `CC`, `CXX` and `LD` variables pointing to the cross-build toolchain compilers and linker, respectively. Adding this section to the profile will invoke the VirtualBuildEnv generator everytime we do a **`conan install`**. This generator will add that environment information to the `conanbuild.sh` script that we will source before building with CMake so that it can use the cross-build toolchain.

Build and host contexts

Now that we have our two profiles prepared, let's have a look at our *conanfile.py*:

Listing 39: **conanfile.py**

```
from conan import ConanFile
from conan.tools.cmake import cmake_layout

class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

    def requirements(self):
        self.requires("zlib/1.2.11")
        self.tool_requires("cmake/3.19.8")

    def layout(self):
        cmake_layout(self)
```

As you can see, this is practically the same *conanfile.py* we used in the *previous example*. We will require **zlib/1.2.11** as a regular dependency and **cmake/3.19.8** as a tool needed for building the application.

We will need the application to build for the Raspberry Pi with the cross-build toolchain and also link the **zlib/1.2.11** library built for the same platform. On the other side, we need the **cmake/3.19.8** binary to run in Ubuntu Linux. Conan manages this internally in the dependency graph differentiating between what we call the "build context" and the "host context":

- The **host context** is populated with the root package (the one specified in the **conan install** or **conan create** command) and all its requirements added via `self.requires()`. In this case, this includes the compressor application and the **zlib/1.2.11** dependency.
- The **build context** contains the tool requirements used in the build machine. This category typically includes all the developer tools like CMake, compilers and linkers. In this case, this includes the **cmake/3.19.8** tool.

These contexts define how Conan will manage each one of the dependencies. For example, as **zlib/1.2.11** belongs to the **host context**, the `[buildenv]` build environment we defined in the **raspberry** profile (profile host) will only apply to the **zlib/1.2.11** library when building and won't affect anything that belongs to the **build context** like the **cmake/3.19.8** dependency.

Now, let's build the application. First, call **conan install** with the profiles for the build and host platforms. This will install the **zlib/1.2.11** dependency built for *armv7hf* architecture and a **cmake/3.19.8** version that runs for 64-bit architecture.

```
$ conan install . --build missing -pr:b=default -pr:h=./profiles/raspberry
```

Then, let's call CMake to build the application. As we did in the previous example we have to activate the **build environment** running `source generators/conanbuild.sh`. That will set the environment variables needed to locate the cross-build toolchain and build the application.

```
$ cd build
$ source generators/conanbuild.sh
Capturing current environment in deactivate_conanbuildenv-release-armv7hf.sh
Configuring environment variables
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=generators/conan_toolchain.cmake -DCMAKE_BUILD_
↪TYPE=Release
$ cmake --build .
...
-- Conan toolchain: C++ Standard 14 with extensions ON
-- The C compiler identification is GNU 9.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/arm-linux-gnueabihf-gcc-9 - skipped
-- Detecting C compile features
-- Detecting C compile features - done    [100%] Built target compressor
...
$ source generators/deactivate_conanbuild.sh
```

You could check that we built the application for the correct architecture by running the `file` Linux utility:

```
$ file compressor
compressor: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux-armhf.so.3,
BuildID[sha1]=2a216076864a1b1f30211debf297ac37a9195196, for GNU/Linux 3.2.0, not
stripped
```

Read more

- *Cross building to Android with the NDK*
- Cross-build using a tool_requires
- How to require test frameworks like gtest: using `test_requires`
- Using Conan to build for iOS
- Link to the VirtualBuildEnv reference

4.1.6 Introduction to versioning

Important: In this example, we will retrieve Conan packages from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

So far we have been using requires with fixed versions like `requires = "zlib/1.2.12"`. But sometimes dependencies evolve, new versions are released and consumers want to update to those versions as easy as possible.

It is always possible to edit the `conanfiles` and explicitly update the versions to the new ones, but there are mechanisms in Conan to allow such updates without even modifying the recipes.

Version ranges

A `requires` can express a dependency to a certain range of versions for a given package, with the syntax `pkgname/[version-range-expression]`. Lets see an example, please, first clone the sources to recreate this project. You can find them in the examples2.0 repository in GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/consuming_packages/versioning
```

We can see that we have there:

Listing 40: **conanfile.py**

```
from conan import ConanFile


class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

    def requirements(self):
        self.requires("zlib/[~1.2]")
```

That `requires` contains the expression `zlib/[~1.2]`, which means "approximately" `1.2` version, that means, it can resolve to any `zlib/1.2.8`, `zlib/1.2.11` or `zlib/1.2.12`, but it will not resolve to something like `zlib/1.3.0`. Among the available matching versions, a version range will always pick the latest one.

If we do a **`conan install`**, we would see something like:

```
$ conan install .

Graph root
    conanfile.py: .../conanfile.py
Requirements
    zlib/1.2.12#87a7211557b6690ef5bf7fc599dd8349 - Downloaded
Resolved version ranges
    zlib/[~1.2]: zlib/1.2.12
```

If we tried instead to use `zlib/[<1.2.12]`, that means that we would like to use a version lower than `1.2.12`, but that one is excluded, so the latest one to satisfy the range would be `zlib/1.2.11`:

```
$ conan install .
```

(continues on next page)

(continued from previous page)

```
Resolved version ranges
    zlib/[<1.2.12]: zlib/1.2.11
```

The same applies to other type of requirements, like `tool_requires`. If we add now to the recipe:

Listing 41: **conanfile.py**

```
from conan import ConanFile


class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

    def requirements(self):
        self.requires("zlib/[~1.2]")
        self.tool_requires("cmake/[>3.10]")
```

Then we would see it resolved to the latest available CMake package, with at least version `3.11`:

```
$ conan install .
...
Graph root
    conanfile.py: .../conanfile.py
Requirements
    zlib/1.2.12#87a7211557b6690ef5bf7fc599dd8349 - Cache
Build requirements
    cmake/3.19.8#f305019023c2db74d1001c5afa5cf362 - Downloaded
Resolved version ranges
    cmake/[>3.10]: cmake/3.19.8
    zlib/[~1.2]: zlib/1.2.12
```

Revisions

What happens when a package creator does some change to the package recipe or to the source code, but they don't bump the `version` to reflect those changes? Conan has an internal mechanism to keep track of those modifications, and it is called the **revisions**.

The recipe revision is the hash that can be seen together with the package name and version in the form `pkgname/version#recipe_revision` or `pkgname/version@user/channel#recipe_revision`. The recipe revision is a hash of the contents of the recipe and the source code. So if something changes either in the recipe, its associated files or in the source code that this recipe is packaging, it will create a new recipe revision.

You can list existing revisions with the **`conan list`** command:

```
conan list recipe-revisions zlib/1.2.12 -r=conanv2

conanv2:
    zlib/1.2.12#87a7211557b6690ef5bf7fc599dd8349 (2022-04-21 11:01:59 UTC)
```

Revisions always resolve to the latest (chronological order of creation or upload to the server) revision. Though it is not a common practice, it is possible to explicitly pin a given recipe revision directly in the `conanfile`, like:

```
def requirements(self):
    self.requires("zlib/1.2.12#87a7211557b6690ef5bf7fc599dd8349")
```

This mechanism can however be tedious to maintain and update when new revisions are created, so probably in the general case, this shouldn't be done.

Lockfiles

The usage of version ranges, and the possibility of creating new revisions of a given package without bumping the version allows to do automatic faster and more convenient updates, without need to edit recipes.

But in some occassions, there is also a need to provide an immutable and reproducible set of dependencies. This process is known as "locking", and the mechanism to allow it is "lockfile" files. A lockfile is a file that contains a fixed list of dependencies, specifying the exact version and exact revision. So, for example, a lockfile will never contain a version range with an expression, but only pinned dependencies.

A lockfile can be seen as a snapshot of a given dependency graph at some point in time. Such snapshot must be "realizable", that is, it needs to be a state that can be actually reproduce from the conanfile recipes. And this lockfile can be used at a later point in time to force that same state, even if there are new created package versions.

Lets see lockfiles in action. First, lets pin the dependency to `zlib/1.2.11` in our example:

```
def requirements(self):
    self.requires("zlib/1.2.11")
```

And lets capture a lockfile:

```
conan lock create .

-------- Computing dependency graph ----------
Graph root
    conanfile.py: .../conanfile.py
Requirements
    zlib/1.2.11#4524fcdd41f33e8df88ece6e755a5dcc - Cache

Generated lockfile: .../conan.lock
```

Lets see what the lockfile `conan.lock` contains:

```
{
    "version": "0.5",
    "requires": [
        "zlib/1.2.11#4524fcdd41f33e8df88ece6e755a5dcc%1650538915.154"
    ],
    "build_requires": [],
    "python_requires": []
}
```

Now, lets restore the original `requires` version range:

```
def requirements(self):
    self.requires("zlib/[~1.2]")
```

And run **`conan install .`**, which by default will find the `conan.lock`, and run the equivalent **`conan install . --lockfile=conan.lock`**

```
conan install .

Graph root
    conanfile.py: .../conanfile.py
Requirements
    zlib/1.2.11#4524fcdd41f33e8df88ece6e755a5dcc - Cache
```

Note how the version range is no longer resolved, and it doesn't get the `zlib/1.2.12` dependency, even if it is the allowed range `zlib/[~1.2]`, because the `conan.lock` lockfile is forcing it to stay in `zlib/1.2.11` and that exact revision too.

Read more

- *Introduction to Versioning and Continuous Integration*

4.2 Creating packages

This section shows how to create Conan packages using a Conan recipe. We begin by creating a basic Conan recipe to package a simple C++ library that you can scaffold using the **`conan new`** command. Then, we will explain the different methods that you can define inside a Conan recipe and the things you can do inside them:

- Using the `source()` method to retrieve sources from external repositories and apply patches to those sources.
- Add requirements to your Conan packages inside the `requirements()` method.
- Use the `generate()` method to prepare the package build, and customize the toolchain.
- Configure settings and options in the `configure()` and `config_options()` methods and how they affect the packages' binary compatibility.
- Use the `build()` method to customize the build process and launch the tests for the library you are packaging.
- Select which files will be included in the Conan package using the `package()` method.
- Define the package information in the `package_info()` method so that consumers of this package can use it.

After this walkthrough around some Conan recipe methods, we will explain some peculiarities of different types of Conan packages like, for example, header-only libraries, packages for pre-built binaries, packaging tools for building other packages or packaging your own applications.

4.2.1 Create your first Conan package

In previous sections, we *consumed* Conan packages (like the *Zlib* one), first using a *conanfile.txt* and then with a *conanfile.py*. But a *conanfile.py* recipe file is not only meant to consume other packages, it can be used to create your own packages as well. In this section, we explain how to create a simple Conan package with a *conanfile.py* recipe and how to use Conan commands to build those packages from sources.

Important: This is a **tutorial** section. You are encouraged to execute these commands. For this concrete example, you will need **CMake** installed in your path. It is not strictly required by Conan to create packages, you can use other build systems (such as VS, Meson, Autotools, and even your own) to do that, without any dependency on CMake.

Use the **`conan new`** command to create a "Hello World" C++ library example project:

```
$ conan new cmake_lib -d name=hello -d version=1.0
```

This will create a Conan package project with the following structure.

```
.
├── CMakeLists.txt
├── conanfile.py
├── include
│   └── hello.h
├── src
```

(continues on next page)

(continued from previous page)

```
│   └── hello.cpp
└── test_package
    ├── CMakeLists.txt
    ├── conanfile.py
    └── src
        └── example.cpp
```

The generated files are:

- **conanfile.py**: On the root folder, there is a *conanfile.py* which is the main recipe file, responsible for defining how the package is built and consumed.
- **CMakeLists.txt**: A simple generic *CMakeLists.txt*, with nothing specific about Conan in it.
- **src** folder: the *src* folder that contains the simple C++ "hello" library.
- **test_package** folder: contains an *example* application that will require and link with the created package. It is not mandatory, but it is useful to check that our package is correctly created.

Let's have a look at the package recipe *conanfile.py*:

```
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    # Optional metadata
    license = "<Put the package license here>"
    author = "<Put your name here> <And your email here>"
    url = "<Package recipe repository url here, for issues about the package>"
    description = "<Description of hello package here>"
    topics = ("<Put some tag here>", "<here>", "<and here>")

    # Binary configuration
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}

    # Sources are located in the same place as this recipe, copy them to the recipe
    exports_sources = "CMakeLists.txt", "src/*", "include/*"

    def config_options(self):
        if self.settings.os == "Windows":
            del self.options.fPIC

    def layout(self):
        cmake_layout(self)

    def generate(self):
        tc = CMakeToolchain(self)
        tc.generate()

    def build(self):
        cmake = CMake(self)
        cmake.configure()
```

(continues on next page)

(continued from previous page)

```
        cmake.build()

    def package(self):
        cmake = CMake(self)
        cmake.install()

    def package_info(self):
        self.cpp_info.libs = ["hello"]
```

Let's explain the different sections of the recipe briefly:

First, you can see the **name and version** of the Conan package defined:

- `name`: a string, with a minimum of 2 and a maximum of 100 **lowercase** characters that defines the package name. It should start with alphanumeric or underscore and can contain alphanumeric, underscore, +, ., - characters.
- `version`: It is a string, and can take any value, matching the same constraints as the `name` attribute. In case the version follows semantic versioning in the form `X.Y.Z-pre1+build2`, that value might be used for requiring this package through version ranges instead of exact versions.

Then you can see, some attributes defining **metadata**. These are optional but recommended and define things like a short `description` for the package, the `author` of the packaged library, the `license`, the `url` for the package repository, and the `topics` that the package is related to.

After that, there is a section related with the binary configuration. This section defines the valid settings and options for the package. As we explained in the *consuming packages section*:

- `settings` are project-wide configuration that cannot be defaulted in recipes. Things like the operating system, compiler or build configuration that will be common to several Conan packages
- `options` are package-specific configuration and can be defaulted in recipes, in this case, we have the option of creating the package as a shared or static library, being static the default.

After that, the `exports_sources` attribute is set to define which sources are part of the Conan package. These are the sources for the library you want to package. In this case the sources for our "hello" library.

Then, several methods are declared:

- The `config_options()` method (together with `configure()` one) allows to fine-tune the binary configuration model, for example, in Windows, there is no `fPIC` option, so it can be removed.
- The `layout()` method declares the locations where we expect to find the source files and also those where we want to save the generated files during the build process. Things like the folder for the generated binaries or all the files that the Conan generators create in the `generate()` method. In this case, as our project uses CMake as the build system, we call to `cmake_layout()`. Calling this function will set the expected locations for a CMake project.
- The `generate()` method prepares the build of the package from source. In this case, it could be simplified to an attribute `generators = "CMakeToolchain"`, but it is left to show this important method. In this case, the execution of `CMakeToolchain` `generate()` method will create a *conan_toolchain.cmake* file that translates the Conan `settings` and `options` to CMake syntax.
- The `build()` method uses the `CMake` wrapper to call CMake commands, it is a thin layer that will manage to pass in this case the `-DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake` argument. It will configure the project and build it from source.
- The `package()` method copies artifacts (headers, libs) from the build folder to the final package folder. It can be done with bare "copy" commands, but in this case, it is leveraging the already existing CMake install

functionality (if the CMakeLists.txt didn't implement it, it is easy to write `self.copy()` commands in this `package()` method.

- Finally, the `package_info()` method defines that consumers must link with a "hello" library when using this package. Other information as include or lib paths can be defined as well. This information is used for files created by generators (as `CMakeDeps`) to be used by consumers. This is generic information about the current package, and is available to the consumers irrespective of the build system they are using and irrespective of the build system we have used in the `build()` method

The **test_package** folder is not critical now for understanding how packages are created. The important bits are:

- **test_package** folder is different from unit or integration tests. These tests are "package" tests, and validate that the package is properly created and that the package consumers will be able to link against it and reuse it.
- It is a small Conan project itself, it contains its `conanfile.py`, and its source code including build scripts, that depends on the package being created, and builds and executes a small application that requires the library in the package.
- It doesn't belong in the package. It only exists in the source repository, not in the package.

Let's build the package from sources with the current default configuration, and then let the `test_package` folder test the package:

```
$ conan create .
-------- Exporting the recipe ----------
hello/1.0: Exporting package recipe
...
[ 50%] Building CXX object CMakeFiles/example.dir/src/example.cpp.o
[100%] Linking CXX executable example
[100%] Built target example

-------- Testing the package: Running test() ----------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release!
  hello/1.0: __x86_64__ defined
  hello/1.0: __cplusplus199711
  hello/1.0: __GNUC__4
  hello/1.0: __GNUC_MINOR__2
  hello/1.0: __clang_major__13
  hello/1.0: __clang_minor__1
  hello/1.0: __apple_build_version__13160021
...
```

If "Hello world Release!" is displayed, it worked. This is what has happened:

- The *conanfile.py* together with the contents of the *src* folder have been copied (**exported**, in Conan terms) to the local Conan cache.
- A new build from source for the `hello/1.0` package starts, calling the `generate()`, `build()` and `package()` methods. This creates the binary package in the Conan cache.
- Conan then moves to the *test_package* folder and executes a **`conan install`** + **`conan build`** + `test()` method, to check if the package was correctly created.

We can now validate that the recipe and the package binary are in the cache:

```
$ conan list recipes hello
Local Cache:
  hello
    hello/1.0
```

The **conan create** command receives the same parameters as **conan install**, so you can pass to it the same settings and options. If we execute the following lines, we will create new package binaries for Debug configuration or to build the hello library as shared:

```
$ conan create . -s build_type=Debug
...
hello/1.0: Hello World Debug!

$ conan create . -o hello/1.0:shared=True
...
hello/1.0: Hello World Release!
```

These new package binaries will be also stored in the Conan cache, ready to be used by any project in this computer, we can see them with:

```
# list the binary built for the hello/1.0 package
# latest is a placeholder to show the package that is the latest created
$ conan list packages hello/1.0#latest
Local Cache:
  hello/1.0#a778356a93c60fe1f687dc2c2ed1449f:46d0e61c1613f12c0e46d007c90dfda85a76a954
    settings:
      arch=x86_64
      build_type=Release
      compiler=apple-clang
      compiler.cppstd=gnu98
      compiler.libcxx=libc++
      compiler.version=13
      os=Macos
    options:
      fPIC=True
      shared=True
  hello/1.0#a778356a93c60fe1f687dc2c2ed1449f:65b76cd1e932112820b979ce174c2c96968f51fb
    settings:
      arch=x86_64
      build_type=Debug
      compiler=apple-clang
      compiler.cppstd=gnu98
      compiler.libcxx=libc++
      compiler.version=13
      os=Macos
    options:
      fPIC=True
      shared=False
  hello/1.0#a778356a93c60fe1f687dc2c2ed1449f:bde82464870a3362a84c3c5d1dd4094fdd4b1bfd
    settings:
      arch=x86_64
      build_type=Release
      compiler=apple-clang
      compiler.cppstd=gnu98
      compiler.libcxx=libc++
      compiler.version=13
      os=Macos
    options:
      fPIC=True
      shared=False
```

Now that we have created a simple Conan package, we will explain each of the methods of the Conanfile in more detail. You will learn how to modify those methods to achieve things like retrieving the sources from an external

repository, adding dependencies to our package, customising our toolchain and much more.

A note about the Conan cache

When you did the **conan create** command, the build of your package did not take place in your local folder but in other folder inside the *Conan cache*. This cache is located in the user home folder under the `.conan2` folder. Conan will use the `~/.conan2` folder to store the built packages and also different configuration files. You already used the **conan list** command to list the recipes and binaries stored in the local cache. There are different subcommands for this command that we will explain in more detail through this tutorial:

- **conan list recipes**: Search available recipes in the local cache or in the remotes.
- **conan list recipe-revisions**: List all the revisions of a recipe reference.
- **conan list packages**: List all the different packages for a given recipe reference.
- **conan list package-revisions**: List all the revisions of a package.

TODO: add note about Conan references, recipe/package references.

Read more

- Create your first Conan package with Autotools.
- Create your first Conan package with Meson.
- Create your first Conan package with Visual Studio.

4.2.2 Handle sources in packages

In the *previous tutorial section* we created a Conan package for a "Hello World" C++ library. We used the `exports_sources` attribute of the Conanfile to declare the location of the sources for the library. This method is the simplest way to define the location of the source files when they are in the same folder as the Conanfile. However, sometimes the source files are stored in a repository or a file in a remote server, and not in the same location as the Conanfile. In this section, we will modify the recipe we created previously by adding a `source()` method and explain how to:

- Retrieve the sources from a *zip* file stored in a remote repository.
- Retrieve the sources from a branch of a *git* repository.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/handle_sources
```

The structure of the project is the same as the one in the previous example but without the library sources:

```
.
├── CMakeLists.txt
├── conanfile.py
└── test_package
    ├── CMakeLists.txt
    ├── conanfile.py
    └── src
        └── example.cpp
```

Sources from a *zip* file stored in a remote repository

Let's have a look at the changes in the *conanfile.py*:

```
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import get


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...

    # Binary configuration
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",
                  strip_root=True)

    def config_options(self):
        if self.settings.os == "Windows":
            del self.options.fPIC

    def layout(self):
        cmake_layout(self)

    def generate(self):
        tc = CMakeToolchain(self)
        tc.generate()

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

    def package(self):
        cmake = CMake(self)
        cmake.install()

    def package_info(self):
        self.cpp_info.libs = ["hello"]
```

As you can see, the recipe is the same but instead declaring the `exports_sources` attribute as we did previously:

```
exports_sources = "CMakeLists.txt", "src/*", "include/*"
```

We declare a `source()` method with this information:

```
def source(self):
    get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",
              strip_root=True)
```

We used the *conan.tools.files.get()* tool that will first **download** the *zip* file from the URL that we pass as an argument and then **unzip** it. Note that we pass the `strip_root=True` argument so that if all the unzipped contents are in a single folder, all the contents are moved to the parent folder (check the *conan.tools.files.unzip()* reference for more details).

The contents of the zip file are the same as the sources we previously had beside the Conan recipe, so if you do a **conan create** the results will be the same as before.

```
$ conan create .

...

-------- Installing packages ----------

Installing (downloading, building) binaries...
hello/1.0: Calling source() in /Users/carlosz/.conan2/p/0fcb5ffd11025446/s/.
Downloading update_source.zip

hello/1.0: Unzipping 3.7KB
Unzipping 100 %
hello/1.0: Copying sources to build folder
hello/1.0: Building your package in /Users/carlosz/.conan2/p/tmp/369786d0fb355069/b

...

-------- Testing the package: Running test() ----------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release!
hello/1.0: __x86_64__ defined
hello/1.0: __cplusplus199711
hello/1.0: __GNUC__4
hello/1.0: __GNUC_MINOR__2
hello/1.0: __clang_major__13
hello/1.0: __clang_minor__1
hello/1.0: __apple_build_version__13160021
```

Please, check the highlighted lines with the messages about the download and unzip operation.

Sources from a branch in a *git* repository

Now, let's modify the `source()` method to bring the sources from a *git* repository instead of a *zip* file. We show just the relevant parts:

```
...

from conan.tools.scm import Git


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...

    def source(self):
        git = Git(self)
        git.clone(url="https://github.com/conan-io/libhello.git", target=".")

    ...
```

Here, we use the *conan.tools.scm.Git()* tool. The `Git` class implements several methods to work with *git* repositories. In this case, we call the clone method to clone the https://github.com/conan-io/libhello.git repository in the default

branch using the same folder for cloning the sources instead of a subfolder (passing the `target="."` argument).

If we wanted to checkout a commit or tag in the repository we could use the `checkout()` method of the Git tool:

```
def source(self):
    git = Git(self)
    git.clone(url="https://github.com/conan-io/libhello.git", target=".")
    git.checkout("<branch name>, <tag> or <commit hash>")
```

For more information about the `Git` class methods, please check the *conan.tools.scm.Git()* reference.

Using the conandata.yml file

We can write a file named `conandata.yml` in the same folder of the `conanfile.py`. This file will be automatically exported and parsed by Conan and we can read that information from the recipe. This is handy for example to extract the URLs of the external sources repositories, zip files etc. This is an example of `conandata.yml`:

```
sources:
  "1.0":
    url: "https://github.com/conan-io/libhello/archive/refs/heads/main.zip"
    sha256: "7bc71c682895758a996ccf33b70b91611f51252832b01ef3b4675371510ee466"
    strip_root: true
  "1.1":
    url: ...
    sha256: ...
```

The recipe doesn't need to be modified for each version of the code. We can pass all the `keys` of the specified version (`url`, `sha256`, and `strip_root`) as arguments to the `get` function, that, in this case, allow us to verify that the downloaded zip file has the correct `sha256`. So we could modify the source method to this:

```
def source(self):
    get(self, **self.conan_data["sources"][self.version])
    # Similar to:
    # data = self.conan_data["sources"][self.version]
    # get(self, data["url"], sha256=data["sha256"], strip_root=data["strip_root"])
```

Read more

- *Patching sources*
- Advanced git repository handling (implement the "scm feature")
- ...

4.2.3 Add dependencies to packages

Important: In this example, we will retrieve the *fmt* Conan package from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

In the *previous tutorial section* we created a Conan package for a "Hello World" C++ library. We used the *conan.tools.scm.Git()* tool to retrieve the sources from a git repository. So far, the package does not have any dependency on other Conan packages. Let's explain how to add a dependency to our package in a very similar way that we did in the *consuming packages section*. We will add some fancy colour output to our "Hello World" library using the fmt library.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/add_requires
```

You will notice some changes in the *conanfile.py* file from the previous recipe. Let's check the relevant parts:

```
...
from conan.tools.build import check_max_cppstd, check_min_cppstd
...

class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...
    generators = "CMakeDeps"
    ...

    def validate(self):
        check_min_cppstd(self, "11")
        check_max_cppstd(self, "14")

    def requirements(self):
        self.requires("fmt/8.1.1")

    def source(self):
        git = Git(self)
        git.clone(url="https://github.com/conan-io/libhello.git", target=".")
        # Please, be aware that using the head of the branch instead of an inmutable␣
↪tag
        # or commit is not a good practice in general
        git.checkout("require_fmt")
```

- First, we set the `generators` class attribute to make Conan invoke the *CMakeDeps* generator. This was not needed in the previous recipe as we did not have dependencies. `CMakeDeps` will generate all the config files CMake needs to find the `fmt` library.
- Next, we use the *requires()* method to add the fmt dependency to our package.
- Also, check that we added an extra line in the *source()* method. We use the *Git().checkout* method to checkout the source code in the require_fmt branch. This branch contains the changes in the source code to add colours to the library messages, and also in the `CMakeLists.txt` to declare that we are using the `fmt` library.
- Finally, note we added the *validate()* method to the recipe. We already used this method in the *consuming packages section* to raise an error for non-supported configurations. Here, we call the *check_min_cppstd()* and *check_max_cppstd()* to check that we are using at least C++11 and at most C++14 standards in our settings.

You can check the new sources, using the fmt library in the require_fmt. You will see that the hello.cpp file adds colours to the output messages:

```
#include <fmt/color.h>

#include "hello.h"

void hello(){
    #ifdef NDEBUG
    fmt::print(fg(fmt::color::crimson) | fmt::emphasis::bold, "hello/1.0: Hello World␣
↪Release!\n");
```

(continues on next page)

(continued from previous page)

```
    #else
    fmt::print(fg(fmt::color::crimson) | fmt::emphasis::bold, "hello/1.0: Hello World␣
↪Debug!\n");
    #endif
    ...
```

Let's build the package from sources with the current default configuration, and then let the `test_package` folder test the package. You should see the output messages with colour now:

```
$ conan create . --build=missing -s compiler.cppstd=gnu11
-------- Exporting the recipe ----------
...
-------- Testing the package: Running test() ----------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release!
  hello/1.0: __x86_64__ defined
  hello/1.0: __cplusplus 201103
  hello/1.0: __GNUC__ 4
  hello/1.0: __GNUC_MINOR__ 2
  hello/1.0: __clang_major__ 13
  hello/1.0: __clang_minor__ 1
  hello/1.0: __apple_build_version__ 13160021
```

Note that we passed the `-s compiler.cppstd=11` argument to the **`conan create`** command to override the C++ standard used in the default profile. You can set the C++ standard of your choice or leave the default one as long as it is higher than C++11.

Read more

- Version ranges
- Requirement traits
- ...

4.2.4 Preparing the build

Important: In this example, we retrieve the *fmt* Conan package from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

In the *previous tutorial section*, we added the fmt requirement to our Conan package to provide colour output to our "Hello World" C++ library. In this section, we focus on the `generate()` method of the recipe. The aim of this method generating all the information that could be needed while running the build step. That means things like:

- Write files to be used in the build step, like *scripts* that inject environment variables, files to pass to the build system, etc.
- Configuring the toolchain to provide extra information based on the settings and options or removing information from the toolchain that Conan generates by default and may not apply for certain cases.

We explain to use this method for a simple example based on the previous tutorial section. We add a *with_fmt* option to the recipe, depending on the value we require the *fmt* library or not. We use the *generate()* method to modify the

toolchain so that it passes a variable to CMake so that we can conditionally add that library and use *fmt* or not in the source code.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/preparing_the_build
```

You will notice some changes in the *conanfile.py* file from the previous recipe. Let's check the relevant parts:

```
...
from conan.tools.build import check_max_cppstd, check_min_cppstd
...

class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...
    options = {"shared": [True, False],
               "fPIC": [True, False],
               "with_fmt": [True, False]}

    default_options = {"shared": False,
                       "fPIC": True,
                       "with_fmt": True}
    ...

    def validate(self):
        if self.info.options.with_fmt:
            check_min_cppstd(self, "11")
            check_max_cppstd(self, "14")

    def source(self):
        git = Git(self)
        git.clone(url="https://github.com/conan-io/libhello.git", target=".")
        # Please, be aware that using the head of the branch instead of an inmutable␣
↪tag
        # or commit is not a good practice in general
        git.checkout("optional_fmt")

    def requirements(self):
        if self.options.with_fmt:
            self.requires("fmt/8.1.1")

    def generate(self):
        tc = CMakeToolchain(self)
        if self.options.with_fmt:
            tc.variables["WITH_FMT"] = True
        tc.generate()

    ...
```

As you can see:

- We declare a new `with_fmt` option with the default value set to `True`
- Based on the value of the `with_fmt` option:
  - We install or not the `fmt/8.1.1` Conan package.

  - We require or not a minimum and a maximum C++ standard as the *fmt* library requires at least C++11 and it will not compile if we try to use a standard above C++14 (just an example, *fmt* can build with more modern standards)
  - We inject the `WITH_FMT` variable with the value `True` to the *CMakeToolchain* so that we can use it in the *CMakeLists.txt* of the **hello** library to add the CMake **fmt::fmt** target conditionally.

- We are cloning another branch of the library. The *optional_fmt* branch contains some changes in the code. Let's see what changed on the CMake side:

Listing 42: **CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.15)
project(hello CXX)

add_library(hello src/hello.cpp)
target_include_directories(hello PUBLIC include)
set_target_properties(hello PROPERTIES PUBLIC_HEADER "include/hello.h")

if (WITH_FMT)
    find_package(fmt)
    target_link_libraries(hello fmt::fmt)
    target_compile_definitions(hello PRIVATE USING_FMT=1)
endif()

install(TARGETS hello)
```

As you can see, we use the `WITH_FMT` we injected in the *CMakeToolchain*. Depending on the value we will try to find the fmt library and link our hello library with it. Also, check that we add the `USING_FMT=1` compile definition that we use in the source code depending on whether we choose to add support for `fmt` or not.

Listing 43: **hello.cpp**

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

#if USING_FMT == 1
#include <fmt/color.h>
#endif

void hello(){
    #if USING_FMT == 1
        #ifdef NDEBUG
        fmt::print(fg(fmt::color::crimson) | fmt::emphasis::bold, "hello/1.0: Hello␣
↪World Release! (with color!)\n");
        #else
        fmt::print(fg(fmt::color::crimson) | fmt::emphasis::bold, "hello/1.0: Hello␣
↪World Debug! (with color!)\n");
        #endif
    #else
        #ifdef NDEBUG
        std::cout << "hello/1.0: Hello World Release! (without color)" << std::endl;
        #else
        std::cout << "hello/1.0: Hello World Debug! (without color)" << std::endl;
        #endif
    #endif
}
```

Let's build the package from sources first using `with_fmt=True` and then `with_fmt=False`. When *test_package*

runs it will show different messages depending on the value of the option.

```
$ conan create . --build=missing -s compiler.cppstd=gnu1 -o with_fmt=True
-------- Exporting the recipe ----------
...

-------- Testing the package: Running test() ----------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release! (with color!)

$ conan create . --build=missing -o with_fmt=False
-------- Exporting the recipe ----------
...

-------- Testing the package: Running test() ----------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release! (without color)
```

This is just a simple example of how to use the `generate()` method to customize the toolchain based on the value of one option, but there are lots of other things that you could do in the `generate()` method like:

- Create a complete custom toolchain based on your needs to use in your build.
- **Access to certain information about the package dependencies, like:**
  - The configuration accessing the defined *conf_info*.
  - Accessing the dependencies options.
  - Import files from dependencies using the *copy tool*. You could also import the files create manifests for the package, collecting all dependencies versions and licenses.
- Use the *Environment tools* to generate information for the system environment.
- Adding custom configurations besides *Release* and *Debug*, taking into account the settings, like *ReleaseShared* or *DebugShared*.

Read more

- Use the `generate()` method to import files from dependencies.
- More based on the examples mentioned above ...

4.2.5 Configure settings and options in recipes

Important: In this example, we retrieve the *fmt* Conan package from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

We already explained *Conan settings and options* and how to use them to build your projects for different configurations like Debug, Release, with static or shared libraries, etc. In this section, we explain how to configure these settings and options in the case that one of them does not apply to a Conan package. We will introduce briefly how Conan models binary compatibility and how that relates to options and settings.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/configure_options_settings
```

You will notice some changes in the **conanfile.py** file from the previous recipe. Let's check the relevant parts:

```
class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...
    options = {"shared": [True, False],
               "fPIC": [True, False],
               "with_fmt": [True, False]}

    default_options = {"shared": False,
                       "fPIC": True,
                       "with_fmt": True}
    ...

    def config_options(self):
        if self.settings.os == "Windows":
            del self.options.fPIC

    def configure(self):
        if self.options.shared:
            del self.options.fPIC
    ...
```

You can see that we added a `configure()` method to the recipe. Let's explain what's the objective of this method and how it's different from the `config_options()` method we already had defined in the recipe:

- `configure()`: use this method to configure which options or settings of the recipe are available. For example, in this case, we **delete the fPIC option**, because it should only be **True** if we are building the library as shared (in fact, some build systems will add this flag automatically when building a shared library).
- `config_options()`: This method is used to **constraint** the available options in a package **before they take a value**. If a value is assigned to a setting or option that is deleted inside this method, Conan will raise an error. In this case we are **deleting the fPIC option** in Windows because that option does not exist for that operating system. Note that this method is executed before the `configure()` method.

Be aware that deleting an option in the `config_options()` or in the `configure()` has not the same result. Deleting it in the `config_options()` **is like if we never declared it in the recipe** and it will raise an exception saying that the option does not exist. Nevertheless, if we delete it in the `configure()` method we can pass the option but it will have no effect. For example, if you try to pass a value to the `fPIC` option in Windows, Conan will raise an error warning that the option does not exist:

Listing 44: Windows

```
$ conan create . --build=missing -o fPIC=True
...
-------- Computing dependency graph --------
ERROR: option 'fPIC' doesn't exist
Possible options are ['shared', 'with_fmt']
```

As you have noticed, the `configure()` and `config_options()` methods **delete an option** if certain conditions meet. Let's explain why we are doing this and the implications of removing that option. It is related to how Conan identifies packages that are binary compatible with the configuration set in the profile. In the next section, we introduce the concept of the **Conan package ID**.

Conan packages binary compatibility: the package ID

We used Conan in previous examples to build for different configurations like *Debug* and *Release*. Each time you create the package for one of those configurations, Conan will build a new binary. Each of them is related to a **generated hash** called **the package ID**. The package ID is just a way to convert a set of settings, options and information about the requirements of the package to a unique identifier.

Let's build our package for *Release* and *Debug* configurations and check the generated binaries package IDs.

```
$ conan create . --build=missing -s compiler.cppstd=gnu11 -s build_type=Release -
↪tf=None # -tf=None will skip buildiing the test_package
...
[ 50%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[100%] Linking CXX static library libhello.a
[100%] Built target hello
hello/1.0: Package '738feca714b7251063cc51448da0cf4811424e7c' built
hello/1.0: Build folder /Users/user/.conan2/p/tmp/7fe7f5af0ef27552/b/build/Release
hello/1.0: Generated conaninfo.txt
hello/1.0: Generating the package
hello/1.0: Temporary package folder /Users/user/.conan2/p/tmp/7fe7f5af0ef27552/p
hello/1.0: Calling package()
hello/1.0: CMake command: cmake --install "/Users/user/.conan2/p/tmp/7fe7f5af0ef27552/
↪b/build/Release" --prefix "/Users/user/.conan2/p/tmp/7fe7f5af0ef27552/p"
hello/1.0: RUN: cmake --install "/Users/user/.conan2/p/tmp/7fe7f5af0ef27552/b/build/
↪Release" --prefix "/Users/user/.conan2/p/tmp/7fe7f5af0ef27552/p"
-- Install configuration: "Release"
-- Installing: /Users/user/.conan2/p/tmp/7fe7f5af0ef27552/p/lib/libhello.a
-- Installing: /Users/user/.conan2/p/tmp/7fe7f5af0ef27552/p/include/hello.h
hello/1.0 package(): Packaged 1 '.h' file: hello.h
hello/1.0 package(): Packaged 1 '.a' file: libhello.a
hello/1.0: Package '738feca714b7251063cc51448da0cf4811424e7c' created
hello/1.0: Created package revision 3bd9faedc711cbb4fdf10b295268246e
hello/1.0: Full package reference: hello/1.0
↪#e6b11fb0cb64e3777f8d62f4543cd6b3:738feca714b7251063cc51448da0cf4811424e7c
↪#3bd9faedc711cbb4fdf10b295268246e
hello/1.0: Package folder /Users/user/.conan2/p/5c497cbb5421cbda/p

$ conan create . --build=missing -s compiler.cppstd=gnu11 -s build_type=Debug -
↪tf=None # -tf=None will skip buildiing the test_package
...
[ 50%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[100%] Linking CXX static library libhello.a
[100%] Built target hello
hello/1.0: Package '3d27635e4dd04a258d180fe03cfa07ae1186a828' built
hello/1.0: Build folder /Users/user/.conan2/p/tmp/19a2e552db727a2b/b/build/Debug
hello/1.0: Generated conaninfo.txt
hello/1.0: Generating the package
hello/1.0: Temporary package folder /Users/user/.conan2/p/tmp/19a2e552db727a2b/p
hello/1.0: Calling package()
hello/1.0: CMake command: cmake --install "/Users/user/.conan2/p/tmp/19a2e552db727a2b/
↪b/build/Debug" --prefix "/Users/user/.conan2/p/tmp/19a2e552db727a2b/p"
hello/1.0: RUN: cmake --install "/Users/user/.conan2/p/tmp/19a2e552db727a2b/b/build/
↪Debug" --prefix "/Users/user/.conan2/p/tmp/19a2e552db727a2b/p"
-- Install configuration: "Debug"
-- Installing: /Users/user/.conan2/p/tmp/19a2e552db727a2b/p/lib/libhello.a
-- Installing: /Users/user/.conan2/p/tmp/19a2e552db727a2b/p/include/hello.h
hello/1.0 package(): Packaged 1 '.h' file: hello.h
hello/1.0 package(): Packaged 1 '.a' file: libhello.a
```

(continues on next page)

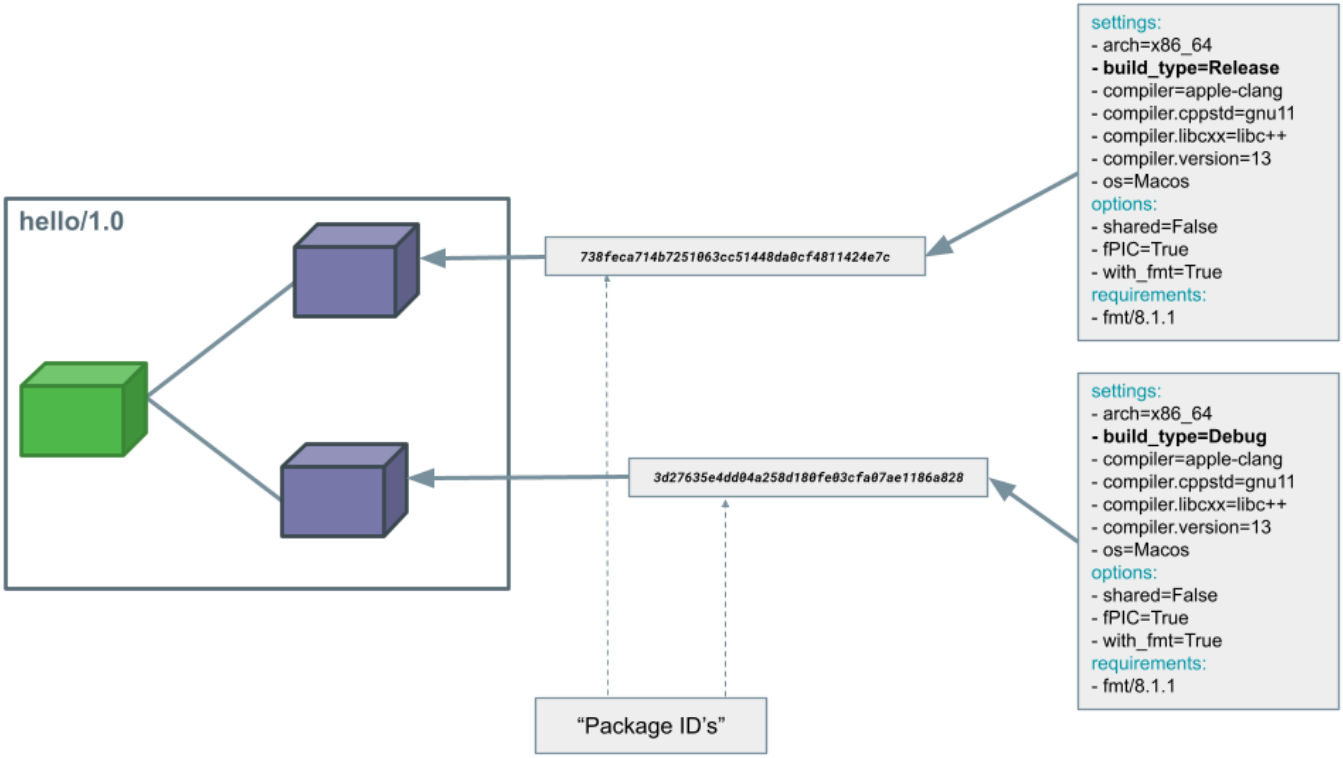
(continued from previous page)

```
hello/1.0: Package '3d27635e4dd04a258d180fe03cfa07ae1186a828' created
hello/1.0: Created package revision 67b887a0805c2a535b58be404529c1fe
hello/1.0: Full package reference: hello/1.0
↪#e6b11fb0cb64e3777f8d62f4543cd6b3:3d27635e4dd04a258d180fe03cfa07ae1186a828
↪#67b887a0805c2a535b58be404529c1fe
hello/1.0: Package folder /Users/user/.conan2/p/c7796386fcad5369/p
```

As you can see Conan generated two package IDs:

- Package *738feca714b7251063cc51448da0cf4811424e7c* for Release
- Package *3d27635e4dd04a258d180fe03cfa07ae1186a828* for Debug

These two package IDs are calculated by taking the **set of settings, options and some information about the requirements** (we will explain this later in the documentation) and **calculating a hash** with them. So, for example, in this case, they are the result of the information depicted in the diagram below.



Those package IDs are different because the **build_type** is different. Now, when you want to install a package, Conan will:

- Collect the settings and options applied, along with some information about the requirements and calculate the hash for the corresponding package ID.
- If that package ID matches one of the packages stored in the local Conan cache Conan will use that. If not, and we have any Conan remote configured, it will search for a package with that package ID in the remotes.
- If that calculated package ID does not exist in the local cache and remotes, Conan will fail with a "missing binary" error message, or will try to build that package from sources (this depends on the value of the `--build` argument). This build will generate a new package ID in the local cache.

This steps are simplified, there is far more to package ID calculation than what we explain here, recipes themselves can even adjust their package ID calculations, we can have different recipe and package revisions besides package IDs

and there's also a built-in mechanism in Conan that can be configured to declare that some packages with a certain package ID are compatible with other.

Maybe you have now the intuition of why we delete settings or options in Conan recipes. If you do that, those values will not be added to the computation of the package ID, so even if you define them, the resulting package ID will be the same. You can check this behaviour, for example with the fPIC option that is deleted when we build with the option `shared=True`. Regardless of the value you pass for the fPIC option the generated package ID will be the same for the **hello/1.0** binary:

```
$ conan conan create . --build=missing -s compiler.cppstd=gnu11 -o shared=True -o␣
↪fPIC=True -tf=None
...
hello/1.0 package(): Packaged 1 '.h' file: hello.h
hello/1.0 package(): Packaged 1 '.dylib' file: libhello.dylib
hello/1.0: Package '2a899fd0da3125064bf9328b8db681cd82899d56' created
hello/1.0: Created package revision f0d1385f4f90ae465341c15740552d7e
hello/1.0: Full package reference: hello/1.0
↪#e6b11fb0cb64e3777f8d62f4543cd6b3:2a899fd0da3125064bf9328b8db681cd82899d56
↪#f0d1385f4f90ae465341c15740552d7e
hello/1.0: Package folder /Users/user/.conan2/p/8a55286c6595f662/p

$ conan conan create . --build=missing -s compiler.cppstd=gnu11 -o shared=True -o␣
↪fPIC=False -tf=None
...
-------- Computing dependency graph --------
Graph root
    virtual
Requirements
    fmt/8.1.1#601209640bd378c906638a8de90070f7 - Cache
    hello/1.0#e6b11fb0cb64e3777f8d62f4543cd6b3 - Cache

-------- Computing necessary packages --------
Requirements
    fmt/8.1.1
↪#601209640bd378c906638a8de90070f7:d1b3f3666400710fec06446a697f9eeddd1235aa
↪#24a2edf207deeed4151bd87bca4af51c - Skip
    hello/1.0
↪#e6b11fb0cb64e3777f8d62f4543cd6b3:2a899fd0da3125064bf9328b8db681cd82899d56
↪#f0d1385f4f90ae465341c15740552d7e - Cache

-------- Installing packages --------

-------- Installing (downloading, building) binaries... --------
hello/1.0: Already installed!
```

As you can see, the first run created the `2a899fd0da3125064bf9328b8db681cd82899d56` package, and the second one, regardless of the different value of the fPIC option, said we already had the `2a899fd0da3125064bf9328b8db681cd82899d56` package installed.

C libraries

There are other typical cases where you want to delete certain settings. Imagine that you are packaging a C library. When you build this library, there are settings like the compiler C++ standard (`settings.compiler.cppstd`) or the standard library used (`self.settings.compiler.libcxx`) that won't affect the resulting binary at all. Then it does no make sense that they affect to the package ID computation, so a typical pattern is to delete them in the `configure()` method:

```
def configure(self):
    del self.settings.compiler.cppstd
    del self.settings.compiler.libcxx
```

Please, note that deleting these settings in the `configure()` method will modify the package ID calculation but will also affect how the toolchain, and the build system integrations work because the C++ settings do not exist.

Header-only libraries

A similar case happens with packages that package *header-only libraries*. In that case, there's no binary code we need to link with, but just some header files to add to our project. In this cases the package ID of the Conan package should not be affected by settings or options. For that case, there's a simplified way of declaring that the generated package ID should not take into account settings, options or any information from the requirement which is using the `self.info.clear()` method inside another recipe method called `package_id()`:

```
def package_id(self):
  self.info.clear()
```

We will explain the `package_id()` method later and explain how you can customize the way the package ID for the package is calculated. You can also check the *Conanfile's methods reference* if you want to know how this method works in more detail.

Read more

- *Header-only packages*
- compatibililty.py
- package types
- package id modes
- ...

4.2.6 Build packages

4.2.7 Install packages

4.2.8 Define the package information

4.2.9 Test Conan packages

4.2.10 Other types of packages

In the previous sections, we saw how to create a new recipe for a classic C++ library but there are other types of packages rather than libraries.

In this section, we are going to review how to create a recipe for `header-only` libraries, how to package already `built libraries`, and how to create recipes for `tool requires` an `applications`.

Header-only packages

In this section, we are going to learn how to create a recipe for a header-only library.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/other_packages/header_only
```

A header-only library is composed only of header files. That means a consumer doesn't link with any library but includes headers, so we need only one binary configuration for a header-only library.

In the *Create your first Conan package* section, we learned about the settings, and how building the recipe applying different `build_type` (Release/Debug) generates a new binary package.

As we only need one binary package, we don't need to declare the *settings* attribute. This is a basic recipe for a header-only recipe:

Listing 45: conanfile.py

```
from conan import ConanFile
from conan.tools.files import copy


class SumConan(ConanFile):
    name = "sum"
    version = "0.1"
    # No settings/options are necessary, this is header only
    exports_sources = "include/*"
    # We can avoid copying the sources to the build folder in the cache
    no_copy_source = True

    def package(self):
        # This will also copy the "include" folder
        copy(self, "*.h", self.source_folder, self.package_folder)
```

Our header-only library is this simple function that sums two numbers:

Listing 46: include/sum.h

```
inline int sum(int a, int b){
    return a + b;
}
```

The folder *examples2/tutorial/creating_packages/other_packages/header_only* in the cloned project contains a `test_package` folder with an example of an application consuming the header-only library. So we can run a `conan create .` command to build the package and test the package:

```
$ conan create .
...
[ 50%] Building CXX object CMakeFiles/example.dir/src/example.cpp.o
[100%] Linking CXX executable example
[100%] Built target example

-------- Testing the package: Running test() ----------
sum/0.1 (test package): Running test()
sum/0.1 (test package): RUN: ./example
1 + 3 = 4
```

After running the `conan create` a new binary package is created for the header-only library, and we can see how the `test_package` project can use it correctly.

We can list the binary packages created running this command:

```
$ conan list packages sum/0.1#latest
Local Cache:
    sum/0.1#154c10cda76635e3ca85bcd7e21c2de9:da39a3ee5e6b4b0d3255bfef95601890afd80709
```

We get one package with the package ID `da39a3ee5e6b4b0d3255bfef95601890afd80709`. Let's see what happen if we run the `conan create` but specifying `-s build_type=Debug`:

```
$ conan create . -s build_type=Debug
$ conan list packages sum/0.1#latest
Local Cache:
    sum/0.1#154c10cda76635e3ca85bcd7e21c2de9:da39a3ee5e6b4b0d3255bfef95601890afd80709
```

Even in the `test_package` executable is built for Debug, we get the same binary package for the header-only library. This is because we didn't specify the `settings` attribute in the recipe, so the changes in the input settings (`-s build_type=Debug`) do not affect the recipe and therefore the generated binary package is always the same.

Header-only library with tests

Important: In this example, we will retrieve the CMake Conan package from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

In the previous example, we saw why a recipe header-only library shouldn't declare the `settings` attribute, but sometimes the recipe needs them to build some executable, for example, for testing the library. Nonetheless, the binary package of the header-only library should still be unique, so we are going to review how to achieve that.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/other_packages/header_only_gtest
```

We have the same header-only library that sums two numbers, but now we have this recipe:

```
import os
from conan import ConanFile
from conan.tools.files import copy
from conan.tools.cmake import cmake_layout, CMake


class SumConan(ConanFile):
    name = "sum"
    version = "0.1"
    settings = "os", "arch", "compiler", "build_type"
    exports_sources = "include/*", "test/*"
    no_copy_source = True
    generators = "CMakeToolchain", "CMakeDeps"

    def requirements(self):
        self.test_requires("gtest/1.11.0")

    def validate(self):
        check_min_cppstd(self, 11)

    def layout(self):
        cmake_layout(self)

    def build(self):
        if not self.conf.get("tools.build:skip_test", default=False):
```

(continues on next page)

(continued from previous page)

```
            cmake = CMake(self)
            cmake.configure(build_script_folder="test")
            cmake.build()
            self.run(os.path.join(self.cpp.build.bindirs[0], "test_sum"))

    def package(self):
        # This will also copy the "include" folder
        copy(self, "*.h", self.source_folder, self.package_folder)

    def package_id(self):
        self.info.clear()
```

These are the changes introduced in the recipe:

- We are introducing a `test_require` to `gtest/1.11.0`. A `test_require` is similar to a regular requirement but it is not propagated to the consumers and cannot conflict.
- `gtest` needs at least C++11 to build. So we introduced a `validate()` method calling `check_min_cppstd`.
- As we are building the `gtest` examples with CMake, we use the generators `CMakeToolchain` and `CMakeDeps`, and we declared the `cmake_layout()` to have a known/standard directory structure.
- We have a `build()` method, building the tests, but only when the standard conf `tools.build:skip_test` is not True. Use that conf as a standard way to enable/disable the testing. It is used by the helpers like `CMake` to skip the `cmake.test()` in case we implement the tests in CMake.
- We have a `package_id()` method calling `self.info.clear()`. This is internally removing the settings from the package ID calculation so we generate only one configuration for our header-only library.

We can call `conan create` to build and test our package.

```
$ conan create . -s compiler.cppstd=14 --build missing
...
Running main() from /Users/luism/.conan2/p/tmp/9bf83ef65d5ff0d6/b/googletest/
↪src/gtest_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from SumTest
[ RUN      ] SumTest.BasicSum
[       OK ] SumTest.BasicSum (0 ms)
[----------] 1 test from SumTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
sum/0.1: Package 'da39a3ee5e6b4b0d3255bfef95601890afd80709' built
...
```

We can run `conan create` again specifying a different `compiler.cppstd` and the built package would be the same:

```
$ conan create . -s compiler.cppstd=17
...
sum/0.1: RUN: ./test_sum
Running main() from /Users/luism/.conan2/p/tmp/9bf83ef65d5ff0d6/b/googletest/
↪src/gtest_main.cc
[==========] Running 1 test from 1 test suite.
```

(continues on next page)

(continued from previous page)

```
[----------] Global test environment set-up.
[----------] 1 test from SumTest
[ RUN      ] SumTest.BasicSum
[       OK ] SumTest.BasicSum (0 ms)
[----------] 1 test from SumTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
sum/0.1: Package 'da39a3ee5e6b4b0d3255bfef95601890afd80709' built
```

Note: Once we have the `sum/0.1` binary package available (in a server, after a `conan upload`, or in the local cache), we can install it even if we don't specify input values for `os`, `arch`, ... etc. This is a new feature of Conan 2.X.

We could call `conan install --require sum/0.1` with an empty profile and would get the binary package from the server. But if we miss the binary and we need to build the package again, it will fail because of the lack of settings.

Package prebuilt binaries

There are specific scenarios in which it is necessary to create packages from existing binaries, for example from 3rd parties or binaries previously built by another process or team that is not using Conan. Under these circumstances, building from sources is not what you want.

You can package the local files in the following scenarios:

1. When you are developing your package locally and you want to quickly create a package with the built artifacts, but as you don't want to rebuild again (clean copy) your artifacts, you don't want to call **`conan create`**. This method will keep your local project build if you are using an IDE.
2. When you cannot build the packages from sources (when only pre-built binaries are available) and you have them in a local directory.
3. Same as 2 but you have the precompiled libraries in a remote repository.

Locally building binaries

Use the **`conan new`** command to create a "Hello World" C++ library example project:

```
$ conan new cmake_lib -d name=hello -d version=1.0
```

This will create a Conan package project with the following structure.

```
.
├── CMakeLists.txt
├── conanfile.py
├── include
│   └── hello.h
├── src
│   └── hello.cpp
└── test_package
    ├── CMakeLists.txt
    ├── conanfile.py
```

(continues on next page)

(continued from previous page)

```
    └── src
        └── example.cpp
```

We have a `CMakeLists.txt` file in the root, an `src` folder with the `cpp` files and, an `include` folder for the headers.

They also have a `test_package/` folder to test that the exported package is working correctly.

Now, for every different configuration (different compilers, architectures, build_type...):

1. We call **`conan install`** to generate the `conan_toolchain.cmake` file and the `CMakeUserPresets.json` that can be used in our IDE or calling CMake (only >= 3.23).

```
$ conan install . -s build_type=Release
```

2. We build our project calling CMake, our IDE, ... etc:

Listing 47: Linux, macOS

```
$ mkdir -p build/Release
$ cd build/Release
$ cmake ../.. -DCMAKE_BUILD_TYPE=Release -DCMAKE_TOOLCHAIN_FILE=../generators/
↪conan_toolchain.cmake
$ cmake --build .
```

Listing 48: Windows

```
$ mkdir -p build
$ cd build
$ cmake ..  -DCMAKE_TOOLCHAIN_FILE=generators/conan_toolchain.cmake
$ cmake --build . --config Release
```

Note: As we are directly using our IDE or CMake to build the library, the `build()` method of the recipe is never called and could be removed.

3. We call **`conan export-pkg`** to package the built artifacts.

```
$ conan export-pkg . -s build_type=Release
...
hello/0.1: Calling package()
hello/0.1 package(): Packaged 1 '.h' file: hello.h
hello/0.1 package(): Packaged 1 '.a' file: libhello.a
...
hello/0.1: Package '54a3ab9b777a90a13e500dd311d9cd70316e9d55' created
```

Let's deep a bit more in the package method. The generated `package()` method is using `cmake.install()` to copy the artifacts from our local folders to the Conan package.

There is an alternative and generic `package()` method that could be used for any build system:

```
def package(self):
    local_include_folder = os.path.join(self.source_folder, self.cpp.source.
↪includedirs[0])
    local_lib_folder = os.path.join(self.build_folder, self.cpp.build.libdirs[0])
    copy(self, "*.h", local_include_folder, os.path.join(self.package_folder,
↪"include"), keep_path=False)
```

(continues on next page)

(continued from previous page)

```
    copy(self, "*.lib", local_lib_folder, os.path.join(self.package_folder, "lib
↪"), keep_path=False)
    copy(self, "*.a", local_lib_folder, os.path.join(self.package_folder, "lib"),␣
↪keep_path=False)
```

This `package()` method is copying artifacts from the following directories that, thanks to the layout(), will always point to the correct places:

- **os.path.join(self.source_folder, self.cpp.source.includedirs[0])** will always point to our local include folder.
- **os.path.join(self.build_folder, self.cpp.build.libdirs[0])** will always point to the location of the libraries when they are built, no matter if using a single-config CMake Generator or a multi-config one.

4. We can test the built package calling **`conan test`**:

```
$ conan test test_package/conanfile.py hello/0.1 -s build_type=Release

-------- Testing the package: Running test() ----------
hello/0.1 (test package): Running test()
hello/0.1 (test package): RUN: ./example
hello/0.1: Hello World Release!
  hello/0.1: __x86_64__ defined
  hello/0.1: __cplusplus199711
  hello/0.1: __GNUC__4
  hello/0.1: __GNUC_MINOR__2
  hello/0.1: __clang_major__13
  hello/0.1: __clang_minor__1
  hello/0.1: __apple_build_version__13160021
```

Now you can try to generate a binary package for `build_type=Debug` running the same steps but changing the `build_type`. You can repeat this process any number of times for different configurations.

Packaging already Pre-built Binaries

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/other_packages/prebuilt_binaries
```

This is an example of scenario 2 explained in the introduction. If you have a local folder containing the binaries for different configurations you can package them using the following approach.

These are the files of our example, (be aware that the library files are only empty files so not valid libraries):

```
.
├── conanfile.py
└── vendor_hello_library
    ├── linux
    │   ├── armv8
    │   │   ├── include
    │   │   │   └── hello.h
    │   │   └── libhello.a
    │   └── x86_64
    │       ├── include
    │       │   └── hello.h
    │       └── libhello.a
```

(continues on next page)

(continued from previous page)

```
    ├── macos
    │   ├── armv8
    │   │   ├── include
    │   │   │   └── hello.h
    │   │   └── libhello.a
    │   └── x86_64
    │       ├── include
    │       │   └── hello.h
    │       └── libhello.a
    └── windows
        ├── armv8
        │   ├── hello.lib
        │   └── include
        │       └── hello.h
        └── x86_64
            ├── hello.lib
            └── include
                └── hello.h
```

We have folders with `os` and subfolders with `arch`. This the recipe of our example:

```
import os
from conan import ConanFile
from conan.tools.files import copy


class helloRecipe(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "arch"

    def layout(self):
        _os = str(self.settings.os).lower()
        _arch = str(self.settings.arch).lower()
        self.folders.build = os.path.join("vendor_hello_library", _os, _arch)
        self.folders.source = self.folders.build
        self.cpp.source.includedirs = ["include"]
        self.cpp.build.libdirs = ["."]

    def package(self):
        local_include_folder = os.path.join(self.source_folder, self.cpp.source.
↪includedirs[0])
        local_lib_folder = os.path.join(self.build_folder, self.cpp.build.libdirs[0])
        copy(self, "*.h", local_include_folder, os.path.join(self.package_folder,
↪"include"), keep_path=False)
        copy(self, "*.lib", local_lib_folder, os.path.join(self.package_folder, "lib
↪"), keep_path=False)
        copy(self, "*.a", local_lib_folder, os.path.join(self.package_folder, "lib"),␣
↪keep_path=False)

    def package_info(self):
        self.cpp_info.libs = ["hello"]
```

- We are not building anything, so the `build` method is not useful here.
- We can keep the same `package` method from the previous example because the location of the artifacts is declared by the `layout()`.

- Both the source folder (with headers) and the build folder (with libraries) are in the same location, in a path that follows:

  ```
  vendor_hello_library/{os}/{arch}
  ```

- The headers are in the `include` subfolder of the `self.source_folder` (we declare it in `self.cpp.source.includedirs`).
- The libraries are in the root of the `self.build_folder` folder (we declare `self.cpp.build.libdirs = ["."]`).
- We removed the `compiler` and the `build_type` because we only have different libraries depending on the operating system and the architecture (it might be a pure C library).

Now, for each different configuration we call **`conan export-pkg`** command, later we can list the binaries so we can check we have one package for each precompiled library:

```
$ conan export-pkg . -s os="Linux" -s arch="x86_64"
$ conan export-pkg . -s os="Linux" -s arch="armv8"
$ conan export-pkg . -s os="Macos" -s arch="x86_64"
$ conan export-pkg . -s os="Macos" -s arch="armv8"
$ conan export-pkg . -s os="Windows" -s arch="x86_64"
$ conan export-pkg . -s os="Windows" -s arch="armv8"

$ conan list packages hello/0.1#latest
Local Cache:
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:522dcea5982a3f8a5b624c16477e47195da2f84f
    settings:
      arch=x86_64
      os=Windows
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:63fead0844576fc02943e16909f08fcdddd6f44b
    settings:
      arch=x86_64
      os=Linux
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:82339cc4d6db7990c1830d274cd12e7c91ab18a1
    settings:
      arch=x86_64
      os=Macos
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:a0cd51c51fe9010370187244af885b0efcc5b69b
    settings:
      arch=armv8
      os=Windows
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:c93719558cf197f1df5a7f1d071093e26f0e44a0
    settings:
      arch=armv8
      os=Linux
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:dcf68e932572755309a5f69f3cee1bede410e907
    settings:
      arch=armv8
      os=Macos
```

In this example, we don't have a `test_package/` folder but you can provide one to test the packages like in the previous example.

Downloading and Packaging Pre-built Binaries

This is an example of scenario 3 explained in the introduction. If we are not building the libraries we likely have them somewhere in a remote repository. In this case, creating a complete Conan recipe, with the detailed retrieval of the binaries could be the preferred method, because it is reproducible, and the original binaries might be traced.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/other_packages/prebuilt_remote_binaries
```

Listing 49: conanfile.py

```
import os
from conan.tools.files import get, copy
from conan import ConanFile


class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "arch"

    def build(self):
        base_url = "https://github.com/conan-io/libhello/releases/download/0.0.1/"

        _os = {"Windows": "win", "Linux": "linux", "Macos": "macos"}.get(str(self.
↪settings.os))
        _arch = str(self.settings.arch).lower()
        url = "{}/{}_{}.tgz".format(base_url, _os, _arch)
        get(self, url)

    def package(self):
        copy(self, "*.h", self.build_folder, os.path.join(self.package_folder,
↪"include"))
        copy(self, "*.lib", self.build_folder, os.path.join(self.package_folder,
↪"lib"))
        copy(self, "*.a", self.build_folder, os.path.join(self.package_folder, "lib
↪"))

    def package_info(self):
        self.cpp_info.libs = ["hello"]
```

Typically, pre-compiled binaries come for different configurations, so the only task that the `build()` method has to implement is to map the `settings` to the different URLs.

We only need to call **`conan create`** with different settings to generate the needed packages:

```
$ conan create . -s os="Linux" -s arch="x86_64"
$ conan create . -s os="Linux" -s arch="armv8"
$ conan create . -s os="Macos" -s arch="x86_64"
$ conan create . -s os="Macos" -s arch="armv8"
$ conan create . -s os="Windows" -s arch="x86_64"
$ conan create . -s os="Windows" -s arch="armv8"

$ conan list packages hello/0.1#latest

Local Cache:
```

(continues on next page)

(continued from previous page)

```
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:522dcea5982a3f8a5b624c16477e47195da2f84f
    settings:
      arch=x86_64
      os=Windows
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:63fead0844576fc02943e16909f08fcdddd6f44b
    settings:
      arch=x86_64
      os=Linux
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:82339cc4d6db7990c1830d274cd12e7c91ab18a1
    settings:
      arch=x86_64
      os=Macos
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:a0cd51c51fe9010370187244af885b0efcc5b69b
    settings:
      arch=armv8
      os=Windows
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:c93719558cf197f1df5a7f1d071093e26f0e44a0
    settings:
      arch=armv8
      os=Linux
  hello/0.1
↪#a7068582757c24d362aac7d92f6a4a92:dcf68e932572755309a5f69f3cee1bede410e907
    settings:
      arch=armv8
      os=Macos
```

It is recommended to include also a small consuming project in a `test_package` folder to verify the package is correctly built, and then upload it to a Conan remote with **`conan upload`**.

The same building policies apply. Having a recipe fails if no Conan packages are created, and the **`--build`** argument is not defined. A typical approach for this kind of package could be to define a **`build_policy="missing"`**, especially if the URLs are also under the team's control. If they are external (on the internet), it could be better to create the packages and store them on your own Conan repository, so that the builds do not rely on third-party URLs being available.

Tool requires packages

In the "*Using build tools as Conan packages*" section we learned how to use a tool require to build (or help building) our project or Conan package. In this section we are going to learn how to create a recipe for a tool require.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/other_packages/tool_requires/tool
```

A simple tool require recipe

This is a recipe for a (fake) application that receiving a path returns 0 if the path is secure. We can check how the following simple recipe covers most of the `tool-require` use-cases:

Listing 50: conanfile.py

```
import os
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import copy


class secure_scannerRecipe(ConanFile):
    name = "secure_scanner"
    version = "1.0"
    package_type = "application"

    # Binary configuration
    settings = "os", "compiler", "build_type", "arch"

    # Sources are located in the same place as this recipe, copy them to the recipe
    exports_sources = "CMakeLists.txt", "src/*"

    def layout(self):
        cmake_layout(self)

    def generate(self):
        tc = CMakeToolchain(self)
        tc.generate()

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

    def package(self):
        extension = ".exe" if self.settings_build.os == "Windows" else ""
        copy(self, "*secure_scanner{}".format(extension),
             self.build_folder, os.path.join(self.package_folder, "bin"), keep_
↪path=False)

    def package_info(self):
        self.buildenv_info.define("MY_VAR", "23")
```

There are few relevant things in this recipe:

1. It declares `package_type = "application"`, this is optional but convenient, it will indicate conan that the current package doesn't contain headers or libraries to be linked. The consumers will know that this package is an application.
2. The `package()` method is packaging the executable into the `bin/` folder, that is declared by default as a bindir: `self.cpp_info.bindirs = ["bin"]`.
3. In the `package_info()` method, we are using `self.buildenv_info` to define a environment variable `MY_VAR` that will also be available in the consumer.

Let's create a binary package for the tool_require:

```
$ conan create .
...
secure_scanner/1.0: Calling package()
secure_scanner/1.0: Copied 1 file: secure_scanner
```

(continues on next page)

(continued from previous page)

```
secure_scanner/1.0 package(): Packaged 1 file: secure_scanner
...
Security Scanner: The path 'mypath' is secure!
```

Let's review the `test_package/conanfile.py`:

```
from conan import ConanFile


class secure_scannerTestConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"

    def requirements(self):
        self.tool_requires(self.tested_reference_str)

    def test(self):
        extension = ".exe" if self.settings_build.os == "Windows" else ""
        self.run("secure_scanner{} mypath".format(extension))
```

We are requiring the `secure_scanner` package as `tool_require` doing `self.tool_requires(self.tested_reference_str)`. In the `test()` method we are running the application, because it is available in the PATH. In the next example we are going to see why the executables from a `tool_require` are available in the consumers.

So, let's create a consumer recipe to test if we can run the `secure_scanner` application of the `tool_require` and read the environment variable. Go to the *examples2/tutorial/creating_packages/other_packages/tool_requires/consumer* folder:

Listing 51: conanfile.py

```
from conan import ConanFile

class MyConsumer(ConanFile):
    name = "my_consumer"
    version = "1.0"
    settings = "os", "arch", "compiler", "build_type"
    tool_requires = "secure_scanner/1.0"

    def build(self):
        extension = ".exe" if self.settings_build.os == "Windows" else ""
        self.run("secure_scanner{} {}".format(extension, self.build_folder))
        if self.settings_build.os != "Windows":
            self.run("echo MY_VAR=$MY_VAR")
        else:
            self.run("set MY_VAR")
```

In this simple recipe we are declaring a `tool_require` to `secure_scanner/1.0` and we are calling directly the packaged application `secure_scanner` in the `build()` method, also printing the value of the `MY_VAR` env variable.

If we build the consumer:

```
$ conan build .

-------- Installing (downloading, building) binaries... --------
secure_scanner/1.0: Already installed!
```

(continues on next page)

(continued from previous page)

```
-------- Finalizing install (deploy, generators) --------
...
conanfile.py (my_consumer/1.0): RUN: secure_scanner /Users/luism/workspace/examples2/
↪tutorial/creating_packages/other_packages/tool_requires/consumer
...
Security Scanner: The path '/Users/luism/workspace/examples2/tutorial/creating_
↪packages/other_packages/tool_requires/consumer' is secure!
...
MY_VAR=23
```

We can see that the executable returned 0 (because our folder is secure) and it printed `Security Scanner: The path is secure!` message. It also printed the "23" value assigned to `MY_VAR` but, why are these automatically available?

- The generators `VirtualBuildEnv` and `VirtualRunEnv` are automatically used.
- The `VirtualRunEnv` is reading the `tool-requires` and is creating a launcher like `conanbuildenv-release-x86_64.sh` appending all `cpp_info.bindirs` to the `PATH`, all the `cpp_info.libdirs` to the `LD_LIBRARY_PATH` environment variable and declaring each variable of `self.buildenv_info`.
- Every time conan executes the `self.run`, by default, activates the `conanbuild.sh` file before calling any command. The `conanbuild.sh` is including the `conanbuildenv-release-x86_64.sh`, so the application is in the PATH and the enviornment variable "MYVAR" has the value declared in the tool-require.

Removing settings in package_id()

With the previous recipe, if we call **`conan create`** with different setting like different compiler versions, we will get different binary packages with a different `package ID`. This might be convenient to, for example, keep better traceability of our tools. In this case, the <MISSING PAGE> compatibility.py plugin can help to locate the best matching binary in case Conan doesn't find the binary for our specific compiler version.

But in some cases we might want to just generate a binary taking into account only the `os`, `arch` or at most adding the `build_type` to know if the application is built for Debug or Release. We can add a `package_id()` method to remove them:

Listing 52: conanfile.py

```
import os
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import copy


class secure_scannerRecipe(ConanFile):
    name = "secure_scanner"
    version = "1.0"
    settings = "os", "compiler", "build_type", "arch"
    ...

    def package_id(self):
        del self.info.settings.compiler
        del self.info.settings.build_type
```

So, if we call **`conan create`** with different `build_type` we will get exactly the same `package_id`.

```
$ conan create .
...
Package '82339cc4d6db7990c1830d274cd12e7c91ab18a1' created

$ conan create . -s build_type=Debug
...
Package '82339cc4d6db7990c1830d274cd12e7c91ab18a1' created
```

We got the same binary `package_id`. The second `conan create . -s build_type=Debug` created and overwrote (created a newer package revision) of the previous Release binary, because they have the same `package_id` identifier. It is typical to create only the `Release` one, and if for any reason managing both Debug and Release binaries is intended, then the approach would be not removing the `del self.info.settings.build_type`

Read more

- Toolchains (compilers)
- Usage of *self.rundenv_info*
- `settings_target`

4.3 Versioning and Continuous Integration

Intro to versions, semver

Intro to version ranges

Intro to revisions

Intro to lockfiles

Intro to versions conflicts

4.3.1 Version ranges

4.3.2 Revisions

4.3.3 Lockfiles

4.3.4 Version conflicts

Explain briefly about potential version conflicts with diamond graphs. Use figure of graph.

```
$  git clone git@github.com:conan-io/examples2.git
$ cd tutorial/consuming/versioning/conflicts
$ conan create math --version=1.0
$ conan create math --version=2.0
$ conan create engine
$ conan create ai
$ conan install game

> ERROR: Version conflicts
# TODO: This message will change and improve
```

The consumer, in this case "game" can decide which is the resolution

CHAPTER

FIVE

INTEGRATIONS

CHAPTER

SIX

EXAMPLES

6.1 Custom commands

6.1.1 Custom command: Clean old recipe and package revisions

Please, first of all, clone the sources to recreate this project. You can find them in the examples2.0 repository in GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/examples/extensions/commands/clean
```

In this example we are going to see how to create/use a custom command: **conan clean**. It removes every recipe and its package revisions from the local cache or the remotes, except the latest package revision from the latest recipe one.

Note: To understand better this example, it is highly recommended to read previously the *Custom commands reference*.

Locate the command

Copy the command file `cmd_clean.py` into your `[YOUR_CONAN_HOME]/extensions/commands/` folder (create it if it's not there). If you don't know where `[YOUR_CONAN_HOME]` is located, you can run **conan config home** to check it.

Run it

Now, you should be able to see the new command in your command prompt:

```
$ conan -h
...
Custom commands
clean        Deletes (from local cache or remotes) all recipe and package revisions␣
↪but the
               latest package revision from the latest recipe revision.

$ conan clean -h
usage: conan clean [-h] [-r REMOTE] [--force]

Deletes (from local cache or remotes) all recipe and package revisions but
the latest package revision from the latest recipe revision.

optional arguments:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  -r REMOTE, --remote REMOTE
                        Will remove from the specified remote
  --force               Remove without requesting a confirmation
```

Finally, if you execute **`conan clean`**:

```
$ conan clean
Do you want to remove all the recipes revisions and their packages ones, except the␣
↪latest package revision from the latest recipe one? (yes/no): yes
other/1.0
Removed package revision: other/1.0
↪#31da245c3399e4124e39bd4f77b5261f:da39a3ee5e6b4b0d3255bfef95601890afd80709
↪#a16985deb2e1aa73a8480faad22b722c [Local cache]
Removed recipe revision: other/1.0#721995a35b1a8d840ce634ea1ac71161 and all its␣
↪package revisions [Local cache]
hello/1.0
Removed package revision: hello/1.0
↪#9a77cdcff3a539b5b077dd811b2ae3b0:da39a3ee5e6b4b0d3255bfef95601890afd80709
↪#cee90a74944125e7e9b4f74210bfec3f [Local cache]
Removed package revision: hello/1.0
↪#9a77cdcff3a539b5b077dd811b2ae3b0:da39a3ee5e6b4b0d3255bfef95601890afd80709
↪#7cddd50952de9935d6c3b5b676a34c48 [Local cache]
libcxx/0.1
```

Nothing should happen if you run it again:

```
$ conan clean
Do you want to remove all the recipes revisions and their packages ones, except the␣
↪latest package revision from the latest recipe one? (yes/no): yes
other/1.0
hello/1.0
libcxx/0.1
```

Code tour

The `conan clean` command has the following code:

Listing 1: cmd_clean.py

```python
from conan.api.conan_api import ConanAPIV2
from conan.api.output import ConanOutput, Color
from conan.cli.command import OnceArgument, conan_command
from conans.client.userio import UserInput

recipe_color = Color.BRIGHT_BLUE
removed_color = Color.BRIGHT_YELLOW


@conan_command(group="Custom commands")
def clean(conan_api: ConanAPIV2, parser, *args):
    """
    Deletes (from local cache or remotes) all recipe and package revisions but
    the latest package revision from the latest recipe revision.
    """
    parser.add_argument('-r', '--remote', action=OnceArgument,
```

(continues on next page)

(continued from previous page)

```
                        help='Will remove from the specified remote')
    parser.add_argument('--force', default=False, action='store_true',
                        help='Remove without requesting a confirmation')
    args = parser.parse_args(*args)

    def confirmation(message):
        return args.force or ui.request_boolean(message)

    ui = UserInput(non_interactive=False)
    out = ConanOutput()
    remote = conan_api.remotes.get(args.remote) if args.remote else None
    output_remote = remote or "Local cache"

    # Getting all the recipes
    recipes = conan_api.search.recipes("*/*", remote=remote)
    if recipes and not confirmation("Do you want to remove all the recipes revisions␣
↪and their packages ones, "
                                    "except the latest package revision from the␣
↪latest recipe one?"):
        return
    for recipe in recipes:
        out.writeln(f"{str(recipe)}", fg=recipe_color)
        all_rrevs = conan_api.list.recipe_revisions(recipe, remote=remote)
        latest_rrev = all_rrevs[0] if all_rrevs else None
        for rrev in all_rrevs:
            if rrev != latest_rrev:
                conan_api.remove.recipe(rrev, remote=remote)
                out.writeln(f"Removed recipe revision: {rrev.repr_notime()} "
                            f"and all its package revisions [{output_remote}]",␣
↪fg=removed_color)
            else:
                all_prevs = conan_api.search.package_revisions(f"{rrev.repr_notime()}
↪:*#*", remote=remote)
                latest_prev = all_prevs[0] if all_prevs else None
                for prev in all_prevs:
                    if prev != latest_prev:
                        conan_api.remove.package(prev, remote=remote)
                        out.writeln(f"Removed package revision: {prev.repr_notime()} [
↪{output_remote}]", fg=removed_color)
```

Let's analize the most important parts.

parser

The `parser` param is an instance of the Python command-line parsing `argparse.ArgumentParser`, so if you want to know more about its API, visit its official website.

User input and user output

Important classes to manage user input and user output:

```
ui = UserInput(non_interactive=False)
out = ConanOutput()
```

- `UserInput(non_interactive)`: class to manage user inputs. In this example we're using `ui.request_boolean("Do you want to proceed?")`, so it'll be automatically translated

to `Do you want to proceed? (yes/no):` in the command prompt. **Note**: you can use `UserInput(non_interactive=conan_api.config.get("core:non_interactive"))` too.

- `ConanOutput()`: class to manage user outputs. In this example, we're using only `out.writeln(message, fg=None, bg=None)` where `fg` is the font foreground, and `bg` is the font background. Apart from that, you have some predefined methods like `out.info()`, `out.success()`, `out.error()`, etc.

Conan public API

The most important part of this example is the usage of the Conan API via `conan_api` parameter. These are some examples which are being used in this custom command:

```
conan_api.remotes.get(args.remote)
conan_api.search.recipes("*/*", remote=remote)
conan_api.list.recipe_revisions(recipe, remote=remote)
conan_api.remove.recipe(rrev, remote=remote)
conan_api.search.package_revisions(f"{rrev.repr_notime()}:*#*", remote=remote)
conan_api.remove.package(prev, remote=remote)
```

- `conan_api.remotes.get(...)`: `[RemotesAPI]` Returns a RemoteRegistry given the remote name.
- `conan_api.search.recipes(...)`: `[SearchAPI]` Returns a list with all the recipes matching the given pattern.
- `conan_api.list.recipe_revisions(...)`: `[ListAPI]` Returns a list with all the recipe revisions given a recipe reference.
- `conan_api.remove.recipe(...)`: `[RemoveAPI]` Removes the given recipe revision.
- `conan_api.search.package_revisions(...)`: `[SearchAPI]` Returns the list of package revisions for a given recipe revision.
- `conan_api.remove.package(...)`: `[RemoveAPI]` Removes the given package revision.

Besides that, it deserves especial attention these lines:

```
all_rrevs = conan_api.list.recipe_revisions(recipe, remote=remote)
latest_rrev = all_rrevs[0] if all_rrevs else None
# .......
all_prevs = conan_api.search.package_revisions(f"{rrev.repr_notime()}:*#*",␣
↪remote=remote)
latest_prev = all_prevs[0] if all_prevs else None
```

Basically, these API calls are returning a list of recipe revisions and package ones respectively, but we're saving the first element as the latest one because these calls are getting an ordered list always. Take into account that it's using `rrev.repr_notime()` because it represents the recipe revision as a string and pruning the timestamps.

If you want to know more about the Conan API, visit the *ConanAPIV2 section*

6.2 tools.cmake

6.2.1 CMakeToolchain: Building your project using CMakePresets

In this example we are going to see how to use `CMakeToolchain`, predefined layouts like `cmake_layout` and the `CMakePresets` CMake feature.

Let's create a basic project based on the template `cmake_exe` as an example of a C++ project:

```
$ conan new -d name=foo -d version=1.0 cmake_exe
```

Generating the toolchain

The recipe from our project declares the generator "CMakeToolchain".

We can call **conan install** to install both `Release` and `Debug` configurations. The `conan_toolchain.cmake` is common for both configurations and located at *build/generators* folder:

```
$ conan install .
$ conan install . -s build_type=Debug
```

Building the project using `CMakePresets`

A `CMakeUserPresets.json` file is generated in the same folder of your `CMakeLists.txt` file, so you can use the `--preset` argument from `cmake >= 3.23` or use an IDE that supports it.

The `CMakeUserPresets.json` is including the `CMakePresets.json` file located at the `build/generators` folder.

The `CMakePresets.json` contain information about the `conan_toolchain.cmake` location and even the `binaryDir` set with the output directory.

Note: CMake >= 3.23 is required because the "include" from `CMakeUserPresets.json` to `CMakePresets.json` is only supported since that version.

If you are using a multi-configuration generator:

```
$ cmake --preset default
$ cmake --build --preset Debug
$ build\Debug\foo.exe
foo/1.0: Hello World Release!

$ cmake --build --preset Release
$ build\Release\foo.exe
foo/1.0: Hello World Release!
```

If you are using a single-configuration generator:

```
$ cmake --preset Debug
$ cmake --build --preset Debug
$ ./build/Debug/foo
foo/1.0: Hello World Debug!


$ cmake --preset Release
$ cmake --build --preset Release
$ ./build/Release/foo
foo/1.0: Hello World Release!
```

Note that we did'nt need to create the `build/Release` or `build/Debug` folders, as we did *in the tutorial*. The output directory is declared by the `cmake_layout()` and automatically managed by the CMake Presets feature.

This behavior is also managed automatically by Conan (with CMake >= 3.15) when you build a package in the Conan cache (with **conan create** command). The CMake >= 3.23 is not required.

6.3 tools.files

6.3.1 Patching sources

In this example we are going to see how to patch the source code. This is necessary sometimes, specially when you are creating a package for a third party library. A patch might be required in the build system scripts or even in the source code of the library if you want, for example, to apply a security patch.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples/tools/files/patches
```

Patching using 'replace_in_file'

The simplest way to patch a file is using the `replace_in_file` tool in your recipe. It searches in a file the specified string and replaces it with another string.

in source() method

The source() method is called only once for all the configurations (different calls to **`conan create`** for different settings/options) so you should patch only in the `source()` method if the changes are common for all the configurations.

Look at the `source()` method at the `conanfile.py`:

```
import os
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import get, replace_in_file


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    # Binary configuration
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",␣
↪strip_root=True)
        replace_in_file(self, os.path.join(self.source_folder, "src", "hello.cpp"),
↪"Hello World", "Hello Friends!")

    ...
```

We are replacing the `"Hello World"` string with "Hello Friends!". We can run `conan create .` and verify that if the replace was done:

```
$ conan create .
...
-------- Testing the package: Running test() --------
hello/1.0: Hello Friends! Release!
...
```

in build() method

In this case, we need to apply a different patch depending on the configuration (*self.settings*, *self.options*...), so it has to be done in the `build()` method. Let's modify the recipe to introduce a change that depends on the `self.options.shared`:

```
class helloRecipe(ConanFile):

    ...

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",␣
↪strip_root=True)

    def build(self):
        replace_in_file(self, os.path.join(self.source_folder, "src", "hello.cpp"),
                        "Hello World",
                        "Hello {} Friends!".format("Shared" if self.options.shared␣
↪else "Static"))
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

    ...
```

If we call `conan create` with different `option.shared` we can check the output:

```
$ conan create .
...
hello/1.0: Hello Static Friends! Release!
...

$ conan create . -o shared=True
...
hello/1.0: Hello Shared Friends! Debug!
...
```

Patching using "patch" tool

If you have a patch file (diff between two versions of a file), you can use the `conan.tools.files.patch` tool to apply it. The rules about where to apply the patch (`source()` or `build()` methods) are the same.

We have this patch file, where we are changing again the message to say "Hello Patched World Release!":

```
--- a/src/hello.cpp
+++ b/src/hello.cpp
@@ -3,9 +3,9 @@

 void hello(){
     #ifdef NDEBUG
-    std::cout << "hello/1.0: Hello World Release!\n";
+    std::cout << "hello/1.0: Hello Patched World Release!\n";
     #else
-    std::cout << "hello/1.0: Hello World Debug!\n";
+    std::cout << "hello/1.0: Hello Patched World Debug!\n";
     #endif

     // ARCHITECTURES
```

Edit the `conanfile.py` to:

1. Import the `patch` tool.
2. Add `exports_sources` to the patch file so we have it available in the cache.
3. Call the `patch` tool.

```
import os
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import get, replace_in_file, patch


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    # Binary configuration
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}
    exports_sources = "*.patch"

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",␣
↪strip_root=True)
        patch_file = os.path.join(self.export_sources_folder, "hello_patched.patch")
        patch(self, patch_file=patch_file)

    ...
```

We can run "conan create" and see that the patch worked:

```
$ conan create .
...
-------- Testing the package: Running test() --------
hello/1.0: Hello Patched World Release!
...
```

We can also use the `conandata.yml` *introduced in the tutorial* so we can declare the patches to apply for each version:

```
patches:
  "1.0":
    - patch_file: "hello_patched.patch"
```

And there are the changes we introduce in the `source()` method:

```
.. code-block:: python

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",␣
↪strip_root=True)
        patches = self.conan_data["patches"][self.version]
        for p in patches:
            patch_file = os.path.join(self.export_sources_folder, p["patch_file"])
            patch(self, patch_file=patch_file)
```

Check *patch* for more details.

If we run the **conan create**, the patch is also applied:

```
$ conan create .
...
-------- Testing the package: Running test() --------
hello/1.0: Hello Patched World Release!
...
```

Patching using "apply_conandata_patches" tool

The example above works but it is a bit complex. If you follow the same yml structure (check the *apply_conandata_patches* to see the full supported yml) you only need to call `apply_conandata_patches`:

```
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import get, apply_conandata_patches


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",␣
↪strip_root=True)
        apply_conandata_patches(self)
```

Let's check if the patch is also applied:

```
$ conan create .
...
-------- Testing the package: Running test() --------
hello/1.0: Hello Patched World Release!
...
```

6.4 tools.meson

6.4.1 Build a simple Meson project using Conan

In this example, we are going to create a string compressor application that uses one of the most popular C++ libraries: Zlib.

Note: This example is based on the main *Build a simple CMake project using Conan* tutorial. So we highly recommend reading it before trying out this one.

Important: We have to retrieve the Zlib Conan package from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

We'll use Meson as build system and pkg-config as helper tool in this case, so you should get them installed before going forward with this example.

Please, at first, clone the sources to recreate this project, you can find them in the examples2.0 repository in GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/examples/tools/meson/mesontoolchain/simple_meson_project
```

We start from a very simple C language project with this structure:

```
.
├── meson.build
└── src
    └── main.c
```

This project contains a basic *meson.build* including the **zlib** dependency and the source code for the string compressor program in *main.c*.

Let's have a look at the *main.c* file:

Listing 2: **main.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <zlib.h>

int main(void) {
    char buffer_in [256] = {"Conan is a MIT-licensed, Open Source package manager for␣
↪C and C++ development "
                            "for C and C++ development, allowing development teams to␣
↪easily and efficiently "
                            "manage their packages and dependencies across platforms␣
↪and build systems."};
    char buffer_out [256] = {0};

    z_stream defstream;
    defstream.zalloc = Z_NULL;
    defstream.zfree = Z_NULL;
    defstream.opaque = Z_NULL;
    defstream.avail_in = (uInt) strlen(buffer_in);
    defstream.next_in = (Bytef *) buffer_in;
    defstream.avail_out = (uInt) sizeof(buffer_out);
    defstream.next_out = (Bytef *) buffer_out;

    deflateInit(&defstream, Z_BEST_COMPRESSION);
    deflate(&defstream, Z_FINISH);
    deflateEnd(&defstream);

    printf("Uncompressed size is: %lu\n", strlen(buffer_in));
    printf("Compressed size is: %lu\n", strlen(buffer_out));

    printf("ZLIB VERSION: %s\n", zlibVersion());

    return EXIT_SUCCESS;
}
```

Also, the contents of *meson.build* are:

Listing 3: **meson.build**

```
project('tutorial', 'c')
zlib = dependency('zlib', version : '1.2.11')
executable('compressor', 'src/main.c', dependencies: zlib)
```

Let's create a *conanfile.txt* with the following content to install **Zlib**:

Listing 4: **conanfile.txt**

```
[requires]
zlib/1.2.11

[generators]
PkgConfigDeps
MesonToolchain
```

In this case, we will use *PkgConfigDeps* to generate information about where the **Zlib** library files are installed thanks to the **.pc* files and *MesonToolchain* to pass build information to *Meson* using a *conan_meson_[native|cross].ini* file that describes the native/cross compilation environment, which in this case is a *conan_meson_native.ini* one.

We will use Conan to install **Zlib** and generate the files that Meson needs to find this library and build our project. We will generate those files in the folder *build*. To do that, run:

```
$ conan install . --output-folder=build --build=missing
```

Now we are ready to build and run our **compressor** app:

Listing 5: Windows

```
$ cd build
$ meson setup --native-file conan_meson_native.ini .. meson-src
$ meson compile -C meson-src
$ meson-src\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

Listing 6: Linux, macOS

```
$ cd build
$ meson setup --native-file conan_meson_native.ini .. meson-src
$ meson compile -C meson-src
$ ./meson-src/compressor
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

6.5 Cross-building to Android

6.5.1 Cross building to Android with the NDK

In this example, we are going to see how to cross-build a Conan package to Android.

First of all, download the Android NDK from the download page and unzip it. In MacOS you can also install it with `brew install android-ndk`.

Then go to the `profiles` folder in the conan config home directory (check it running **`conan config home`**) and create a file named `android` with the following contents:

```
include(default)

[settings]
os=Android
os.api_level=21
arch=armv8
compiler=clang
compiler.version=12
compiler.libcxx=c++_static
compiler.cppstd=14

[conf]
tools.android:ndk_path=/usr/local/share/android-ndk
```

You might need to modify:

- `compiler.version`: Check the NDK documentation or find a `bin` folder containing the compiler executables like `x86_64-linux-android31-clang`. In a Macos installation it is found in the NDK path + `toolchains/llvm/prebuilt/darwin-x86_64/bin`. Run `./x86_64-linux-android31-clang --version` to check the running `clang` version and adjust the profile.
- `compiler.libcxx`: The supported values are `c++_static` and `c++_shared`.
- `compiler.cppstd`: The C++ standard version, adjust as your needs.
- `os.api_level`: You can check here the usage of each Android Version/API level and choose the one that fits better with your requirements. This is typically a balance between new features and more compatible applications.
- `arch`: There are several architectures supported by Android: `x86`, `x86_64`, `armv7`, and `armv8`.
- `tools.android:ndk_path` conf: Write the location of the unzipped NDK.

Use the **`conan new`** command to create a "Hello World" C++ library example project:

```
$ conan new cmake_lib -d name=hello -d version=1.0
```

Then we can specify the `android` profile and our hello library will be built for Android:

```
$ conan create . --profile android

[ 50%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[100%] Linking CXX static library libhello.a
[100%] Built target hello
...
[ 50%] Building CXX object CMakeFiles/example.dir/src/example.cpp.o
[100%] Linking CXX executable example
[100%] Built target example
```

Both the library and the `test_package` executable are built for Android, so we cannot use them in our local computer.

Unless you have access to a *root* Android device, running the test application or using the built library is not possible directly so it is more common to build an Android application that uses the `hello` library.

Read more

- Check the example *Integrating Conan in Android Studio* to know how to use your c++ libraries in a native Android application.
- Check the tutorial *How to cross-compile your applications using Conan*.

6.5.2 Integrating Conan in Android Studio

At the *Cross building to Android with the NDK* we learned how to build a package for Android using the NDK. In this example we are going to learn how to do it with the Android Studio and how to use the libraries in a real Android application.

Creating a new project

First of all, download and install the Android Studio IDE.

Then create a `new project` selecting `Native C++` from the templates.

In the next wizard window, select a name for your application, for example *MyConanApplication*, you can leave the "Minimum SDK" with the suggested value (21 in our case), but remember the value as we are using it later in the Conan profile at `os.api_level`

Select a "C++ Standard" in the next window, again, remember the choice as later we should use the same in the profile at `compiler.cppstd`.

Important: In this example, we will retrieve Conan packages from a Conan repository with packages compatible with Conan 2.0. To run this example successfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

In the project generated with the wizard we have a folder `cpp` with a `native-lib.cpp`. We are going to modify that file to use `zlib` and print a message with the used `zlib` version. Copy only the highlighted lines, it is important to keep the function name.

Listing 7: native-lib.cpp

```
#include <jni.h>
#include <string>
#include "zlib.h"

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_myconanapp_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject /* this */) {
    std::string hello = "Hello from C++, zlib version: ";
    hello.append(zlibVersion());
    return env->NewStringUTF(hello.c_str());
}
```

Now we are going to learn how to introduce a requirement to the `zlib` library and how to prepare our project.

Introducing dependencies with Conan

conanfile.txt

We need to provide the `zlib` package with Conan. Create a file `conanfile.txt` in the `cpp` folder:

Listing 8: conanfile.txt

```
[requires]
zlib/1.2.12

[generators]
CMakeToolchain
CMakeDeps

[layout]
cmake_layout
```

build.gradle

We are going to automate calling `conan install` before building the Android project, so the requires are prepared, open the `build.gradle` file in the `My_Conan_App.app` (Find it in the *Gradle Scripts* section of the Android project view). Paste the `task conanInstall` contents after the `plugins` and before the `android` elements:

Listing 9: build.gradle

```
plugins {
  ...
}


task conanInstall {
    def buildDir = new File("app/build")
    buildDir.mkdirs()
    ["Debug", "Release"].each { String build_type ->
        ["armv7", "armv8", "x86", "x86_64"].each { String arch ->
            def cmd = "conan install " +
                      "../src/main/cpp --profile android -s build_type="+ build_type +"␣
↪-s arch=" + arch +
                      " --build missing -c tools.cmake.cmake_layout:build_folder_vars=[
↪'settings.arch']"
            print(">> ${cmd} \n")

            def sout = new StringBuilder(), serr = new StringBuilder()
            def proc = cmd.execute(null, buildDir)
            proc.consumeProcessOutput(sout, serr)
            proc.waitFor()
            println "$sout $serr"
            if (proc.exitValue() != 0) {
                throw new Exception("out> $sout err> $serr" + "\nCommand: ${cmd}")
            }
        }
    }
}

android {
    compileSdk 32

    defaultConfig {

...
```

The `conanInstall` task is calling **`conan install`** for Debug/Release and for each architecture we want to

build, you can adjust these values to match your requirements.

If we focus on the `conan install` task we can see:

1. We are passing a `--profile android`, so we need to create the proile. Go to the `profiles` folder in the conan config home directory (check it running **`conan config home`**) and create a file named `android` with the following contents:

```
include(default)

[settings]
os=Android
os.api_level=21
compiler=clang
compiler.version=12
compiler.libcxx=c++_static
compiler.cppstd=14

[conf]
tools.android:ndk_path=/Users/luism/Library/Android/sdk/ndk/21.4.7075529/
```

   You might need to modify:

   - `tools.android:ndk_path` conf: The location of the NDK provided by Android Studio. You should be able to see the path to the NDK if you open the `cpp/includes` folder in your IDE.
   - `compiler.version`: Check the NDK documentation or find a `bin` folder containing the compiler executables like `x86_64-linux-android31-clang`. In a Macos installation it is found in the NDK path + `toolchains/llvm/prebuilt/darwin-x86_64/bin`. Run `./x86_64-linux-android31-clang --version` to check the running `clang` version and adjust the profile.
   - `compiler.libcxx`: The supported values are `c++_static` and `c++_shared`.
   - `compiler.cppstd`: The C++ standard version, this should be the value you selected in the Wizard.
   - `os.api_level`: Use the same value you selected in the Wizard.

2. We are passing `-c tools.cmake.cmake_layout:build_folder_vars=['settings.arch']`, thanks to that, Conan will create a different folder for the specified `settings.arch` so we can have all the configurations available at the same time.

To make Conan work we need to pass CMake a custom toolchain. We can do it introducing a single line in the same file, in the `android/defaultConfig/externalNativeBuild/cmake` element:

Listing 10: build.gradle

```
android {
    compileSdk 32

    defaultConfig {
        applicationId "com.example.myconanapp"
        minSdk 21
        targetSdk 21
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
        externalNativeBuild {
            cmake {
```

(continues on next page)

(continued from previous page)

```
                cppFlags '-v'
                arguments("-DCMAKE_TOOLCHAIN_FILE=conan_android_toolchain.cmake")
            }
        }
```

conan_android_toolchain.cmake

Create a file called `conan_android_toolchain.cmake` in the `cpp` folder, that file will be responsible of including the right toolchain depending on the `ANDROID_ABI` variable that indicates the build configuration that the IDE is currently running:

Listing 11: conan_android_toolchain.cmake

```
if(${ANDROID_ABI} STREQUAL "x86_64")
    include("${CMAKE_CURRENT_LIST_DIR}/build/x86_64/generators/conan_toolchain.cmake")
elseif(${ANDROID_ABI} STREQUAL "x86")
    include("${CMAKE_CURRENT_LIST_DIR}/build/x86/generators/conan_toolchain.cmake")
elseif(${ANDROID_ABI} STREQUAL "arm64-v8a")
    include("${CMAKE_CURRENT_LIST_DIR}/build/armv8/generators/conan_toolchain.cmake")
elseif(${ANDROID_ABI} STREQUAL "armeabi-v7a")
    include("${CMAKE_CURRENT_LIST_DIR}/build/armv7/generators/conan_toolchain.cmake")
else()
    message(FATAL "Not supported configuration")
endif()
```

CMakeLists.txt

Finally, we need to modify the `CMakeLists.txt` to link with the `zlib` library:

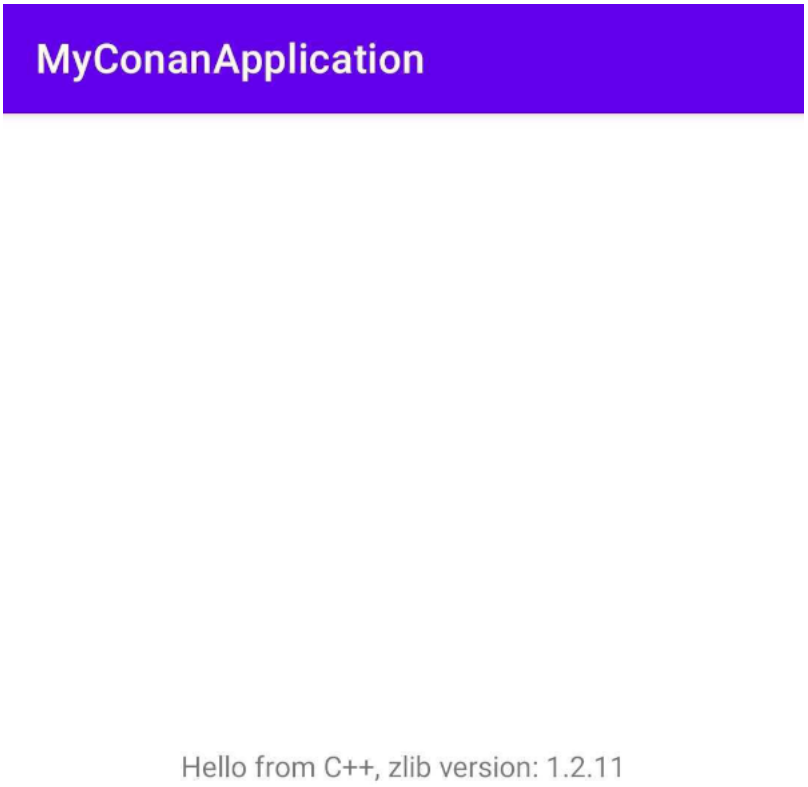
Listing 12: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.18.1)
project("myconanapp")
add_library(myconanapp SHARED native-lib.cpp)

find_library(log-lib log)

find_package(ZLIB CONFIG)

target_link_libraries(myconanapp ${log-lib} ZLIB::ZLIB)
```

Building the application

If we build our project we can see that *conan install* is called multiple times building the different configurations of `zlib`.

Then if we run the application in a Virtual Device or in a real device pairing it with the QR code we can see:



Once we have our project configured, it is very easy to change our dependencies and keep developing the application, for example, we can edit the `conanfile.txt` file and change the `zlib` to the version `1.12.2`:

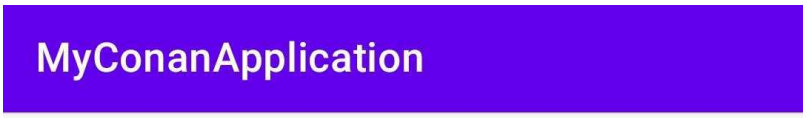
```
[requires]
zlib/1.2.12

[generators]
CMakeToolchain
CMakeDeps
```

(continues on next page)

(continued from previous page)

```
[layout]
cmake_layout
```

If we click build and then run the application, we will see that the zlib dependency has been updated:



Hello from C++, zlib version: 1.2.12

CHAPTER

SEVEN

REFERENCE

7.1 conanfile.py

The `conanfile.py` is the recipe file of a package, responsible for defining how to build it and consume it.

```
from conan import ConanFile

class HelloConan(ConanFile):
    ...
```

Important: *conanfile.py* recipes uses a variety of attributes and methods to operate. In order to avoid collisions and conflicts, follow these rules:

- Public attributes and methods, like `build()`, `self.package_folder`, are reserved for Conan. Don't use public members for custom fields or methods in the recipes.
- Use "protected" access for your own members, like `self._my_data` or `def _my_helper(self):`. Conan only reserves "protected" members starting with `_conan`.

Contents:

7.1.1 Attributes

name

`ConanFile.`**`name = None`**

String corresponding to the `<name>` at the recipe reference `<name>/version@user/channel`

A valid name has:

- A minimum of 2 and a maximum of 101 characters (though shorter names are recommended).
- Matches the following regex `^[a-z0-9_][a-z0-9_+.-]{1,100}$`: so starts with alphanumeric or _, then from 1 to 100 characters between alphanumeric, _, +, . or -.

The name is only necessary for `export`-ing the recipe into the local cache (`export` and `create` commands), if they are not defined in the command line.

version

`ConanFile.`**`version = None`**

String corresponding to the `<version>` at the recipe reference `name/<version>@user/channel`

A valid version follows the same rules than the `name` attribute. In case the version follows semantic versioning in the form `X.Y.Z-pre1+build2`, that value might be used for requiring this package through version ranges instead of exact versions.

The version is only strictly necessary for `export`-ing the recipe into the local cache (`export` and `create` commands), if they are not defined in the command line.

package_type

`ConanFile.`**`package_type = None`**

Optional. Declaring the `package_type` will help Conan:

- To choose better the default `package_id_mode` for each dependency, that is, how a change in a dependency should affect the `package_id` to the current package.
- Which information from the dependencies should be propagated to the consumers, like headers, libraries, runtime information...

The valid values are:

- **application**: The package is an application.
- **library**: The package is a generic library. It will try to determine the type of library (from `shared-library`, `static-library`, `header-library`) reading the `self.options.shared` (if declared) and the `self.options.header_only`
- **shared-library**: The package is a shared library.
- **static-library**: The package is a static library.
- **header-library**: The package is a header only library.
- **build-scripts**: The package only contains build scripts.
- **python-require**: The package is a python require.
- **unknown**: The type of the package is unknown.

description

`ConanFile.`**`description = None`**

Description of the package and any information that might be useful for the consumers. The first line might be used as a short description of the package.

This is an optional, but strongly recommended text field, containing the description of the package, and any information that might be useful for the consumers. The first line might be used as a short description of the package.

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    description = """This is a Hello World library.
                     A fully featured, portable, C++ library to say Hello World in␣
↪the stdout,
                     with incredible iostreams performance"""
```

homepage

`ConanFile.`**`homepage = None`**

The home web page of the library being packaged.

Used to link the recipe to further explanations of the library itself like an overview of its features, documentation, FAQ as well as other related information.

```
class EigenConan(ConanFile):
    name = "eigen"
    version = "3.3.4"
    homepage = "http://eigen.tuxfamily.org"
```

url

`ConanFile.`**`url = None`**

URL of the package repository, i.e. not necessarily of the original source code. Recommended, but not mandatory attribute.

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    url = "https://github.com/conan-io/hello.git"
```

license

`ConanFile.`**`license = None`**

License of the **target** source code and binaries, i.e. the code that is being packaged, not the `conanfile.py` itself. Can contain several, comma separated licenses. It is a text string, so it can contain any text, but it is strongly recommended that recipes of Open Source projects use SPDX identifiers from the SPDX license list

This will help people wanting to automate license compatibility checks, like consumers of your package, or you if your package has Open-Source dependencies.

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    license = "MIT"
```

author

`ConanFile.`**`author = None`**

Main maintainer/responsible for the package, any format. This is an optional attribute.

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    author = "John J. Smith (john.smith@company.com)"
```

topics

`ConanFile.`**`topics = None`**

Tags to group related packages together and describe what the code is about. Used as a search filter in conan-center. Optional attribute. It should be a tuple of strings.

```
class ProtocInstallerConan(ConanFile):
    name = "protoc_installer"
    version = "0.1"
    topics = ("protocol-buffers", "protocol-compiler", "serialization", "rpc")
```

user, channel

ConanFile.**user = None**
String corresponding to the <user> at the recipe reference name/version@<user>/channel

A valid string for the user field follows the same rules than the name attribute. This is an optional attribute. It is sometimes used to identify a forked recipe, giving a different namespace to the recipe reference.

ConanFile.**channel = None**
String corresponding to the <channel> at the recipe reference name/version@user/<channel>.

A valid string for the channel field follows the same rules than the name attribute. This is an optional attribute. It is sometimes used to identify a maturity of the package (stable, testing...).

The value of these fields can be accessed from within a conanfile.py:

```
from conan import ConanFile

class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"

    def requirements(self):
        self.requires("common-lib/version")
        if self.user and self.channel:
            # If the recipe is using them, I want to consume my fork.
            self.requires("say/0.1@%s/%s" % (self.user, self.channel))
        else:
            # otherwise, I'll consume the community one
            self.requires("say/0.1")
```

Only packages that have already been exported (packages in the local cache or in a remote server) can have a user/channel assigned. For package recipes working in the user space, there is no current user/channel by default, although they can be defined at **conan install** time with:

```
$ conan install <path to conanfile.py> --user my_user --channel my_channel
```

settings

ConanFile.**settings = None**
List of strings with the first level settings (from settings.yml) that the recipe need, because:

- They are read for building (e.g: *if self.settings.compiler == "gcc"*)
- They affect the package_id. If a value of the declared setting changes, the package_id has to be different.

The most common is to declare:

```
settings = "os", "compiler", "build_type", "arch"
```

Once the recipe is loaded by Conan, the settings are processed and they can be read in the recipe, also the sub-settings:

```
settings = "os", "arch"

def build(self):
    if self.settings.compiler == "gcc":
        if self.settings.compiler.cppstd == "gnu20":
            # do some special build commands
```

If you try to access some setting that doesn't exist, like `self.settings.compiler.libcxx` for the `msvc` setting, Conan will fail telling that `libcxx` does not exist for that compiler.

If you want to do a safe check of settings values, you could use the `get_safe()` method:

```
def build(self):
    # Will be None if doesn't exist (not declared)
    arch = self.settings.get_safe("arch")
    # Will be None if doesn't exist (doesn't exist for the current compiler)
    compiler_version = self.settings.get_safe("compiler.version")
    # Will be the default version if the return is None
    build_type = self.settings.get_safe("build_type", default="Release")
```

The `get_safe()` method will return `None` if that setting or sub-setting doesn't exist and there is no default value assigned.

See also:

- Removing settings in the `package_id()` method. <MISSING PAGE>

options

`ConanFile.`**`options = None`**

Dictionary with traits that affects only the current recipe, where the key is the option name and the value is a list of different values that the option can take. By default any value change in an option, changes the `package_id`. Check the `default_options` field to define default values for the options.

Values for each option can be typed or plain strings (`"value"`, `True`, `42`,...).

There are two special values:

- `None`: Allow the option to have a `None` value (not specified) without erroring.
- `"ANY"`: For options that can take any value, not restricted to a set.

```
class MyPkg(ConanFile):
    ...
    options = {
        "shared": [True, False],
        "option1": ["value1", "value2"],
        "option2": ["ANY"],
        "option3": [None, "value1", "value2"],
        "option4": [True, False, "value"],
 }
```

Once the recipe is loaded by Conan, the `options` are processed and they can be read in the recipe. You can also use the method `.get_safe()` (see *settings attribute*) to avoid Conan raising an Exception if the option doesn't exist:

```
class MyPkg(ConanFile):
    options = {"shared": [True, False]}

    def build(self):
        if self.options.shared:
            # build the shared library
        if self.options.get_safe("foo", True):
            pass
```

In boolean expressions, like `if self.options.shared`:

- equals `True` for the values `True`, `"True"` and `"true"`, and any other value that would be evaluated the same way in Python code.
- equals `False` for the values `False`, `"False"` and `"false"`, also for the empty string and for `0` and `"0"` as expected.

Notice that a comparison using `is` is always `False` because the types would be different as it is encapsulated inside a Python class.

See also:

- Read the *Getting started, creating packages* to know how to declare and how to define a value to an option.
- Removing options in the `package_id()` method. <MISSING PAGE>
- About the package_type and how it plays when a `shared` option is declared. <MISSING PAGE>

default_options

`ConanFile.`**`default_options = None`**

The attribute `default_options` defines the default values for the options, both for the current recipe and for any requirement. This attribute should be defined as a python dictionary.

```
class MyPkg(ConanFile):
    ...
    requires = "zlib/1.2.8", "zwave/2.0"
    options = {"build_tests": [True, False],
               "option2": "ANY"}
    default_options = {"build_tests": True,
                       "option1": 42,
                       "z*: shared": True}
```

You can also assign default values for options of your requirements using "<reference_pattern>: option_name", being a valid `reference_pattern` a `name/version` or any pattern with `*` like the example above.

You can also set the options conditionally to a final value with `configure()` instead of using `default_options`:

```
class OtherPkg(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    options = {"some_option": [True, False]}
    # Do NOT declare 'default_options', use 'config_options()'

    def configure(self):
        if self.options.some_option == None:
            if self.settings.os == 'Android':
                self.options.some_option = True
            else:
                self.options.some_option = False
```

Take into account that if a value is assigned in the `configure()` method it cannot be overridden.

See also:

Read more about the <MISSING PAGE>method_configure_config_options method.

options_description

TODO: Complete, https://github.com/conan-io/conan/pull/11295

requires

ConanFile.**requires = None**

List or tuple of strings for regular dependencies in the host context, like a library.

```
class MyLibConan(ConanFile):
    requires = "hello/1.0", "OtherLib/2.1@otheruser/testing"
```

You can specify version ranges, the syntax is using brackets:

```
class HelloConan(ConanFile):
    requires = "pkg/[>1.0 <1.8]"
```

Accepted expressions would be:

```
>1.1 <2.1     # In such range
2.8           # equivalent to =2.8
~=3.0         # compatible, according to semver
>1.1 || 0.8   # conditions can be OR'ed
```

See also:

- Check <MISSING PAGE> version_ranges if you want to learn more about version ranges.
- Check <MISSING PAGE> requires() conanfile.py method.

tool_requires

ConanFile.**tool_requires = None**

List or tuple of strings for dependencies. Represents a build tool like "cmake". If there is an existing pre-compiled binary for the current package, the binaries for the tool_require won't be retrieved. They cannot conflict.

```
class MyPkg(ConanFile):
    tool_requires = "tool_a/0.2", "tool_b/0.2@user/testing"
```

This is the declarative way to add `tool_requires`. Check the <MISSING PAGE> tool_requires() conanfile.py method to learn a more flexible way to add them.

build_requires

ConanFile.**build_requires = None**

List or tuple of strings for dependencies. Generic type of build dependencies that are not applications (nothing runs), like build scripts. If there is an existing pre-compiled binary for the current package, the binaries for the build_require won't be retrieved. They cannot conflict.

```
class MyPkg(ConanFile):
    build_requires = ["my_build_scripts/1.3",]
```

This is the declarative way to add `build_requires`. Check the <MISSING PAGE> build_requires() conanfile.py method to learn a more flexible way to add them.

test_requires

ConanFile.**test_requires = None**

List or tuple of strings for dependencies in the host context only. Represents a test tool like "gtest". Used when the current package is built from sources. They don't propagate information to the downstream consumers.

If there is an existing pre-compiled binary for the current package, the binaries for the test_require won't be retrieved. They cannot conflict.

```
class MyPkg(ConanFile):
    test_requires = "gtest/1.11.0", "other_test_tool/0.2@user/testing"
```

This is the declarative way to add `test_requires`. Check the <MISSING PAGE> test_requires() conanfile.py method to learn a more flexible way to add them.

exports

`ConanFile.`**`exports = None`**

List or tuple of strings with *file names* or fnmatch patterns that should be exported and stored side by side with the *conanfile.py* file to make the recipe work: other python files that the recipe will import, some text file with data to read,...

For example, if we have some python code that we want the recipe to use in a `helpers.py` file, and have some text file *info.txt* we want to read and display during the recipe evaluation we would do something like:

```
exports = "helpers.py", "info.txt"
```

Exclude patterns are also possible, with the `!` prefix:

```
exports = "*.py", "!*tmp.py"
```

See also:

- Check <MISSING PAGE> exports() conanfile.py method.

exports_sources

`ConanFile.`**`exports_sources = None`**

List or tuple of strings with file names or fnmatch patterns that should be exported and will be available to generate the package. Unlike the `exports` attribute, these files shouldn't be used by the `conanfile.py` Python code, but to compile the library or generate the final package. And, due to its purpose, these files will only be retrieved if requested binaries are not available or the user forces Conan to compile from sources.

This is an alternative to getting the sources with the `source()` method. Used when we are not packaging a third party library and we have together the recipe and the C/C++ project:

```
exports_sources = "include*", "src*"
```

Exclude patterns are also possible, with the `!` prefix:

```
exports_sources = "include*", "src*", "!src/build/*"
```

Note, if the recipe defines the `layout()` method and specifies a `self.folders.source = "src"` it won't affect where the files (from the `exports_sources`) are copied. They will be copied to the base source folder. So, if you want to replace some file that got into the `source()` method, you need to explicitly copy it from the parent folder or even better, from `self.export_sources_folder`.

```
import os, shutil
from conan import ConanFile
from conan.tools.files import save, load

class Pkg(ConanFile):
    ...
```

(continues on next page)

(continued from previous page)

```
    exports_sources = "CMakeLists.txt"

    def layout(self):
        self.folders.source = "src"
        self.folders.build = "build"

    def source(self):
        # emulate a download from web site
        save(self, "CMakeLists.txt", "MISTAKE: Very old CMakeLists to be replaced")
        # Now I fix it with one of the exported files
        shutil.copy("../CMakeLists.txt", ".")
        shutil.copy(os.path.join(self.export_sources_folder, "CMakeLists.txt", ".
↪"))
```

generators

`ConanFile.`**`generators = []`**

List or tuple of strings with names of generators.

```
class MyLibConan(ConanFile):
    generators = "CMakeDeps", "CMakeToolchain"
```

The generators can also be instantiated explicitly in the <MISSING PAGE> generate() method.

```
from conan.tools.cmake import CMakeToolchain

class MyLibConan(ConanFile):
    ...

    def generate(self):
        tc = CMakeToolchain(self)
        tc.generate()
```

Warning: Do not specify the same generator in the `generators` attribute and at the `generate()` method at the same recipe, Conan will use both and unexpected results might happen.

build_policy

`ConanFile.`**`build_policy = None`**

Controls when the current package is built during a `conan install`. The allowed values are:

- `"missing"`: Conan builds it from source if there is no binary available.
- `"always"`: Conan always builds it from source.
- `"never"`: This package cannot be built from sources, it is always created with `conan export-pkg`
- `None` (default value): This package won't be build unless the policy is specified in the command line (e.g `--build=foo*`)

```
class PocoTimerConan(ConanFile):
    build_policy = "always" # "missing"
```

no_copy_source

ConanFile.**no_copy_source = False**

The attribute `no_copy_source` tells the recipe that the source code will not be copied from the `source_folder` to the `build_folder`. This is mostly an optimization for packages with large source codebases or header-only, to avoid extra copies.

If you activate it (`no_copy_source=True`), is **mandatory** that the source code must not be modified at all by the configure or build scripts, as the source code will be shared among all builds.

The recipes should always use `self.source_folder` attribute, which will point to the `build` folder when `no_copy_source=False` and will point to the `source` folder when `no_copy_source=True`.

See also:

Read <MISSING PAGE> header-only section for an example using `no_copy_source` attribute.

source_folder

ConanFile.**source_folder**

The folder in which the source code lives. The path is built joining the base directory (a cache directory when running in the cache or the `output folder` when running locally) with the value of `folders.source` if declared in the `layout()` method.

Returns A string with the path to the source folder.

Note that the base directory for the `source_folder` when running in the cache will point to the base folder of the build unless *no_copy_source* is set to `True`. But anyway it will always point to the correct folder where the source code is.

export_sources_folder

ConanFile.**export_sources_folder**

The value depends on the method you access it:

- At `source(self)`: Points to the base source folder (that means self.source_folder but without taking into account the `folders.source` declared in the `layout()` method). The declared *exports_sources* are copied to that base source folder always.
- At `exports_sources(self)`: Points to the folder in the cache where the export sources have to be copied.

Returns A string with the mentioned path.

See also:

- Read <MISSING PAGE> `export_sources` method.
- Read <MISSING PAGE> `source` method.

build_folder

ConanFile.**build_folder**

The folder used to build the source code. The path is built joining the base directory (a cache directory when running in the cache or the `output folder` when running locally) with the value of `folders.build` if declared in the `layout()` method.

Returns A string with the path to the build folder.

package_folder

ConanFile.**package_folder**

The folder to copy the final artifacts for the binary package. In the local cache a package folder is created for every different package ID.

Returns A string with the path to the package folder.

The most common usage of `self.package_folder` is to `copy` the files at the <MISSING PAGE> package() method:

```
import os
from conan import ConanFile
from conan.tools.files import copy

class MyRecipe(ConanFile):
    ...

    def package(self):
        copy(self, "*.so", self.build_folder, os.path.join(self.package_folder, "lib
↪"))
        ...
```

recipe_folder

ConanFile.**recipe_folder = None**

The folder where the recipe *conanfile.py* is stored, either in the local folder or in the cache. This is useful in order to access files that are exported along with the recipe, or the origin folder when exporting files in `export(self)` and `export_sources(self)` methods.

The most common usage of `self.recipe_folder` is in the `export(self)` and `export_sources(self)` methods, as the folder from where we copy the files:

```
from conan import ConanFile
from conan.tools.files import copy

class MethodConan(ConanFile):
    exports = "file.txt"
    def export(self):
        copy(self, "LICENSE.md", self.recipe_folder, self.export_folder)
```

folders

<MISSING REFERENCE>

The `folders` attribute has to be set only in the `layout()` method.

- **self.folders.source**: To specify a folder where your sources are.
- **self.folders.build**: To specify a subfolder where the files from the build are (or will be).
- **self.folders.generators**: To specify a subfolder where to write the files from the generators and the toolchains (e.g. the *xx-config.cmake* files from the `CMakeDeps` generator).
- **self.folders.imports**: To specify a subfolder where to write the files copied when using the `imports(self)` method in a `conanfile.py`.
- **self.folders.root**: To specify the relative path from the `conanfile.py` to the root of the project, in case the `conanfile.py` is in a subfolder and not in the project root. If defined, all the other paths will be relative to the project root, not to the location of the `conanfile.py`.

- **self.folders.subproject**: To use together with `self.folders.root`, in case such a subproject needs to access some files that are located in a sibling folder.

See also:

Read more about the usage of the `layout()` in *this tutorial*.

cpp

`ConanFile.`**`cpp = None`**

Object storing all the information needed by the consumers of a package: include directories, library names, library paths... Both for editable and regular packages in the cache. It is only available at the `layout()` method.

- `self.cpp.package`: For a regular package being used from the Conan cache. Same as declaring `self.cpp_info` at the `package_info()` method.
- `self.cpp.source`: For "editable" packages, to describe the artifacts under `self.source_folder`
- `self.cpp.build`: For "editable" packages, to describe the artifacts under `self.build_folder`.

See also:

Read more about the *CppInfo* model.

Important: This attribute should be only filled in the <MISSING METHOD> `layout()` method.

---

cpp_info

`ConanFile.`**`cpp_info`**

Same as using `self.cpp.package` in the `layout()` method. Use it if you need to read the `package_folder` to locate the already located artifacts.

See also:

Read more about the *CppInfo* model.

Important: This attribute is only defined inside `package_info()` method being *None* elsewhere.

---

buildenv_info

`ConanFile.`**`buildenv_info = None`**

For the dependant recipes, the declared environment variables will be present during the build process. Should be only filled in the `package_info()` method.

Important: This attribute is only defined inside `package_info()` method being *None* elsewhere.

```
def package_info(self):
    self.buildenv_info.append_path("PATH", self.package_folder)
```

See also:

Check the reference of the *Environment* object to know how to fill the `self.buildenv_info`.

runenv_info

`ConanFile.runenv_info = None`

For the dependant recipes, the declared environment variables will be present at runtime. Should be only filled in the `package_info()` method.

Important: This attribute is only defined inside `package_info()` method being *None* elsewhere.

```
def package_info(self):
    self.runenv_info.define_path("RUNTIME_VAR", "c:/path/to/exe")
```

See also:

Check the reference of the *Environment* object to know how to fill the `self.runenv_info`.

conf_info

`ConanFile.conf_info = None`

Configuration variables to be passed to the dependant recipes. Should be only filled in the `package_info()` method.

```
class Pkg(ConanFile):
    name = "pkg"

    def package_info(self):
        self.conf_info.define("tools.microsoft.msbuild:verbosity", "Diagnostic")
        self.conf_info.get("tools.microsoft.msbuild:verbosity")  # == "Diagnostic"
        self.conf_info.append("user.myconf.build:ldflags", "--flag3")  # == ["--flag1
↪", "--flag2", "--flag3"]
        self.conf_info.update("tools.microsoft.msbuildtoolchain:compile_options", {
↪"ExpandAttributedSource": "false"})
        self.conf_info.unset("tools.microsoft.msbuildtoolchain:compile_options")
        self.conf_info.remove("user.myconf.build:ldflags", "--flag1")  # == ["--flag0
↪", "--flag2", "--flag3"]
        self.conf_info.pop("tools.system.package_manager:sudo")
```

See also:

Read here *the complete reference of self.conf_info*.

dependencies

Conan recipes provide access to their dependencies via the `self.dependencies` attribute.

```
class Pkg(ConanFile):
    requires = "openssl/0.1"

    def generate(self):
        openssl = self.dependencies["openssl"]
        # access to members
        openssl.ref.version
        openssl.ref.revision # recipe revision
        openssl.options
        openssl.settings
```

See also:

Read here *the complete reference of self.dependencies*.

conf

In the `self.conf` attribute we can find all the conf entries declared in the <MISSING PAGE> [conf] section of the profiles. in addition of the declared <MISSING PAGE> self.conf_info entries from the first level tool requirements. The profile entries have priority.

```
from conan import ConanFile

class MyConsumer(ConanFile):

  tool_requires = "my_android_ndk/1.0"

  def generate(self):
      # This is declared in the tool_requires
      self.output.info("NDK host: %s" % self.conf.get("tools.android:ndk_path"))
      # This is declared in the profile at [conf] section
      self.output.info("Custom var1: %s" % self.conf.get("user.custom.var1"))
```

info

Object used in:

- The <MISSING PAGE> `validate(self)` method to check if a current configuration of the package is correct or not:

```
def validate(self):
    if self.info.settings.os == "Windows":
        raise ConanInvalidConfiguration("Package does not work in Windows!
↪")
```

- The <MISSING PAGE> `package(self)` method to control the unique ID for a package:

```
def package_id(self):
    self.info.clear()
```

revision_mode

This attribute allow each recipe to declare how the revision for the recipe itself should be computed. It can take two different values:

- `"hash"` (by default): Conan will use the checksum hash of the recipe manifest to compute the revision for the recipe.
- `"scm"`: the commit ID will be used as the recipe revision if it belongs to a known repository system (Git or SVN). If there is no repository it will raise an error.

python_requires

This class attribute allows to define a dependency to another Conan recipe and reuse its code. Its basic syntax is:

```
from conan import ConanFile

class Pkg(ConanFile):
    python_requires = "pyreq/0.1@user/channel"  # recipe to reuse code from

    def build(self):
        self.python_requires["pyreq"].module # access to the whole conanfile.py module
        self.python_requires["pyreq"].module.myvar  # access to a variable
```

(continues on next page)

(continued from previous page)

```
        self.python_requires["pyreq"].module.myfunct()  # access to a global function
        self.python_requires["pyreq"].path # access to the folder where the reused␣
↪file is
```

Read more about this attribute in <MISSING PAGE>

python_requires_extend

This class attribute defines one or more classes that will be injected in runtime as base classes of the recipe class. Syntax for each of these classes should be a string like `pyreq.MyConanfileBase` where the `pyreq` is the name of a `python_requires` and `MyConanfileBase` is the name of the class to use.

```
from conan import ConanFile

class Pkg(ConanFile):
    python_requires = "pyreq/0.1@user/channel", "utils/0.1@user/channel"
    python_requires_extend = "pyreq.MyConanfileBase", "utils.UtilsBase"  # class/es␣
↪to inject
```

conan_data

Read only attribute with a dictionary with the keys and values provided in a <MISSING PAGE> conandata_yml file format placed next to the *conanfile.py*. This YAML file is automatically exported with the recipe and automatically loaded with it too.

You can declare information in the *conandata.yml* file and then access it inside any of the methods of the recipe. For example, a *conandata.yml* with information about sources that looks like this:

```
sources:
  "1.1.0":
    url: "https://www.url.org/source/mylib-1.0.0.tar.gz"
    sha256: "8c48baf3babe0d505d16cfc0cf272589c66d3624264098213db0fb00034728e9"
  "1.1.1":
    url: "https://www.url.org/source/mylib-1.0.1.tar.gz"
    sha256: "15b6393c20030aab02c8e2fe0243cb1d1d18062f6c095d67bca91871dc7f324a"
```

```
def source(self):
    tools.get(**self.conan_data["sources"][self.version])
```

deprecated

`ConanFile.`**`deprecated = None`**

This attribute declares that the recipe is deprecated, causing a user-friendly warning message to be emitted whenever it is used

For example, the following code:

```
from conan import ConanFile

class Pkg(ConanFile):
    name = "cpp-taskflow"
    version = "1.0"
    deprecated = True
```

may emit a warning like:

```
cpp-taskflow/1.0: WARN: Recipe 'cpp-taskflow/1.0' is deprecated. Please, consider␣
↪changing your requirements.
```

Optionally, the attribute may specify the name of the suggested replacement:

```
from conan import ConanFile

class Pkg(ConanFile):
    name = "cpp-taskflow"
    version = "1.0"
    deprecated = "taskflow"
```

This will emit a warning like:

```
cpp-taskflow/1.0: WARN: Recipe 'cpp-taskflow/1.0' is deprecated in favor of
↪'taskflow'. Please, consider changing your requirements.
```

If the value of the attribute evaluates to `False`, no warning is printed.

provides

`ConanFile.`**`provides = None`**

This attribute declares that the recipe provides the same functionality as other recipe(s). The attribute is usually needed if two or more libraries implement the same API to prevent link-time and run-time conflicts (ODR violations). One typical situation is forked libraries. Some examples are:

- LibreSSL, BoringSSL and OpenSSL
- libav and ffmpeg
- MariaDB client and MySQL client

If Conan encounters two or more libraries providing the same functionality within a single graph, it raises an error:

```
At least two recipes provides the same functionality:
 - 'libjpeg' provided by 'libjpeg/9d', 'libjpeg-turbo/2.0.5'
```

The attribute value should be a string with a recipe name or a tuple of such recipe names.

For example, to declare that `libjpeg-turbo` recipe offers the same functionality as `libjpeg` recipe, the following code could be used:

```
from conan import ConanFile

class LibJpegTurbo(ConanFile):
    name = "libjpeg-turbo"
    version = "1.0"
    provides = "libjpeg"
```

To declare that a recipe provides the functionality of several different recipes at the same time, the following code could be used:

```
from conan import ConanFile

class OpenBLAS(ConanFile):
    name = "openblas"
    version = "1.0"
    provides = "cblas", "lapack"
```

If the attribute is omitted, the value of the attribute is assumed to be equal to the current package name. Thus, it's redundant for `libjpeg` recipe to declare that it provides `libjpeg`, it's already implicitly assumed by Conan.

win_bash

`ConanFile.`**`win_bash = None`**

When `True` it enables the new run in a subsystem bash in Windows mechanism.

```
from conan import ConanFile

class FooRecipe(ConanFile):
    ...
    win_bash = True
```

It can also be declared as a `property` based on any condition:

```
from conan import ConanFile

class FooRecipe(ConanFile):
    ...


    @property
    def win_bash(self):
        return self.settings.arch == "armv8"
```

7.1.2 Other

There are other complex objects used in a `conanfile.py` that need to be declared or used only in some recipe methods:

Contents:

self.cpp and self.cpp_info

Properties to declare all the information needed by the consumers of a package: include directories, library names, library paths... Used both for editable packages and regular packages in the cache.

Usage

There are four instances available, only while running the following methods:

- **At `layout(self)` method:**
  - **self.cpp.package**: For a regular package being used from the Conan cache.
  - **self.cpp.source**: For "editable" packages, to describe the artifacts under `self.source_folder`.
  - **self.cpp.build**: For "editable" packages, to describe the artifacts under `self.build_folder`.

```
def layout(self):
    ...
    self.folders.source = "src"
    self.folders.build = "build"

    # In the local folder (before a conan create) the artifacts can␣
↪be found:
    self.cpp.source.includedirs = ["my_includes"]
```

(continues on next page)

(continued from previous page)

```
    self.cpp.build.libdirs = ["lib/x86_64"]
    self.cpp.build.libs = ["foo"]

    # In the Conan cache, we packaged everything at the default␣
↪standard directories, the library to link
    # is "foo"
    self.cpp.package.libs = ["foo"]
```

- **At `package_info(self)` method:**
  - **self.cpp_info**: Same as *self.cpp.package* but, at this point, the package contents are in `self.package_folder` so you can access the artifacts to fill the `self.cpp_info`, for example, using the *collect_libs()* tool.

```
def package_info(self):
    self.cpp_info.libs = ["foo"]
```

Attributes

NAME	DESCRIPTION
.includedirs	Ordered list with include paths. Defaulted to `["include"]`
.libdirs	Ordered list with lib paths. Defaulted to `["lib"]`
.resdirs	Ordered list of resource (data) paths. Defaulted to `["res"]`
.bindirs	Ordered list with paths to binaries (executables, dynamic libraries,...). Defaulted to `["bin"]`
.builddirs	Ordered list with build scripts directory paths. Defaulted to `[]` (empty) CMakeDeps generator will search in these dirs for files like *findXXX.cmake* or *include("XXX.cmake")*
.libs	Ordered list with the library names, Defaulted to `[]` (empty)
.defines	Preprocessor definitions. Defaulted to `[]` (empty)
.cflags	Ordered list with pure C flags. Defaulted to `[]` (empty)
.cxxflags	Ordered list with C++ flags. Defaulted to `[]` (empty)
.sharedlinkflags	Ordered list with linker flags (shared libs). Defaulted to `[]` (empty)
.exelinkflags	Ordered list with linker flags (executables). Defaulted to `[]` (empty)
.frameworks	Ordered list with the framework names (OSX), Defaulted to `[]` (empty)
.frameworkdirs	

Properties

Any CppInfo object can declare "properties" that can be read by the generators. The value of a property can be of any type. Check each generator reference to see the properties used on it.

```
def set_property(self, property_name, value)
def get_property(self, property_name):
```

Example:

```
def package_info(self):
    self.cpp_info.set_property("cmake_find_mode", "both")
```

Components

If your package is composed by more than one library, it is possible to declare components that allow to define a `CppInfo` object per each of those libraries and also requirements between them and to components of other packages (the following case is not a real example):

```
def package_info(self):
    self.cpp_info.components["crypto"].set_property("cmake_file_name", "Crypto")
    self.cpp_info.components["crypto"].libs = ["libcrypto"]
    self.cpp_info.components["crypto"].defines = ["DEFINE_CRYPTO=1"]
    self.cpp_info.components["crypto"].requires = ["zlib::zlib"]  # Depends on all␣
↪components in zlib package

    self.cpp_info.components["ssl"].set_property("cmake_file_name", "SSL")
    self.cpp_info.components["ssl"].includedirs = ["include/headers_ssl"]
    self.cpp_info.components["ssl"].libs = ["libssl"]
    self.cpp_info.components["ssl"].requires = ["crypto",
                                                "boost::headers"]  # Depends on␣
↪headers component in boost package

    obj_ext = "obj" if platform.system() == "Windows" else "o"
    self.cpp_info.components["ssl-objs"].objects = [os.path.join("lib", "ssl-object.{}
↪".format(obj_ext))]
```

Dependencies among components and to components of other requirements can be defined using the `requires` attribute and the name of the component. The dependency graph for components will be calculated and values will be aggregated in the correct order for each field.

self.conf_info

Allow to declare, remove and modify configurations that are passed to the dependant recipes.

`Conf.`**`define`**(*name*, *value*)

Define a value for the given configuration name.

Parameters

- **`name`** – Name of the configuration.
- **`value`** – Value of the configuration.

```
def package_info(self):
    # Setting values
    self.conf_info.define("tools.microsoft.msbuild:verbosity", "Diagnostic")
    self.conf_info.define("tools.system.package_manager:sudo", True)
```

(continues on next page)

(continued from previous page)

```
    self.conf_info.define("tools.microsoft.msbuild:max_cpu_count", 2)
    self.conf_info.define("user.myconf.build:ldflags", ["--flag1", "--flag2"])
    self.conf_info.define("tools.microsoft.msbuildtoolchain:compile_options", {
↪"ExceptionHandling": "Async"})
```

Conf.**get**(*conf_name*, *default=None*, *check_type=None*)

Get all the values of the given configuration name.

Parameters

- **conf_name** – Name of the configuration.
- **default** – Default value in case of conf does not have the conf_name key.
- **check_type** – Check the conf type(value) is the same as the given by this param. There are two default smart conversions for bool and str types.

```
def package_info(self):
    self.conf_info.get("tools.microsoft.msbuild:verbosity")  # == "Diagnostic"
    # Getting default values from configurations that don't exist yet
    self.conf_info.get("user.myotherconf.build:cxxflags", default=["--flag3"])  #␣
↪== ["--flag3"]
    # Getting values and ensuring the gotten type is the passed one otherwise an␣
↪exception will be raised
    self.conf_info.get("tools.system.package_manager:sudo", check_type=bool)  #␣
↪== True
    self.conf_info.get("tools.system.package_manager:sudo", check_type=int)  #␣
↪ERROR! It raises a ConanException
```

Conf.**pop**(*conf_name*, *default=None*)

Remove the given configuration, returning its value.

Parameters

- **conf_name** – Name of the configuration.
- **default** – Default value to return in case the configuration doesn't exist.

Returns

```
def package_info(self):
    value = self.conf_info.pop("tools.system.package_manager:sudo")
```

Conf.**append**(*name*, *value*)

Append a value to the given configuration name.

Parameters

- **name** – Name of the configuration.
- **value** – Value to append.

```
def package_info(self):
    # Modifying configuration list-like values
    self.conf_info.append("user.myconf.build:ldflags", "--flag3")  # == ["--flag1
↪", "--flag2", "--flag3"]
```

Conf.**prepend**(*name*, *value*)

Prepend a value to the given configuration name.

Parameters

- **name** – Name of the configuration.
- **value** – Value to prepend.

```
def package_info(self):
    self.conf_info.prepend("user.myconf.build:ldflags", "--flag0")  # == ["--flag0
↪", "--flag1", "--flag2", "--flag3"]
```

Conf.**update**(*name*, *value*)

Update the value to the given configuration name.

Parameters

- **name** – Name of the configuration.
- **value** – Value of the configuration.

```
def package_info(self):
    # Modifying configuration dict-like values
    self.conf_info.update("tools.microsoft.msbuildtoolchain:compile_options", {
↪"ExpandAttributedSource": "false"})
```

Conf.**remove**(*name*, *value*)

Remove a value from the given configuration name.

Parameters

- **name** – Name of the configuration.
- **value** – Value to remove.

```
def package_info(self):
    # Remove
    self.conf_info.remove("user.myconf.build:ldflags", "--flag1")  # == ["--flag0
↪", "--flag2", "--flag3"]
```

Conf.**unset**(*name*)

Clears the variable, equivalent to a unset or set XXX=

Parameters **name** – Name of the configuration.

```
def package_info(self):
    # Unset any value
    self.conf_info.unset("tools.microsoft.msbuildtoolchain:compile_options")
```

self.dependencies

Conan recipes provide access to their dependencies via the `self.dependencies` attribute. This attribute is generally used by generators like `CMakeDeps` or `MSBuildDeps` to generate the necessary files for the build.

This section documents the `self.dependencies` attribute, as it might be used by users both directly in recipe or indirectly to create custom build integrations and generators.

Dependencies interface

It is possible to access each one of the individual dependencies of the current recipe, with the following syntax:

```
class Pkg(ConanFile):
    requires = "openssl/0.1"

```

(continues on next page)

(continued from previous page)

```
    def generate(self):
        openssl = self.dependencies["openssl"]
        # access to members
        openssl.ref.version
        openssl.ref.revision # recipe revision
        openssl.options
        openssl.settings
```

Some **important** points:

- All the information is **read only**. Any attempt to modify dependencies information is an error and can raise at any time, even if it doesn't raise yet.
- It is not possible either to call any methods or any attempt to reuse code from the dependencies via this mechanism.
- This information does not exist in some recipe methods, only in those methods that evaluate after the full dependency graph has been computed. It will not exist in `configure()`, `config_options`, `export()`, `export_source()`, `set_name()`, `set_version()`, `requirements()`, `build_requirements()`, `system_requirements()`, `source()`, `init()`, `layout()`. Any attempt to use it in these methods can raise an error at any time.
- At the moment, this information should only be used in `generate()` and `validate()` methods. Any other use, please submit a Github issue.

Not all fields of the dependency conanfile are exposed, the current fields are:

- **package_folder**: The folder location of the dependency package binary
- **ref**: An object that contains `name`, `version`, `user`, `channel` and `revision` (recipe revision)
- **pref**: An object that contains `ref`, `package_id` and `revision` (package revision)
- **buildenv_info**: `Environment` object with the information of the environment necessary to build
- **runenv_info**: `Environment` object with the information of the environment necessary to run the app
- **cpp_info**: includedirs, libdirs, etc for the dependency.
- **settings**: The actual settings values of this dependency
- **settings_build**: The actual build settings values of this dependency
- **options**: The actual options values of this dependency
- **context**: The context (build, host) of this dependency
- **conf_info**: Configuration information of this dependency, intended to be applied to consumers.
- **dependencies**: The transitive dependencies of this dependency
- **is_build_context**: Return `True` if `context == "build"`.

Iterating dependencies

It is possible to iterate in a dict-like fashion all dependencies of a recipe. Take into account that `self.dependencies` contains all the current dependencies, both direct and transitive. Every upstream dependency of the current one that has some effect on it, will have an entry in this `self.dependencies`.

Iterating the dependencies can be done as:

```
requires = "zlib/1.2.11", "poco/1.9.4"

def generate(self):
    for require, dependency in self.dependencies.items():
        self.output.info("Dependency is direct={}: {}".format(require.direct,␣
↪dependency.ref))
```

will output:

```
conanfile.py (hello/0.1): Dependency is direct=True: zlib/1.2.11
conanfile.py (hello/0.1): Dependency is direct=True: poco/1.9.4
conanfile.py (hello/0.1): Dependency is direct=False: pcre/8.44
conanfile.py (hello/0.1): Dependency is direct=False: expat/2.4.1
conanfile.py (hello/0.1): Dependency is direct=False: sqlite3/3.35.5
conanfile.py (hello/0.1): Dependency is direct=False: openssl/1.1.1k
conanfile.py (hello/0.1): Dependency is direct=False: bzip2/1.0.8
```

Where the `require` dictionary key is a "requirement", and can contain specifiers of the relation between the current recipe and the dependency. At the moment they can be:

- `require.direct`: boolean, `True` if it is direct dependency or `False` if it is a transitive one.
- `require.build`: boolean, `True` if it is a `build_require` in the build context, as `cmake`.
- `require.test`: boolean, `True` if its a `build_require` in the host context (defined with `self.test_requires()`), as `gtest`.

The `dependency` dictionary value is the read-only object described above that access the dependency attributes.

The `self.dependencies` contains some helpers to filter based on some criteria:

- `self.dependencies.host`: Will filter out requires with `build=True`, leaving regular dependencies like `zlib` or `poco`.
- `self.dependencies.direct_host`: Will filter out requires with `build=True` or `direct=False`
- `self.dependencies.build`: Will filter out requires with `build=False`, leaving only `tool_requires` in the build context, as `cmake`.
- `self.dependencies.direct_build`: Will filter out requires with `build=False` or `direct=False`
- `self.dependencies.test`: Will filter out requires with `build=True` or with `test=False`, leaving only test requirements as `gtest` in the host context.

They can be used in the same way:

```
requires = "zlib/1.2.11", "poco/1.9.4"

def generate(self):
    cmake = self.dependencies.direct_build["cmake"]
    for require, dependency in self.dependencies.build.items():
        # do something, only build deps here
```

Dependencies `cpp_info` interface

The `cpp_info` interface is heavily used by build systems to access the data. This object defines global and per-component attributes to access information like the include folders:

```
def generate(self):
    cpp_info = self.dependencies["mydep"].cpp_info
    cpp_info.includedirs
    cpp_info.libdirs

    cpp_info.components["mycomp"].includedirs
    cpp_info.components["mycomp"].libdirs
```

All the paths declared in the `cppinfo` object (like `cpp_info.includedirs`) are absolute paths and works whether the dependency is in the cache or is an editable package.

See also:

Read more about the *CppInfo* model.

7.1.3 Methods

requirements()

Requirement traits

Traits are properties of a requires clause. They determine how various parts of a dependency are treated and propagated by Conan. Values for traits are usually computed by Conan based on dependency's *package_type*, but can also be specified manually.

A good introduction to traits is provided in the Advanced Dependencies Model in Conan 2.0 presentation.

In the example below `headers` and `libs` are traits.

```
self.requires("math/1.0", headers=True, libs=True)
```

headers

Indicates that there are headers that are going to be `#included` from this package at compile time. The dependency will be in the host context.

libs

The dependency contains some library or artifact that will be used at link time of the consumer. This trait will typically be `True` for direct shared and static libraries, but could be false for indirect static libraries that are consumed via a shared library. The dependency will be in the host context.

build

This dependency is a build tool, an application or executable, like cmake, that is used exclusively at build time. It is not linked/embedded into binaries, and will be in the build context.

run

This dependency contains some executables, either apps or shared libraries that need to be available to execute (typically in the path, or other system env-vars). This trait can be `True` for `build=False`, in that case, the package will contain some executables that can run in the host system when installing it, typically like an end-user application. This trait can be `True` for `build=True`, the package will contain executables that will run in the build context, typically while being used to build other packages.

visible

This `require` will be propagated downstream, even if it doesn't propagate `headers`, `libs` or `run` traits. Requirements that propagate downstream can cause version conflicts. This is typically `True`, because in most cases, having 2 different versions of the same library in the same dependency graph is at least complicated, if not directly violating ODR or causing linking errors. It can be set to `False` in advanced scenarios, when we want to use different versions of the same package during the build.

transitive_headers

If `True` the headers of the dependency will be visible downstream.

transitive_libs

If `True` the libraries to link with of the dependency will be visible downstream.

test

This requirement is a test library or framework, like Catch2 or gtest. It is mostly a library that needs to be included and linked, but that will not be propagated downstream.

package_id_mode

If the recipe wants to specify how the dependency version affects the current package `package_id`, can be directly specified here.

While it could be also done in the `package_id()` method, it seems simpler to be able to specify it in the `requires` while avoiding some ambiguities.

```
# We set the package_id_mode so it is part of the package_id
self.tool_requires("tool/1.1.1", package_id_mode="minor_mode")
```

Which would be equivalent to:

```
def package_id(self):
  self.info.requires["tool"].minor_mode()
```

force

This `requires` will force its version in the dependency graph upstream, overriding other existing versions even of transitive dependencies, and also solving potential existing conflicts.

override

The same as the `force` trait, but not adding a `direct` dependency. If there is no transitive dependency to override, this `require` will be discarded. This trait only exists at the time of defining a `requires`, but it will not exist as an actual `requires` once the graph is fully evaluated

direct

If the dependency is a direct one, that is, it has explicitly been declared by the current recipe, or if it is a transitive one.

validate_build()

The `validate_build()` method is used to verify if a configuration is valid for building a package. It is different from the `validate()` method that checks if the binary package is "impossible" or invalid for a given configuration.

The `validate()` method should do the checks of the settings and options using the `self.info.settings` and `self.info.options`.

The `validate_build()` method has to use always the `self.settings` and `self.options`:

```
from conan import ConanFile
from conan.errors import ConanInvalidConfiguration
class myConan(ConanFile):
    name = "foo"
    version = "1.0"
    settings = "os", "arch", "compiler"
    def package_id(self):
        # For this package, it doesn't matter the compiler used for the binary package
        del self.info.settings.compiler
    def validate_build(self):
        # But we know this cannot be build with "gcc"
        if self.settings.compiler == "gcc":
            raise ConanInvalidConfiguration("This doesn't build in GCC")
    def validate(self):
        # We shouldn't check here the self.info.settings.compiler because it has been␣
↪removed in the package_id()
        # so it doesn't make sense to check if the binary is compatible with gcc␣
↪because the compiler doesn't matter
        pass
```

7.2 Conan commands

7.2.1 conan search

Search existing recipes in remotes. This command is equivalent to `conan list recipes <query> -r=*`, and is provided for simpler UX.

```
conan search -h
usage: conan search [-h] [-f {cli,json}] [-r REMOTE] query

Searches for package recipes in a remote or remotes

positional arguments:
query                 Search query to find package recipe reference, e.g., 'boost',
↪'lib*'

optional arguments:
-h, --help            show this help message and exit
-f {cli,json}, --format {cli,json}
                      Select the output format: cli, json. 'cli' is the default␣
↪output.
-r REMOTE, --remote REMOTE
                      Remote names. Accepts wildcards. If not specified it searches␣
↪in all remotes
```

```
$ conan search zlib
conancenter:
```

(continues on next page)

(continued from previous page)

```
zlib
    zlib/1.2.11
    zlib/1.2.8

$ conan search zlib -r=conancenter
conancenter:
zlib
    zlib/1.2.11
    zlib/1.2.8

$ conan search zlib/1.2.1* -r=conancenter
conancenter:
zlib
    zlib/1.2.11

$ conan search zlib/1.2.1* -r=conancenter --format=json
[
    {
        "remote": "conancenter",
        "error": null,
        "results": [
            {
                "name": "zlib",
                "id": "zlib/1.2.11"
            }
        ]
    }
]
```

7.2.2 conan list

conan list recipes

```
$ conan list recipes zlib -r=conancenter
conancenter:
zlib
    zlib/1.2.11
    zlib/1.2.8

$ conan list recipes zlib/1.2.1* -r=conancenter
conancenter:
zlib
    zlib/1.2.11

$ conan list recipes zlib/1.2.1* -r=conancenter --format=json
[
    {
        "remote": "conancenter",
        "error": null,
        "results": [
            {
                "name": "zlib",
                "id": "zlib/1.2.11"
            }
        ]
    }
```

(continues on next page)

(continued from previous page)

```
]
```

conan list package-ids

```
$ conan list package-ids zlib/1.2.11 -r=conancenter
...
zlib/1.2.11:1513b3452ef7e2a2dd5f931247c5e02edeb98cc9
    settings:
    os=Macos
    arch=x86_64
    compiler=apple-clang
    build_type=Debug
    compiler.version=10.0
    options:
    shared=False
    fPIC=True
zlib/1.2.11:963bb116781855de98dbb23aaac41621e5d312d8
    settings:
    os=Windows
    compiler.runtime=MTd
    arch=x86_64
    compiler=Visual Studio
    build_type=Debug
    compiler.version=15
    options:
    shared=False
zlib/1.2.11:bf6871a88a66b609883bce5de4dd61adb1e033a7
    settings:
    os=Linux
    arch=x86_64
    compiler=gcc
    build_type=Debug
    compiler.version=5
    options:
    shared=True
...
```

conan list recipe-revisions

```
$ conan list recipe-revisions zlib/1.2.11 -r=conancenter
conancenter:
...
  zlib/1.2.11#b3eaf63da20a8606f3d84602c2cfa854 (2021-08-27T20:02:46Z)
  zlib/1.2.11#08c5163c8e302d1482d8fa2be93736af (2021-05-05T16:17:39Z)
  zlib/1.2.11#b291478a29f383b998e1633bee1c0536 (2021-03-25T10:03:21Z)
  zlib/1.2.11#514b772abf9c36ad9be48b84cfc6fdc2 (2021-02-19T14:33:26Z)
```

conan list package-revisions

```
$conan list package-revisions zlib/1.2.11
↪#b3eaf63da20a8606f3d84602c2cfa854:963bb116781855de98dbb23aaac41621e5d312d8 -
↪r=conancenter
conancenter:
  zlib/1.2.11
↪#b3eaf63da20a8606f3d84602c2cfa854:963bb116781855de98dbb23aaac41621e5d312d8
↪#dd44f4a86108e836f0c2d35af89cd8cd (2021-08-27T20:12:00Z)
```

(continues on next page)

(continued from previous page)

7.2.3 Creator commands

7.2.4 Custom commands

Custom commands

It's possible to create your own Conan commands to solve self-needs thanks to Python and Conan public API powers altogether.

Location and naming

All the custom commands must be located in `[YOUR_CONAN_HOME]/extensions/commands/` folder. If you don't know where `[YOUR_CONAN_HOME]` is located, you can run **`conan config home`** to check it.

If _commands_ sub-directory is not created yet, you will have to create it. Those custom commands files must be Python files and start with the prefix `cmd_[your_command_name].py`. The call to the custom commands is like any other existing Conan one: **`conan your_command_name`**.

Scoping

It's possible to have another folder layer to group some commands under the same topic.

For instance:

```
| - [YOUR_CONAN_HOME]/extensions/commands/greet/
      | - cmd_hello.py
      | - cmd_bye.py
```

The call to those commands change a little bit: **`conan [topic_name]:your_command_name`**. Following the previous example:

```
$ conan greet:hello
$ conan greet:bye
```

Note: It's possible for only one folder layer, so it won't work to have something like `[YOUR_CONAN_HOME]/extensions/commands/topic1/topic2/cmd_command.py`

Decorators

conan_command(group=None, formatters=None)

Main decorator to declare a function as a new Conan command. Where the parameters are:

- `group` is the name of the group of commands declared under the same name. This grouping will appear executing the **`conan -h`** command.
- `formatters` is a dict-like Python object where the `key` is the formatter name and the value is the function instance where will be processed the information returned by the command one.

Listing 1: cmd_hello.py

```
import json

from conan.api.conan_api import ConanAPIV2
from conan.api.output import ConanOutput
from conan.cli.command import conan_command

def output_json(msg):
    return json.dumps({"greet": msg})


@conan_command(group="Custom commands", formatters={"json": output_json})
def hello(conan_api: ConanAPIV2, parser, *args):
    """
    Simple command to print "Hello World!" line
    """
    msg = "Hello World!"
    ConanOutput().info(msg)
    return msg
```

Important: The function decorated by `@conan_command(....)` must have the same name as the suffix used by the Python file. For instance, the previous example, the file name is `cmd_hello.py`, and the command function decorated is `def hello(....)`.

conan_subcommand(formatters=None)

Similar to `conan_command`, but this one is declaring a sub-command of an existing custom command. For instance:

Listing 2: cmd_hello.py

```
from conan.api.conan_api import ConanAPIV2
from conan.api.output import ConanOutput
from conan.cli.command import conan_command, conan_subcommand


@conan_subcommand()
def hello_moon(conan_api, parser, subparser, *args):
    """
    Sub-command of "hello" that prints "Hello Moon!" line
    """
    ConanOutput().info("Hello Moon!")


@conan_command(group="Custom commands")
def hello(conan_api: ConanAPIV2, parser, *args):
    """
    Simple command "hello"
    """
```

The command call looks like **`conan hello moon`**.

Note: Notice that to declare a sub-command is required an empty Python function acts as the main command.

Command function arguments

These are the passed arguments to any custom command and its sub-commands functions:

Listing 3: cmd_command.py

```
from conan.cli.command import conan_command, conan_subcommand


@conan_subcommand()
def command_subcommand(conan_api, parser, subparser, *args):
    """
    subcommand information. This info will appear on ``conan command subcommand -h``.

    :param conan_api: <object conan.api.conan_api.ConanAPIV2> instance
    :param parser: root <object argparse.ArgumentParser> instance (coming from main␣
↪command)
    :param subparser: <object argparse.ArgumentParser> instance for sub-command
    :param args: ``list`` of all the arguments passed after sub-command call
    :return: (optional) whatever is returned will be passed to formatters functions␣
↪(if declared)
    """
    # ...


@conan_command(group="Custom commands")
def command(conan_api, parser, *args):
    """
    command information. This info will appear on ``conan command -h``.

    :param conan_api: <object conan.api.conan_api.ConanAPIV2> instance
    :param parser: root <object argparse.ArgumentParser> instance
    :param args: ``list`` of all the arguments passed after command call
    :return: (optional) whatever is returned will be passed to formatters functions␣
↪(if declared)
    """
    # ...
```

- `conan_api`: instance of `ConanAPIV2` class. See more about it in *conan.api.conan_api.ConanAPIV2 section*
- `parser`: root instance of Python `argparse.ArgumentParser` class to be used by the main command function. See more information in argparse official website.
- `subparser` (only for sub-commands): child instance of Python `argparse.ArgumentParser` class for each sub-command function.
- `*args`: list of all the arguments passed via command line to be parsed and used inside the command function. Normally, they'll be parsed as `args = parser.parse_args(*args)`. For instance, running **`conan mycommand arg1 arg2 arg3`**, the command function will receive them as a Python list-like `["arg1", "arg2", "arg3"]`.

Read more

- *Custom command to remove recipe and package revisions but the latest package one from the latest recipe one*.

7.3 Python API

7.3.1 Conan API Reference

orphan

Warning: The Conan API is experimental and subject to breaking changes.

class ConanAPIV2(*cache_folder=None*)

Read more

- Creating Conan custom commands
- ...

7.3.2 Remotes API

orphan

Warning: The Conan API is experimental and subject to breaking changes.

class RemotesAPI(*conan_api*)

7.3.3 Search API

orphan

Warning: The Conan API is experimental and subject to breaking changes.

class SearchAPI(*conan_api*)

recipe_revisions(*expression*, *remote=None*, *none_revision_allowed=True*)

Parameters

- **expression** – A RecipeReference that can contain "*" at any field
- **remote** – Remote in case we want to check the references in a remote

Returns a list of complete RecipeReference

package_revisions(*expression*, *query=None*, *remote=None*)

Resolve an expression like lib*/1*#*:9283*, filtering by query and obtaining all the package revisions

Parameters

- **expression** – lib*/1*#*:9283*
- **query** – package configuration query like "os=Windows AND (arch=x86 OR compiler=gcc)"
- **remote** – Remote object

Returns a List of PkgReference

7.3.4 List API

orphan

Warning: The Conan API is experimental and subject to breaking changes.

class ListAPI(*conan_api*)

Get references from the recipes and packages in the cache or a remote

filter_packages_configurations(*pkg_configurations*, *query*)

Parameters

- **pkg_configurations** – Dict[PkgReference, PkgConfiguration]
- **query** – str like "os=Windows AND (arch=x86 OR compiler=gcc)"

Returns Dict[PkgReference, PkgConfiguration]

7.3.5 Profiles API

orphan

Warning: The Conan API is experimental and subject to breaking changes.

class ProfilesAPI(*conan_api*)

get_default_host()

Returns the path to the default "host" profile, either in the cache or as defined by the user in configuration

get_default_build()

Returns the path to the default "build" profile, either in the cache or as defined by the user in configuration

get_profile(*profiles*, *settings=None*, *options=None*, *conf=None*, *cwd=None*)

Computes a Profile as the result of aggregating all the user arguments, first it loads the "profiles", composing them in order (last profile has priority), and finally adding the individual settings, options (priority over the profiles)

get_path(*profile*, *cwd=None*, *exists=True*)

Returns the resolved path of the given profile name, that could be in the cache, or local, depending on the "cwd"

list()

List all the profiles file sin the cache :return: an alphabetically ordered list of profile files in the default cache location

detect()

Returns an automatically detected Profile, with a "best guess" of the system settings

7.3.6 Install API

orphan

Warning: The Conan API is experimental and subject to breaking changes.

class InstallAPI(*conan_api*)

install_binaries(*deps_graph*, *remotes=None*, *update=False*)
Install binaries for dependency graph :param deps_graph: Dependency graph to intall packages for :param remotes: :param update:

install_consumer(*deps_graph*, *generators=None*, *source_folder=None*, *output_folder=None*, *deploy=False*)
Once a dependency graph has been installed, there are things to be done, like invoking generators for the root consumer. This is necessary for example for conanfile.txt/py, or for "conan install <ref> -g

7.3.7 Graph API

orphan

Warning: The Conan API is experimental and subject to breaking changes.

class GraphAPI(*conan_api*)

load_root_test_conanfile(*path*, *tested_reference*, *profile_host*, *profile_build*, *update=None*, *remotes=None*, *lockfile=None*)
Create and initialize a root node from a test_package/conanfile.py consumer

Parameters

- **lockfile** – Might be good to lock python-requires, build-requires
- **path** – The full path to the test_package/conanfile.py being used
- **tested_reference** – The full RecipeReference of the tested package
- **profile_host** –
- **profile_build** –
- **update** –
- **remotes** –

Returns a graph Node, recipe=RECIPE_CONSUMER

load_graph(*root_node*, *profile_host*, *profile_build*, *lockfile=None*, *remotes=None*, *update=False*, *check_update=False*)
Compute the dependency graph, starting from a root package, evaluation the graph with the provided configuration in profile_build, and profile_host. The resulting graph is a graph of recipes, but packages are not computed yet (package_ids) will be empty in the result. The result might have errors, like version or configuration conflicts, but it is still possible to inspect it. Only trying to install such graph will fail

Parameters

- **root_node** – the starting point, an already initialized Node structure, as returned by the "load_root_node" api

- **profile_host** – The host profile
- **profile_build** – The build profile
- **lockfile** – A valid lockfile (None by default, means no locked)
- **remotes** – list of remotes we want to check
- **update** – (False by default), if Conan should look for newer versions or revisions for already existing recipes in the Conan cache
- **check_update** – For "graph info" command, check if there are recipe updates

analyze_binaries(*graph*, *build_mode=None*, *remotes=None*, *update=None*, *lockfile=None*)

Given a dependency graph, will compute the package_ids of all recipes in the graph, and evaluate if they should be built from sources, downloaded from a remote server, of if the packages are already in the local Conan cache

Parameters

- **lockfile** –
- **graph** – a Conan dependency graph, as returned by "load_graph()"
- **build_mode** – TODO: Discuss if this should be a BuildMode object or list of arguments
- **remotes** – list of remotes
- **update** – (False by default), if Conan should look for newer versions or revisions for already existing recipes in the Conan cache

load_conanfile_class(*path*)

Given a path to a conanfile.py file, it loads its class (not instance) to allow inspecting the class attributes, like 'name', 'version', 'description', 'options' etc

7.3.8 Export API

orphan

> **Warning:** The Conan API is experimental and subject to breaking changes.

class ExportAPI(*conan_api*)

7.3.9 Remove API

orphan

> **Warning:** The Conan API is experimental and subject to breaking changes.

class RemoveAPI(*conan_api*)

7.3.10 Config API

orphan

> **Warning:** The Conan API is experimental and subject to breaking changes.

class ConfigAPI(*conan_api*)

7.3.11 New API

orphan

> **Warning:** The Conan API is experimental and subject to breaking changes.

class NewAPI(*conan_api*)

get_template(*template_folder*)
Load a template from a user absolute folder

get_home_template(*template_name*)
Load a template from the Conan home templates/command/new folder

7.3.12 Upload API

orphan

> **Warning:** The Conan API is experimental and subject to breaking changes.

class UploadAPI(*conan_api*)

check_integrity(*upload_data*)
Check if the recipes and packages are corrupted (it will raise a ConanExcepcion)

check_upstream(*upload_bundle*, *remote*, *force=False*)
Check if the artifacts are already in the specified remote, skipping them from the upload_bundle in that case

prepare(*upload_bundle*)
Compress the recipes and packages and fill the upload_data objects with the complete information. It doesn't perform the upload nor checks upstream to see if the recipe is still there

7.3.13 Download API

orphan

> **Warning:** The Conan API is experimental and subject to breaking changes.

class DownloadAPI(*conan_api*)

7.4 tools

Tools are all things that can be imported and used in Conan recipes.

The import path is always like:

```
from conan.tools.cmake import CMakeToolchain, CMakeDeps, CMake
form conan.tools.microsoft import MSBuildToolchain, MSBuildDeps, MSBuild
```

The main guidelines are:

- Everything that recipes can import belong to `from conan.tools`. Any other thing is private implementation and shouldn't be used in recipes.
- Only documented, public (not preceded by _) tools can be used in recipes.

Contents:

7.4.1 conan.tools.cmake

CMakeDeps

The `CMakeDeps` generator produces the necessary files for each dependency to be able to use the cmake `find_package()` function to locate the dependencies. It can be used like:

```
from conan import ConanFile

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    generators = "CMakeDeps"
```

The full instantiation, that allows custom configuration can be done in the `generate()` method:

```
from conan import ConanFile
from conan.tools.cmake import CMakeDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"

    def generate(self):
        cmake = CMakeDeps(self)
        cmake.generate()
```

Listing 4: **CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.15)
project(compressor C)

find_package(hello REQUIRED)

add_executable(${PROJECT_NAME} src/main.c)
target_link_libraries(${PROJECT_NAME} hello::hello)
```

By default, for a `hello` requires, you need to use `find_package(hello)` and link with the target `hello::hello`. Check *the properties affecting CMakeDeps* like `cmake_target_name` to customize the file and the target names in the conanfile.py of the dependencies and their components.

Note: The `CMakeDeps` is intended to run with the `CMakeToolchain` generator. It will set `CMAKE_PREFIX_PATH` and `CMAKE_MODULE_PATH` to the right folder (`conanfile.generators_folder`) so CMake can locate the generated config/module files.

Generated files

- **XXX-config.cmake**: By default, the CMakeDeps generator will create config files declaring the targets for the dependencies and their components (if declared).
- **FindXXX.cmake**: Only when the property `cmake_find_mode` is set by the dependency with "module" or "both". See *The properties affecting CMakeDeps* is set in the dependency.
- **Other necessary *.cmake**: files like version, flags and directory data or configuration.

Customization

There are some attributes you can adjust in the created `CMakeDeps` object to change the default behavior:

configuration

Allows to define custom user CMake configuration besides the standard Release, Debug, etc ones.

```
def generate(self):
    deps = CMakeDeps(self)
    # By default, ``deps.configuration`` will be ``self.settings.build_type``
    if self.options["hello"].shared:
        # Assuming the current project ``CMakeLists.txt`` defines the ReleasedShared␣
↪configuration.
        deps.configuration = "ReleaseShared"
    deps.generate()
```

The `CMakeDeps` is a *multi-configuration* generator, it can correctly create files for Release/Debug configurations to be simultaneously used by IDEs like Visual Studio. In single configuration environments, it is necessary to have a configuration defined, which must be provided via the `cmake ... -DCMAKE_BUILD_TYPE=<build-type>` argument in command line (Conan will do it automatically when necessary, in the `CMake.configure()` helper).

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()
```

build_context_suffix

When you have the same package as a **build-require** and as a **regular require** it will cause a conflict in the generator because the file names of the config files will collide as well as the targets names, variables names etc.

For example, this is a typical situation with some requirements (capnproto, protobuf...) that contain a tool used to generate source code at build time (so it is a **build_require**), but also providing a library to link to the final application, so you also have a **regular require**. Solving this conflict is specially important when we are cross-building because the tool (that will run in the building machine) belongs to a different binary package than the library, that will "run" in the host machine.

You can use the **build_context_suffix** attribute to specify a suffix for a requirement, so the files/targets/variables of the requirement in the build context (tool require) will be renamed:

```
tool_requires = ["my_tool/0.0.1"]
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"]
    # disambiguate the files, targets, etc
    cmake.build_context_suffix = {"my_tool": "_BUILD"}
    cmake.generate()
```

build_context_build_modules

Also there is another issue with the **build_modules**. As you may know, the recipes of the requirements can declare a *cppinfo.build_modules* entry containing one or more **.cmake** files. When the requirement is found by the cmake `find_package()` function, Conan will include automatically these files.

By default, Conan will include only the build modules from the `host` context (regular requires) to avoid the collision, but you can change the default behavior.

Use the **build_context_build_modules** attribute to specify require names to include the **build_modules** from **tool_requires**:

```
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"]
    # Choose the build modules from "build" context
    cmake.build_context_build_modules = ["my_tool"]
    cmake.generate()
```

Reference

class CMakeDeps(*conanfile*)

generate()
This method will save the generated files to the conanfile.generators_folder

Properties

The following properties affect the CMakeDeps generator:

- **cmake_file_name**: The config file generated for the current package will follow the `<VALUE>-config.cmake` pattern, so to find the package you write `find_package(<VALUE>)`.
- **cmake_target_name**: Name of the target to be consumed.
- **cmake_target_aliases**: List of aliases that Conan will create for an already existing target.
- **cmake_find_mode**: Defaulted to `config`. Possible values are:
  - `config`: The CMakeDeps generator will create config scripts for the dependency.
  - `module`: Will create module config (FindXXX.cmake) scripts for the dependency.
  - `both`: Will generate both config and modules.

  - `none`: Won't generate any file. It can be used, for instance, to create a system wrapper package so the consumers find the config files in the CMake installation config path and not in the generated by Conan (because it has been skipped).

- **cmake_module_file_name**: Same as **cmake_file_name** but when generating modules with `cmake_find_mode=module/both`. If not specified it will default to **cmake_file_name**.
- **cmake_module_target_name**: Same as **cmake_target_name** but when generating modules with `cmake_find_mode=module/both`. If not specified it will default to **cmake_target_name**.
- **cmake_build_modules**: List of `.cmake` files (route relative to root package folder) that are automatically included when the consumer run the `find_package()`. This property cannot be set in the components, only in the root `self.cpp_info`.
- **cmake_set_interface_link_directories**: boolean value that should be only used by dependencies that don't declare *self.cpp_info.libs* but have `#pragma comment(lib, "foo")` (automatic link) declared at the public headers. Those dependencies should add this property to their *conanfile.py* files at root `cpp_info` level (components not supported for now).
- **nosoname**: boolean value that should be used only by dependencies that are defined as `SHARED` and represent a library built without the `soname` flag option.

Example:

```
def package_info(self):
    ...
    # MyFileName-config.cmake
    self.cpp_info.set_property("cmake_file_name", "MyFileName")
    # Names for targets are absolute, Conan won't add any namespace to the target␣
↪names automatically
    self.cpp_info.set_property("cmake_target_name", "Foo::Foo")
    # Automatically include the lib/mypkg.cmake file when calling find_package()
    # This property cannot be set in a component.
    self.cpp_info.set_property("cmake_build_modules", [os.path.join("lib", "mypkg.
↪cmake")])

    # Create a new target "MyFooAlias" that is an alias to the "Foo::Foo" target
    self.cpp_info.set_property("cmake_target_aliases", ["MyFooAlias"])

    self.cpp_info.components["mycomponent"].set_property("cmake_target_name",
↪"Foo::Var")

    # Create a new target "VarComponent" that is an alias to the "Foo::Var" component␣
↪target
    self.cpp_info.components["mycomponent"].set_property("cmake_target_aliases", [
↪"VarComponent"])

    # Skip this package when generating the files for the whole dependency tree in␣
↪the consumer
    # note: it will make useless the previous adjustements.
    # self.cpp_info.set_property("cmake_find_mode", "none")

    # Generate both MyFileNameConfig.cmake and FindMyFileName.cmake
    self.cpp_info.set_property("cmake_find_mode", "both")
```

CMakeToolchain

The `CMakeToolchain` is the toolchain generator for CMake. It produces the toolchain file that can be used in the command line invocation of CMake with the `-DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake`. This generator translates the current package configuration, settings, and options, into CMake toolchain syntax.

It can be declared as:

```
from conan import ConanFile

class Pkg(ConanFile):
    generators = "CMakeToolchain"
```

Or fully instantiated in the `generate()` method:

```
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    generators = "CMakeDeps"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}

    def generate(self):
        tc = CMakeToolchain(self)
        tc.variables["MYVAR"] = "MYVAR_VALUE"
        tc.preprocessor_definitions["MYDEFINE"] = "MYDEF_VALUE"
        tc.generate()
```

Note: The `CMakeToolchain` is intended to run with the `CMakeDeps` dependencies generator. Please do not use other CMake legacy generators (like `cmake`, or `cmake_paths`) with it.

Generated files

This will generate the following files after a `conan install` (or when building the package in the cache) with the information provided in the `generate()` method as well as information translated from the current `settings`:

- **conan_toolchain.cmake**: containing the translation of Conan settings to CMake variables. Some things that will be defined in this file:
  - Definition of the CMake generator platform and generator toolset
  - Definition of the `CMAKE_POSITION_INDEPENDENT_CODE`, based on `fPIC` option.
  - Definition of the C++ standard as necessary
  - Definition of the standard library used for C++
  - Deactivation of rpaths in OSX
- **conanvcvars.bat**: In some cases, the Visual Studio environment needs to be defined correctly for building, like when using the Ninja or NMake generators. If necessary, the `CMakeToolchain` will generate this script, so defining the correct Visual Studio prompt is easier.
- **CMakePresets.json**: The toolchain also generates a `CMakePresets.json` standard file, check the documentation here. It is currently using the version "3" of the JSON schema. Conan creates a `default` configure preset with the information:
  - The `generator` to be used.
  - The path to the `conan_toolchain.cmake`.
  - Some cache variables corresponding to the specified settings cannot work if specified in the toolchain.

  - The `CMAKE_BUILD_TYPE` variable when using a single-configuration generators.
  - If you run several `conan install` with different `-s build_type` values, it will generate the corresponding `buildPresets` and `configurePresets`.

- **CMakeUserPresets.json**: If you declare a `layout()` in the recipe and your `CMakeLists.txt` file is found at the `conanfile.source_folder` folder, a `CMakeUserPresets.json` file will be generated (if doesn't exist already) including automatically the `CMakePresets.json` (at the `conanfile.generators_folder`) to allow your IDE (Visual Studio, Visual Studio Code, CLion...) or `cmake` tool to locate the `CMakePresets.json`. The version schema of the generated `CMakeUserPresets.json` is "4" and requires CMake >= 3.23.

  Note: Conan will skip the generation of the `CMakeUserPresets.json` if it already exists and was not generated by Conan.

By default, the version schema of the generated `CMakeUserPresets.json` is 4 and the schema for the `CMakePresets.json` is 3, so they require CMake >= 3.23. You can control the version of the generated `CMakePresets.json` and `CMakeUserPresets.json` with a *configuration* `tools.cmake.cmaketoolchain.presets:max_schema_version`.

It can be set in the *global.conf*, with *-c* in the **`conan install`** command, or in a profile. The minimum accepted value for this conf is `2`:

```
conan install . -c tools.cmake.cmaketoolchain.presets:max_schema_version=2
```

Customization

preprocessor_definitions

This attribute allows defining compiler preprocessor definitions, for multiple configurations (Debug, Release, etc).

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.preprocessor_definitions["MYDEF"] = "MyValue"
    tc.preprocessor_definitions.debug["MYCONFIGDEF"] = "MyDebugValue"
    tc.preprocessor_definitions.release["MYCONFIGDEF"] = "MyReleaseValue"
    tc.generate()
```

This will be translated to:

- One `add_definitions()` definition for `MYDEF` in `conan_toolchain.cmake` file.
- One `add_definitions()` definition, using a cmake generator expression in `conan_toolchain.cmake` file, using the different values for different configurations.

variables

This attribute allows defining CMake variables, for multiple configurations (Debug, Release, etc).

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.variables["MYVAR"] = "MyValue"
    tc.variables.debug["MYCONFIGVAR"] = "MyDebugValue"
    tc.variables.release["MYCONFIGVAR"] = "MyReleaseValue"
    tc.generate()
```

This will be translated to:

- One `set()` definition for `MYVAR` in `conan_toolchain.cmake` file.

- One `set()` definition, using a cmake generator expression in `conan_toolchain.cmake` file, using the different values for different configurations.

The booleans assigned to a variable will be translated to `ON` and `OFF` symbols in CMake:

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.variables["FOO"] = True
    tc.variables["VAR"] = False
    tc.generate()
```

Will generate the sentences: `set(FOO ON ...)` and `set(VAR OFF ...)`.

cache_variables

This attribute allows defining CMake cache-variables. These variables, unlike the `variables`, are single-config. They will be stored in the `CMakePresets.json` file (at the *cacheVariables* in the *configurePreset*) and will be applied with `-D` arguments when calling `cmake.configure` using the *CMake() build helper*.

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.cache_variables["foo"] = True
    tc.cache_variables["foo2"] = False
    tc.cache_variables["var"] = "23"
```

The booleans assigned to a cache_variable will be translated to `ON` and `OFF` symbols in CMake.

Using a custom toolchain file

There are two ways of providing custom CMake toolchain files:

- The `conan_toolchain.cmake` file can be completely skipped and replaced by a user one, defining the `tools.cmake.cmaketoolchain:toolchain_file=<filepath>` configuration value.
- A custom user toolchain file can be added (included from) to the `conan_toolchain.cmake` one, by using the `user_toolchain` block described below, and defining the `tools.cmake.cmaketoolchain:user_toolchain=["<filepath>"]` configuration value.

  The configuration `tools.cmake.cmaketoolchain:user_toolchain=["<filepath>"]` can be defined in the `global.conf`. but also creating a Conan package for your toolchain and using `self.conf_info` to declare the toolchain file:

  ```
  import os
  from conan import ConanFile
  class MyToolchainPackage(ConanFile):
      ...
      def package_info(self):
          f = os.path.join(self.package_folder, "mytoolchain.cmake")
          self.conf_info.define("tools.cmake.cmaketoolchain:user_toolchain",
  ↪ [f])
  ```

  If you declare the previous package as a `tool_require`, the toolchain will be automatically applied.

- If you have more than one `tool_requires` defined, you can easily append all the user toolchain values together using the `append` method in each of them, for instance:

  ```
  import os
  from conan import ConanFile
  ```

(continues on next page)

(continued from previous page)

```
class MyToolRequire(ConanFile):
    ...
    def package_info(self):
        f = os.path.join(self.package_folder, "mytoolchain.cmake")
        # Appending the value to any existing one
        self.conf_info.append("tools.cmake.cmaketoolchain:user_toolchain",
↪ f)
```

So, they'll be automatically applied by your `CMakeToolchain` generator without writing any extra code:

```
from conan import ConanFile
from conan.tools.cmake import CMake
class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    exports_sources = "CMakeLists.txt"
    tool_requires = "toolchain1/0.1", "toolchain2/0.1"
    generators = "CMakeToolchain"

    def build(self):
        cmake = CMake(self)
        cmake.configure()
```

Extending and advanced customization

`CMakeToolchain` implements a powerful capability for extending and customizing the resulting toolchain file.

The contents are organized by `blocks` that can be customized. The following predefined blocks are available, and added in this order:

- **user_toolchain**: Allows to include user toolchains from the `conan_toolchain.cmake` file. If the configuration `tools.cmake.cmaketoolchain:user_toolchain=["xxxx", "yyyy"]` is defined, its values will be `include(xxx)\ninclude(yyyy)` as the first lines in `conan_toolchain.cmake`.
- **generic_system**: Defines `CMAKE_SYSTEM_NAME`, `CMAKE_SYSTEM_VERSION`, `CMAKE_SYSTEM_PROCESSOR`, `CMAKE_GENERATOR_PLATFORM`, `CMAKE_GENERATOR_TOOLSET`, `CMAKE_C_COMPILER`, `CMAKE_CXX_COMPILER`
- **android_system**: Defines `ANDROID_PLATFORM`, `ANDROID_STL`, `ANDROID_ABI` and includes `ANDROID_NDK_PATH/build/cmake/android.toolchain.cmake` where `ANDROID_NDK_PATH` comes defined in `tools.android:ndk_path` configuration value.
- **apple_system**: Defines `CMAKE_OSX_ARCHITECTURES`, `CMAKE_OSX_SYSROOT` for Apple systems.
- **fpic**: Defines the `CMAKE_POSITION_INDEPENDENT_CODE` when there is a `options.fPIC`
- **arch_flags**: Defines C/C++ flags like `-m32, -m64` when necessary.
- **libcxx**: Defines `-stdlib=libc++` flag when necessary as well as `_GLIBCXX_USE_CXX11_ABI`.
- **vs_runtime**: Defines the `CMAKE_MSVC_RUNTIME_LIBRARY` variable, as a generator expression for multiple configurations.
- **cppstd**: defines `CMAKE_CXX_STANDARD`, `CMAKE_CXX_EXTENSIONS`
- **parallel**: defines `/MP` parallel build flag for Visual.
- **cmake_flags_init**: defines `CMAKE_XXX_FLAGS` variables based on previously defined Conan variables. The blocks above only define `CONAN_XXX` variables, and this block will define CMake ones like `set(CMAKE_CXX_FLAGS_INIT "${CONAN_CXX_FLAGS}" CACHE STRING "" FORCE)`.

- **try_compile**: Stop processing the toolchain, skipping the blocks below this one, if `IN_TRY_COMPILE` CMake property is defined.
- **find_paths**: Defines `CMAKE_FIND_PACKAGE_PREFER_CONFIG`, `CMAKE_MODULE_PATH`, `CMAKE_PREFIX_PATH` so the generated files from `CMakeDeps` are found.
- **rpath**: Defines `CMAKE_SKIP_RPATH`. By default it is disabled, and it is needed to define `self.blocks["rpath"].skip_rpath=True` if you want to activate `CMAKE_SKIP_RPATH`
- **shared**: defines `BUILD_SHARED_LIBS`.
- **output_dirs**: Define the `CMAKE_INSTALL_XXX` variables.
  - **CMAKE_INSTALL_PREFIX**: Is set with the `package_folder`, so if a "cmake install" operation is run, the artifacts go to that location.
  - **CMAKE_INSTALL_BINDIR**, **CMAKE_INSTALL_SBINDIR** and **CMAKE_INSTALL_LIBEXECDIR**: Set by default to `bin`.
  - **CMAKE_INSTALL_LIBDIR**: Set by default to `lib`.
  - **CMAKE_INSTALL_INCLUDEDIR** and **CMAKE_INSTALL_OLDINCLUDEDIR**: Set by default to `include`.
  - **CMAKE_INSTALL_DATAROOTDIR**: Set by default to `res`.

  If you want to change the default values, adjust the `cpp.package` object at the `layout()` method:

```
def layout(self):
    ...
    # For CMAKE_INSTALL_BINDIR, CMAKE_INSTALL_SBINDIR and CMAKE_
↪INSTALL_LIBEXECDIR, takes the first value:
    self.cpp.package.bindirs = ["mybin"]
    # For CMAKE_INSTALL_LIBDIR, takes the first value:
    self.cpp.package.libdirs = ["mylib"]
    # For CMAKE_INSTALL_INCLUDEDIR, CMAKE_INSTALL_OLDINCLUDEDIR,␣
↪takes the first value:
    self.cpp.package.includedirs = ["myinclude"]
    # For CMAKE_INSTALL_DATAROOTDIR, takes the first value:
    self.cpp.package.resdirs = ["myres"]
```

Note: It is **not valid** to change the self.cpp_info at the `package_info()` method.

Customizing the content blocks

Every block can be customized in different ways:

```
# remove an existing block
def generate(self):
    tc = CMakeToolchain(self)
    tc.blocks.remove("generic_system")

# modify the template of an existing block
def generate(self):
    tc = CMakeToolchain(self)
    tmp = tc.blocks["generic_system"].template
    new_tmp = tmp.replace(...)  # replace, fully replace, append...
    tc.blocks["generic_system"].template = new_tmp
```

(continues on next page)

(continued from previous page)

```
# modify one or more variables of the context
def generate(self):
    tc = CMakeToolchain(conanfile)
    # block.values is the context dictionary
    toolset = tc.blocks["generic_system"].values["toolset"]
    tc.blocks["generic_system"].values["toolset"] = "other_toolset"

# modify the whole context values
def generate(self):
    tc = CMakeToolchain(conanfile)
    tc.blocks["generic_system"].values = {"toolset": "other_toolset"}

# modify the context method of an existing block
import types

def generate(self):
    tc = CMakeToolchain(self)
    generic_block = toolchain.blocks["generic_system"]

    def context(self):
        assert self  # Your own custom logic here
        return {"toolset": "other_toolset"}
    generic_block.context = types.MethodType(context, generic_block)

# completely replace existing block
from conan.tools.cmake import CMakeToolchain

def generate(self):
    tc = CMakeToolchain(self)
    # this could go to a python_requires
    class MyGenericBlock:
        template = "HelloWorld"

        def context(self):
            return {}

    tc.blocks["generic_system"] = MyGenericBlock

# add a completely new block
from conan.tools.cmake import CMakeToolchain
def generate(self):
    tc = CMakeToolchain(self)
    # this could go to a python_requires
    class MyBlock:
        template = "Hello {{myvar}}!!!"

        def context(self):
            return {"myvar": "World"}

    tc.blocks["mynewblock"] = MyBlock
```

For more information about these blocks, please have a look at the source code.

Cross building

The `generic_system` block contains some basic cross-building capabilities. In the general case, the user would want to provide their own user toolchain defining all the specifics, which can be done with the configuration `tools.cmake.cmaketoolchain:user_toolchain`. If this conf value is defined, the `generic_system` block will include the provided file or files, but no further define any CMake variable for cross-building.

If `user_toolchain` is not defined and Conan detects it is cross-building, because the build and host profiles contain different OS or architecture, it will try to define the following variables:

- `CMAKE_SYSTEM_NAME`: `tools.cmake.cmaketoolchain:system_name` configuration if defined, otherwise, it will try to autodetect it. This block will consider cross-building if Android systems (that is managed by other blocks), and not 64bits to 32bits builds in x86_64, sparc and ppc systems.
- `CMAKE_SYSTEM_VERSION`: `tools.cmake.cmaketoolchain:system_version` conf if defined, otherwise `os.version` subsetting (host) when defined
- `CMAKE_SYSTEM_PROCESSOR`: `tools.cmake.cmaketoolchain:system_processor` conf if defined, otherwise `arch` setting (host) if defined

Reference

class CMakeToolchain(*conanfile*, *generator=None*)

generate()
This method will save the generated files to the conanfile.generators_folder

conf

CMakeToolchain is affected by these `[conf]` variables:

- **tools.cmake.cmaketoolchain:toolchain_file** user toolchain file to replace the `conan_toolchain.cmake` one.
- **tools.cmake.cmaketoolchain:user_toolchain** list of user toolchains to be included from the `conan_toolchain.cmake` file.
- **tools.android:ndk_path** value for `ANDROID_NDK_PATH`.
- **tools.cmake.cmaketoolchain:system_name** is not necessary in most cases and is only used to force-define `CMAKE_SYSTEM_NAME`.
- **tools.cmake.cmaketoolchain:system_version** is not necessary in most cases and is only used to force-define `CMAKE_SYSTEM_VERSION`.
- **tools.cmake.cmaketoolchain:system_processor** is not necessary in most cases and is only used to force-define `CMAKE_SYSTEM_PROCESSOR`.
- **tools.cmake.cmaketoolchain:toolset_arch**: Will add the `,host=xxx` specifier in the `CMAKE_GENERATOR_TOOLSET` variable of `conan_toolchain.cmake` file.
- **tools.cmake.cmake_layout:build_folder_vars**: Settings and Options that will produce a different build folder and different CMake presets names.
- **tools.cmake.cmaketoolchain.presets:max_schema_version**: Generate CMakeUserPreset.json compatible with the supplied schema version.
- **tools.build:cxxflags** list of extra C++ flags that will be appended to `CMAKE_CXX_FLAGS_INIT`.
- **tools.build:cflags** list of extra of pure C flags that will be appended to `CMAKE_C_FLAGS_INIT`.

- **tools.build:sharedlinkflags** list of extra linker flags that will be appended to `CMAKE_SHARED_LINKER_FLAGS_INIT`.
- **tools.build:exelinkflags** list of extra linker flags that will be appended to `CMAKE_EXE_LINKER_FLAGS_INIT`.
- **tools.build:defines** list of preprocessor definitions that will be used by `add_definitions()`.
- **tools.build:tools.apple:enable_bitcode** boolean value to enable/disable Bitcode Apple Clang flags, e.g., `CMAKE_XCODE_ATTRIBUTE_ENABLE_BITCODE`.
- **tools.build:tools.apple:enable_arc** boolean value to enable/disable ARC Apple Clang flags, e.g., `CMAKE_XCODE_ATTRIBUTE_CLANG_ENABLE_OBJC_ARC`.
- **tools.build:tools.apple:enable_visibility** boolean value to enable/disable Visibility Apple Clang flags, e.g., `CMAKE_XCODE_ATTRIBUTE_GCC_SYMBOLS_PRIVATE_EXTERN`.
- **tools.build:sysroot** defines the value of `CMAKE_SYSROOT`.

CMake

The `CMake` build helper is a wrapper around the command line invocation of cmake. It will abstract the calls like `cmake --build . --config Release` into Python method calls. It will also add the argument `-DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake` (from the generator `CMakeToolchain`) to the `configure()` call, as well as other possible arguments like `-DCMAKE_BUILD_TYPE=<config>`. The arguments that will be used are obtained from a generated `CMakePresets.json` file.

The helper is intended to be used in the `build()` method, to call CMake commands automatically when a package is being built directly by Conan (create, install)

```
from conan import ConanFile
from conan.tools.cmake import CMake, CMakeToolchain, CMakeDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}

    def generate(self):
        tc = CMakeToolchain(self)
        tc.generate()
        deps = CMakeDeps(self)
        deps.generate()

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
```

Reference

class CMake(*conanfile*)

CMake helper to use together with the CMakeToolchain feature

Parameters **conanfile** – The current recipe object. Always use `self`.

configure(*variables=None*, *build_script_folder=None*)

Reads the `CMakePresets.json` file generated by the *CMakeToolchain* to get:

- The generator, to append `-G="xxx"`.

- The path to the toolchain and append `-DCMAKE_TOOLCHAIN_FILE=/path/conan_toolchain.cmake`
- The declared `cache variables` and append `-Dxxx`.

and call `cmake`.

Parameters

- **variables** – Should be a dictionary of CMake variables and values, that will be mapped to command line `-DVAR=VALUE` arguments. Recall that in the general case information to CMake should be passed in `CMakeToolchain` to be provided in the `conan_toolchain.cmake` file. This `variables` argument is intended for exceptional cases that wouldn't work in the toolchain approach.
- **build_script_folder** – Path to the CMakeLists.txt in case it is not in the declared `self.folders.source` at the `layout()` method.

build(*build_type=None*, *target=None*, *cli_args=None*, *build_tool_args=None*)

Parameters

- **build_type** – Use it only to override the value defined in the `settings.build_type` for a multi-configuration generator (e.g. Visual Studio, XCode). This value will be ignored for single-configuration generators, they will use the one defined in the toolchain file during the install step.
- **target** – Name of the build target to run
- **cli_args** – A list of arguments `[arg1, arg2, ...]` that will be passed to the `cmake --build ... arg1 arg2` command directly.
- **build_tool_args** – A list of arguments `[barg1, barg2, ...]` for the underlying build system that will be passed to the command line after the `--` indicator: `cmake --build ... -- barg1 barg2`

install(*build_type=None*)

Equivalent to run `cmake --build . --target=install`

Parameters **build_type** – Use it only to override the value defined in the settings.build_type. It can fail if the build is single configuration (e.g. Unix Makefiles), as in that case the build type must be specified at configure time, not build type.

test(*build_type=None*, *target=None*, *cli_args=None*, *build_tool_args=None*)

Equivalent to running cmake –build . –target=RUN_TESTS.

Parameters

- **build_type** – Use it only to override the value defined in the `settings.build_type`. It can fail if the build is single configuration (e.g. Unix Makefiles), as in that case the build type must be specified at configure time, not build time.
- **target** – Name of the build target to run, by default `RUN_TESTS` or `test`
- **cli_args** – Same as above `build()`
- **build_tool_args** – Same as above `build()`

conf

CMake() helper is affected by these `[conf]` variables:

- `tools.microsoft.msbuild:verbosity` will accept one of `"Quiet", "Minimal", "Normal", "Detailed", "Diagnostic"` to be passed to the `CMake.build()` command,

when a Visual Studio generator (MSBuild build system) is being used for CMake. It is passed as an argument to the underlying build system via the call `cmake --build . --config Release -- /verbosity:Diagnostic`

- `tools.build:jobs` argument for the `--jobs` parameter when running Ninja generator.
- `tools.microsoft.msbuild:max_cpu_count` argument for the `/m` (`/maxCpuCount`) when running `MSBuild`

cmake_layout

The `cmake_layout()` sets the *folders* and *cpp* attributes to follow the structure of a typical CMake project.

```
from conan.tools.cmake import cmake_layout

def layout(self):
    cmake_layout(self)
```

Note: To try it you can use the `conan new hello/0.1 --template=cmake_lib` template.

The assigned values depend on the CMake generator that will be used. It can be defined with the `tools.cmake.cmaketoolchain:generator` [conf] entry or passing it in the recipe to the `cmake_layout(self, cmake_generator)` function. The assigned values are different if it is a multi-config generator (like Visual Studio or Xcode), or a single-config generator (like Unix Makefiles).

These are the values assigned by the `cmake_layout`:

- `conanfile.folders.source`: *src_folder* argument or `.` if not specified.
- **`conanfile.folders.build`:**
  - `build`: if the cmake generator is multi-configuration.
  - `build/Release` or `build/Debug`: if the cmake generator is single-configuration, depending on the build_type.
- `conanfile.folders.generators`: `build/generators`
- `conanfile.cpp.source.includedirs`: `["include"]`
- **`conanfile.cpp.build.libdirs` and `conanfile.cpp.build.bindirs`:**
  - `["Release"]` or `["Debug"]` for a multi-configuration cmake generator.
  - `.` for a single-configuration cmake generator.

```
def layout(self):
    cmake_layout(self, src_folder="subfolder")
```

Multi-setting/option cmake_layout

The `folders.build` and `conanfile.folders.generators` can be customized to take into account the `settings` and `options` and not only the `build_type`. Use the `tools.cmake.cmake_layout:build_folder_vars` conf to declare a list of settings or options:

```
conan install . -c tools.cmake.cmake_layout:build_folder_vars="['settings.compiler',
↪'options.shared']"
```

For the previous example, the values assigned by the `cmake_layout` (installing the Release/static default configuration) would be:

- **conanfile.folders.build:**
  - build/apple-clang-shared_false: if the cmake generator is multi-configuration.
  - build/apple-clang-shared_false/Debug: if the cmake generator is single-configuration.
- conanfile.folders.generators: build/generators

If we repeat the previous install with a different configuration:

```
conan install . -o shared=True -c tools.cmake.cmake_layout:build_folder_vars="[
↪'settings.compiler', 'options.shared']"
```

The values assigned by the cmake_layout (installing the Release/shared configuration) would be:

- **conanfile.folders.build:**
  - build/apple-clang-shared_true: if the cmake generator is multi-configuration.
  - build/apple-clang-shared_true/Debug: if the cmake generator is single-configuration.
- conanfile.folders.generators: build-apple-clang-shared_true/generators

So we can keep separated folders for any number of different configurations that we want to install.

The CMakePresets.json file generated at the *CMakeToolchain* generator, will also take this tools.cmake.cmake_layout:build_folder_vars config into account to generate different names for the presets, being very handy to install N configurations and building our project for any of them by selecting the chosen preset.

Note: The settings.build_type value is forbidden in tools.cmake.cmake_layout:build_folder_vars because the build_type is already managed automatically with multi-config support in CMakeDeps and CMakeToolchain.

Reference

cmake_layout(*conanfile*, *generator=None*, *src_folder='.'*, *build_folder='build'*)

Parameters

- **conanfile** – The current recipe object. Always use self.
- **generator** – Allow defining the CMake generator. In most cases it doesn't need to be passed, as it will get the value from the configuration tools.cmake.cmaketoolchain:generator, or it will automatically deduce the generator from the settings
- **src_folder** – Value for conanfile.folders.source, change it if your source code (and CMakeLists.txt) is in a subfolder.

7.4.2 conan.tools.gnu

AutotoolsDeps

The AutotoolsDeps is the dependencies generator for Autotools. It will generate shell scripts containing environment variable definitions that the autotools build system can understand.

It can be used by name in conanfiles:

Listing 5: conanfile.py

```
class Pkg(ConanFile):
    generators = "AutotoolsDeps"
```

Listing 6: conanfile.txt

```
[generators]
AutotoolsDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = AutotoolsDeps(self)
        tc.generate()
```

Generated files

It will generate the file `conanautotoolsdeps.sh` or `conanautotoolsdeps.bat`:

```
$ conan install conanfile.py # default is Release
$ source conanautotoolsdeps.sh
# or in Windows
$ conanautotoolsdeps.bat
```

These launchers will define aggregated variables `CPPFLAGS`, `LIBS`, `LDFLAGS`, `CXXFLAGS`, `CFLAGS` that accumulate all dependencies information, including transitive dependencies, with flags like `-I<path>`, `-L<path>`, etc.

At this moment, only the `requires` information is generated, the `tool_requires` one is not managed by this generator yet.

Customization

To modify the computed values, you can access the `.environment` property that returns an *Environment* class.

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = AutotoolsDeps(self)
        tc.environment.remove("CPPFLAGS", "undesired_value")
        tc.environment.append("CPPFLAGS", "var")
        tc.environment.define("OTHER", "cat")
        tc.environment.unset("LDFLAGS")
        tc.generate()
```

Reference

class AutotoolsDeps(*conanfile*)

environment

Returns An `Environment` object containing the computed variables. If you need to modify some of the computed values you can access to the `environment` object.

AutotoolsToolchain

The `AutotoolsToolchain` is the toolchain generator for Autotools. It will generate shell scripts containing environment variable definitions that the autotools build system can understand.

This generator can be used by name in conanfiles:

Listing 7: conanfile.py

```
class Pkg(ConanFile):
    generators = "AutotoolsToolchain"
```

Listing 8: conanfile.txt

```
[generators]
AutotoolsToolchain
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = AutotoolsToolchain(self)
        tc.generate()
```

Generated files

It will generate the file `conanautotoolstoolchain.sh` or `conanautotoolstoolchain.bat` files:

```
$ conan install conanfile.py # default is Release
$ source conanautotoolstoolchain.sh
# or in Windows
$ conanautotoolstoolchain.bat
```

This launchers will append information to the `CPPFLAGS`, `LDFLAGS`, `CXXFLAGS`, `CFLAGS` environment variables that translate the settings and options to the corresponding build flags like `-stdlib=libstdc++`, `-std=gnu14`, architecture flags, etc. It will also append the folder where the Conan generators are located to the `PKG_CONFIG_PATH` environment variable.

This generator will also generate a file called `conanbuild.conf` containing two keys:

- **configure_args**: Arguments to call the `configure` script.
- **make_args**: Arguments to call the `make` script.
- **autoreconf_args**: Arguments to call the `autoreconf` script.

The *Autotools build helper* will use that `conanbuild.conf` file to seamlessly call the configure and make script using these precalculated arguments.

Customization

You can change some attributes before calling the `generate()` method if you want to change some of the precalculated values:

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = AutotoolsToolchain(self)
        tc.configure_args.append("--my_argument")
        tc.generate()
```

- **configure_args: Additional arguments to be passed to the configure script.**
  - **By default the following arguments are passed:**
    - `--prefix`: With the self.package_folder value.
    - `--bindir=${prefix}/bin`
    - `--sbindir=${prefix}/bin`
    - `--libdir=${prefix}/lib`
    - `--includedir=${prefix}/include`
    - `--oldincludedir=${prefix}/include`
    - `--datarootdir=${prefix}/res`
  - **Also if the shared option exists it will add by default:**
    - `--enable-shared`, `--disable-static` if `shared==True`
    - `--disable-shared`, `--enable-static` if `shared==False`
- **make_args** (Defaulted to `[]`): Additional arguments to be passed to he make script.
- **autoreconf_args** (Defaulted to `["--force", "--install"]`): Additional arguments to be passed to he make script.
- **defines** (Defaulted to `[]`): Additional defines.
- **extra_defines** (Defaulted to `[]`): Additional defines.
- **extra_cxxflags** (Defaulted to `[]`): Additional cxxflags.
- **extra_cflags** (Defaulted to `[]`): Additional cflags.
- **extra_ldflags** (Defaulted to `[]`): Additional ldflags.
- **ndebug**: "NDEBUG" if the `settings.build_type` != *Debug*.
- **gcc_cxx11_abi**: "_GLIBCXX_USE_CXX11_ABI" if `gcc/libstdc++`.
- **libcxx**: Flag calculated from `settings.compiler.libcxx`.
- **fpic**: True/False from `options.fpic` if defined.

- **cppstd**: Flag from `settings.compiler.cppstd`
- **arch_flag**: Flag from `settings.arch`
- **build_type_flags**: Flags from `settings.build_type`
- **apple_arch_flag**: Only when cross-building with Apple systems. Flags from `settings.arch`.
- **apple_isysroot_flag**: Only when cross-building with Apple systems. Path to the root sdk.
- **msvc_runtime_flag**: Flag from `settings.compiler.runtime_type` when compiler is `msvc` or `settings.compiler.runtime` when using the deprecated `Visual Studio`.
- **default_configure_install_args** (Defaulted to `True`): If True it will pass automatically the following flags to the configure script:
  - `--prefix`: With the self.package_folder value.
  - `--bindir=${prefix}/bin`
  - `--sbindir=${prefix}/bin`
  - `--libdir=${prefix}/lib`
  - `--includedir=${prefix}/include`
  - `--oldincludedir=${prefix}/include`
  - `--datarootdir=${prefix}/res`

The following attributes are ready-only and will contain the calculated values for the current configuration and customized attributes. Some recipes might need to read them to generate custom build files (not strictly Autotools) with the configuration:

- **defines**
- **cxxflags**
- **cflags**
- **ldflags**

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    def generate(self):
        tc = AutotoolsToolchain(self)
        # Customize the flags
        tc.extra_cxxflags = ["MyFlag"]
        # Read the computed flags and use them (write custom files etc)
        tc.defines
        tc.cxxflags
        tc.cflags
        tc.ldflags
```

If you want to change the default values for `configure_args`, adjust the `cpp.package` object at the `layout()` method:

```
def layout(self):
    ...
    # For bindir and sbindir takes the first value:
    self.cpp.package.bindirs = ["mybin"]
```

(continues on next page)

(continued from previous page)

```
    # For libdir takes the first value:
    self.cpp.package.libdirs = ["mylib"]
    # For includedir and oldincludedir takes the first value:
    self.cpp.package.includedirs = ["myinclude"]
    # For datarootdir takes the first value:
    self.cpp.package.resdirs = ["myres"]
```

Note: It is **not valid** to change the self.cpp_info at the `package_info()` method.

Customizing the environment

If your `Makefile` or `configure` scripts need some other environment variable rather than `CPPFLAGS`, `LDFLAGS`, `CXXFLAGS` or `CFLAGS`, you can customize it before calling the `generate()` method. Call the `environment()` method to calculate the mentioned variables and then add the variables that you need. The `environment()` method returns an *Environment* object:

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        at = AutotoolsToolchain(self)
        env = at.environment()
        env.define("FOO", "BAR")
        at.generate(env)
```

The `AutotoolsToolchain` also sets `CXXFLAGS`, `CFLAGS`, `LDFLAGS` and `CPPFLAGS` reading variables from the `[conf]` section in the profiles. *See the conf reference below*.

Reference

class AutotoolsToolchain(*conanfile*, *namespace=None*)

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **namespace** – This argument avoids collisions when you have multiple toolchain calls in the same recipe. By setting this argument, the *conanbuild.conf* file used to pass information to the build helper will be named as *<namespace>_conanbuild.conf*. The default value is `None` meaning that the name of the generated file is *conanbuild.conf*. This namespace must be also set with the same value in the constructor of the Autotools build helper so that it reads the information from the proper file.

conf

- `tools.build:cxxflags` list of extra C++ flags that will be used by `CXXFLAGS`.
- `tools.build:cflags` list of extra of pure C flags that will be used by `CFLAGS`.
- `tools.build:sharedlinkflags` list of extra linker flags that will be used by `LDFLAGS`.
- `tools.build:exelinkflags` list of extra linker flags that will be used by by `LDFLAGS`.

- `tools.build:defines` list of preprocessor definitions that will be used by `CPPFLAGS`.
- `tools.build:sysroot` defines the `--sysroot` flag to the compiler.

Autotools

The `Autotools` build helper is a wrapper around the command line invocation of autotools. It will abstract the calls like `./configure` or `make` into Python method calls.

Usage:

```
from conan import ConanFile
from conan.tools.gnu import Autotools

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def build(self):
        autotools = Autotools(self)
        autotools.configure()
        autotools.make()
```

It will read the `conanbuild.conf` file generated by the *AutotoolsToolchain* to know read the arguments for calling the configure and make scripts:

- **configure_args**: Arguments to call the `configure` script.
- **make_args**: Arguments to call the `make` script.

Reference

class Autotools(*conanfile*, *namespace=None*)

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **namespace** – this argument avoids collisions when you have multiple toolchain calls in the same recipe. By setting this argument, the *conanbuild.conf* file used to pass information to the toolchain will be named as: *<namespace>_conanbuild.conf*. The default value is `None` meaning that the name of the generated file is *conanbuild.conf*. This namespace must be also set with the same value in the constructor of the AutotoolsToolchain so that it reads the information from the proper file.

configure(*build_script_folder=None*, *args=None*)

Call the configure script.

Parameters

- **args** – List of arguments to use for the `configure` call.
- **build_script_folder** – Subfolder where the *configure* script is located. If not specified conanfile.source_folder is used.

make(*target=None*, *args=None*)

Call the make program.

Parameters

- **target** – (Optional, Defaulted to `None`): Choose which target to build. This allows building of e.g., docs, shared libraries or install for some AutoTools projects
- **args** – (Optional, Defaulted to `None`): List of arguments to use for the `make` call.

install(*args=None*)
This is just an "alias" of `self.make(target="install")`

Parameters **args** – (Optional, Defaulted to `None`): List of arguments to use for the `make` call. By default an argument `DESTDIR=self.package_folder` is added to the call if the passed value is `None`.

autoreconf(*args=None*)
Call `autoreconf`

Parameters **args** – (Optional, Defaulted to `None`): List of arguments to use for the `autoreconf` call.

A note about relocatable shared libraries in macOS built the Autotools build helper

When building a shared library with Autotools in macOS a section `LC_ID_DYLIB` and another `LC_LOAD_DYLIB` are added to the `.dylib`. These sections store `install_name` information, which is the location of the folder where the library or its dependencies are installed. You can check the install_name of your shared libraries using the otool command:

```
$ otool -l path/to/libMyLib.dylib
...
cmd LC_ID_DYLIB
    cmdsize 48
        name path/to/libMyLib.dylib (offset 24)
time stamp 1 Thu Jan  1 01:00:01 1970
    current version 1.0.0
compatibility version 1.0.0
...
Load command 11
        cmd LC_LOAD_DYLIB
    cmdsize 48
        name path/to/dependency.dylib (offset 24)
time stamp 2 Thu Jan  1 01:00:02 1970
    current version 1.0.0
compatibility version 1.0.0
...
```

Why is this a problem when using Conan?

When using Conan the library will be built in the local cache and this means that this location will point to Conan's local cache folder where the library was installed. This location is where the library tells any other binaries using it where to load it at runtime. This is a problem since you can build the shared library in one machine, then upload it to a server and install it in another machine to use it. In this case, as Autotools behaves by default, you would have a library storing an `install_name` pointing to a folder that does not exist in your current machine so you would get linker errors when building.

How to adress this problem in Conan

The only thing Conan can do to make these shared libraries relocatable is to patch the built binaries after installation. To do this, when using the `Autotools` build helper and after running the Makefile's `install()` step, you can use the *fix_apple_shared_install_name()* tool to search for the built `.dylib` files and patch them by running the `install_name_tool` macOS utility, like this:

```
from conan.tools.apple import fix_apple_shared_install_name
class HelloConan(ConanFile):
```

(continues on next page)

(continued from previous page)

```
  ...
  def package(self):
      autotools = Autotools(self)
      autotools.install()
      fix_apple_shared_install_name(self)
```

This will change the value of the `LC_ID_DYLIB` and `LC_LOAD_DYLIB` sections in the `.dylib` file to:

```
$ otool -l path/to/libMyLib.dylib
...
cmd LC_ID_DYLIB
    cmdsize 48
        name @rpath/libMyLib.dylib (offset 24)
time stamp 1 Thu Jan  1 01:00:01 1970
    current version 1.0.0
compatibility version 1.0.0
...
Load command 11
        cmd LC_LOAD_DYLIB
    cmdsize 48
        name @rpath/dependency.dylib (offset 24)
time stamp 2 Thu Jan  1 01:00:02 1970
    current version 1.0.0
compatibility version 1.0.0
```

The `@rpath` special keyword will tell the loader to search a list of paths to find the library. These paths can be defined by the consumer of that library by defining the `LC_RPATH` field. This is done by passing the `-Wl,-rpath -Wl,/path/to/libMyLib.dylib` linker flag when building the consumer of the library. Then if Conan builds an executable that consumes the `libMyLib.dylib` library, it will automatically add the `-Wl,-rpath -Wl,/path/to/libMyLib.dylib` flag so that the library is correctly found when building.

PkgConfigDeps

The `PkgConfigDeps` is the dependencies generator for pkg-config. Generates pkg-config files named `<PKG-NAME>.pc` containing a valid pkg-config file syntax.

This generator can be used by name in conanfiles:

Listing 9: conanfile.py

```
class Pkg(ConanFile):
    generators = "PkgConfigDeps"
```

Listing 10: conanfile.txt

```
[generators]
PkgConfigDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.gnu import PkgConfigDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "zlib/1.2.11"
```

(continues on next page)

(continued from previous page)

```
    def generate(self):
        pc = PkgConfigDeps(self)
        pc.generate()
```

Generated files

pkg-config format files named `<PKG-NAME>.pc`, containing a valid pkg-config file syntax. The `prefix` variable is automatically adjusted to the `package_folder`:

```
prefix=/Users/YOUR_USER/.conan/data/zlib/1.2.11/_/_/package/
↪647afeb69d3b0a2d3d316e80b24d38c714cc6900
libdir=${prefix}/lib
includedir=${prefix}/include

Name: zlib
Description: Conan package: zlib
Version: 1.2.11
Libs: -L"${libdir}" -lz -F Frameworks
Cflags: -I"${includedir}"
```

Customization

Naming

By default, the `*.pc` files will be named following these rules:

- For packages, it uses the package name, e.g., package `zlib/1.2.11` -> `zlib.pc`.
- For components, the package name + hyphen + component name, e.g., `openssl/3.0.0` with `self.cpp_info.components["crypto"]` -> `openssl-crypto.pc`.

You can change that default behavior with the `pkg_config_name` and `pkg_config_aliases` properties. See *Properties section below*.

If a recipe uses **components**, the files generated will be `<[PKG-NAME]-[COMP-NAME]>.pc` with their corresponding flags and require relations.

Additionally, a `<PKG-NAME>.pc` is generated to maintain compatibility for consumers with recipes that start supporting components. This `<PKG-NAME>.pc` file declares all the components of the package as requires while the rest of the fields will be empty, relying on the propagation of flags coming from the components `<[PKG-NAME]-[COMP-NAME]>.pc` files.

Reference

class PkgConfigDeps(*conanfile*)

content
Get all the .pc files content

generate()
Save all the **.pc* files

Properties

The following properties affect the `PkgConfigDeps` generator:

- **pkg_config_name** property will define the name of the generated `*.pc` file (`xxxxx.pc`)
- **pkg_config_aliases** property sets some aliases of any package/component name for *pkg_config* generator. This property only accepts list-like Python objects.
- **pkg_config_custom_content** property will add user defined content to the *.pc* files created by this generator.
- **component_version** property sets a custom version to be used in the `Version` field belonging to the created `*.pc` file for that component.

These properties can be defined at global `cpp_info` level or at component level.

Example:

```
def package_info(self):
    custom_content = "datadir=${prefix}/share"
    self.cpp_info.set_property("pkg_config_custom_content", custom_content)
    self.cpp_info.set_property("pkg_config_name", "myname")
    self.cpp_info.components["mycomponent"].set_property("pkg_config_name",
↪"componentname")
    self.cpp_info.components["mycomponent"].set_property("pkg_config_aliases", [
↪"alias1", "alias2"])
    self.cpp_info.components["mycomponent"].set_property("component_version", "1.14.12
↪")
```

PkgConfig

This tool can execute `pkg_config` executable to extract information from existing `.pc` files. This can be useful for example to create a "system" package recipe over some system installed library, as a way to automatically extract the `.pc` information from the system. Or if some proprietary package has a build system that only outputs `.pc` files.

Usage:

Read a `pc` file and access the information:

```
pkg_config = PkgConfig(conanfile, "libastral", pkg_config_path=<somedir>)

print(pkg_config.provides) # something like"libastral = 6.6.6"
print(pkg_config.version) # something like"6.6.6"
print(pkg_config.includedirs) # something like['/usr/local/include/libastral']
print(pkg_config.defines) # something like['_USE_LIBASTRAL']
print(pkg_config.libs) # something like['astral', 'm']
print(pkg_config.libdirs) # something like['/usr/local/lib/libastral']
print(pkg_config.linkflags) # something like['-Wl,--whole-archive']
print(pkg_config.variables['prefix']) # something like'/usr/local'
```

Use the `pc` file information to fill a `cpp_info` object:

```
def package_info(self):
    pkg_config = PkgConfig(conanfile, "libastral", pkg_config_path=tmp_dir)
    pkg_config.fill_cpp_info(self.cpp_info, is_system=False, system_libs=["m", "rt"])
```

Reference

class PkgConfig(*conanfile*, *library*, *pkg_config_path=None*)

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **library** – The library which `.pc` file is to be parsed. It must exist in the pkg_config path.

- **pkg_config_path** – If defined it will be prepended to `PKG_CONFIG_PATH` environment variable, so the execution finds the required files.

fill_cpp_info(*cpp_info*, *is_system=True*, *system_libs=None*)
Method to fill a cpp_info object from the PkgConfig configuration

Parameters

- **cpp_info** – Can be the global one (self.cpp_info) or a component one (self.components["foo"].cpp_info).
- **is_system** – If `True`, all detected libraries will be assigned to `cpp_info.system_libs`, and none to `cpp_info.libs`.
- **system_libs** – If `True`, all detected libraries will be assigned to `cpp_info.system_libs`, and none to `cpp_info.libs`.

conf

This helper will listen to `tools.gnu:pkg_config` *global_conf* to define the `pkg_config` executable name or full path. It will by default it is `pkg-config`.

7.4.3 conan.tools.apple

XcodeDeps

The `XcodeDeps` tool is the dependency information generator for *Xcode*. It will generate multiple *.xcconfig* configuration files, the can be used by consumers using *xcodebuild* or *Xcode*. To use them just add the generated configuration files to the Xcode project or set the `-xcconfig` argument from the command line.

The `XcodeDeps` generator can be used by name in conanfiles:

Listing 11: conanfile.py

```
class Pkg(ConanFile):
    generators = "XcodeDeps"
```

Listing 12: conanfile.txt

```
[generators]
XcodeDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 13: conanfile.py

```
from conan import ConanFile
from conan.tools.apple import XcodeDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "libpng/1.6.37@" # Note libpng has zlib as transitive dependency

    def generate(self):
        xcode = XcodeDeps(self)
        xcode.generate()
```

When the `XcodeDeps` generator is used, every invocation of `conan install` will generate several configuration files, per dependency and configuration. For the *conanfile.py* above, for example:

```
$ conan install conanfile.py # default is Release
$ conan install conanfile.py -s build_type=Debug
```

This generator is multi-configuration. It will generate different files for the different *Debug/Release* configurations for each requirement. It will also generate one single file (*conandeps.xcconfig*) aggregating all the files for the direct dependencies (just *libpng* in this case). The above commands generate the following files:

```
.
├── conan_config.xcconfig
├── conan_libpng.xcconfig
├── conan_libpng_libpng.xcconfig
├── conan_libpng_libpng_debug_x86_64.xcconfig
├── conan_libpng_libpng_release_x86_64.xcconfig
├── conan_zlib.xcconfig
├── conan_zlib_zlib.xcconfig
├── conan_zlib_zlib_debug_x86_64.xcconfig
├── conan_zlib_zlib_release_x86_64.xcconfig
└── conandeps.xcconfig
```

The first `conan install` with the default *Release* and *x86_64* configuration generates:

- *conan_libpng_libpng_release_x86_64.xcconfig*: declares variables with conditional logic to be considered only for the active configuration in *Xcode* or the one passed by command line to *xcodebuild*.
- *conan_libpng_libpng.xcconfig*: includes *conan_libpng_libpng_release_x86_64.xcconfig* and declares the following *Xcode* build settings: `HEADER_SEARCH_PATHS`, `GCC_PREPROCESSOR_DEFINITIONS`, `OTHER_CFLAGS`, `OTHER_CPLUSPLUSFLAGS`, `FRAMEWORK_SEARCH_PATHS`, `LIBRARY_SEARCH_PATHS`, `OTHER_LDFLAGS`. It also includes the generated *xcconfig* files for transitive dependencies (*conan_zlib_zlib.xcconfig* in this case).
- *conan_libpng.xcconfig*: in this case it only includes *conan_libpng_libpng.xcconfig*, but in the case that the required package has components, this file will include all of the components of the package.
- Same 3 files will be generated for each dependency in the graph. In this case, as *zlib* is a dependency of *libpng* it will generate: *conan_zlib_zlib_release_x86_64.xcconfig*, *conan_zlib_zlib.xcconfig* and *conan_zlib.xcconfig*.
- *conandeps.xcconfig*: configuration files including all direct dependencies, in this case, it just includes `conan_libpng.xcconfig`.
- The main *conan_config.xcconfig* file, to be added to the project. Includes both the files from this generator and the generated by the *XcodeToolchain* in case it was also set.

The second `conan install -s build_type=Debug` generates:

- *conan_libpng_libpng_debug_x86_64.xcconfig*: same variables as the one below for *Debug* configuration.
- *conan_libpng_libpng.xcconfig*: this file has been already creted by the previous command, now it's modified to add the include for *conan_libpng_debug_x86_64.xcconfig*.
- *conan_libpng.xcconfig*: this file will stay the same.
- Like in the previous command the same 3 files will be generated for each dependency in the graph. In this case, as *zlib* is a dependency of *libpng* it will generate: *conan_zlib_zlib_debug_x86_64.xcconfig*, *conan_zlib_zlib.xcconfig* and *conan_zlib.xcconfig*.
- *conandeps.xcconfig*: configuration files including all direct dependencies, in this case, it just includes `conan_libpng.xcconfig`.
- The main *conan_config.xcconfig* file, to be added to the project. Includes both the files from this generator and the generated by the *XcodeToolchain* in case it was also set.

If you want to add this dependencies to you Xcode project, you just have to add the *conan_config.xcconfig* configuration file for all of the configurations you want to use (usually *Debug* and *Release*).

Components support

This generator supports packages with components. That means that:

- If a **dependency** `package_info()` declares `cpp_info.requires` on some components, the generated *.xcconfig* files will contain includes to only those components.
- The current package `requires` will be fully dependent on and all components. Recall that the `package_info()` only applies for consumers, but not to the current package.

Custom configurations

If your Xcode project defines custom configurations, like `ReleaseShared`, or `MyCustomConfig`, it is possible to define it into the `XcodeDeps` generator, so different project configurations can use different set of dependencies. Let's say that our current project can be built as a shared library, with the custom configuration `ReleaseShared`, and the package also controls this with the `shared` option:

```
from conan import ConanFile
from conan.tools.apple import XcodeDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    options = {"shared": [True, False]}
    default_options = {"shared": False}
    requires = "zlib/1.2.11"

    def generate(self):
        xcode = XcodeDeps(self)
        # We assume that -o *:shared=True is used to install all shared deps too
        if self.options.shared:
            xcode.configuration = str(self.settings.build_type) + "Shared"
        xcode.generate()
```

This will manage to generate new *.xcconfig* files for this custom configuration, and when you switch to this configuration in the IDE, the build system will take the correct values depending wether we want to link with shared or static libraries.

XcodeToolchain

The `XcodeToolchain` is the toolchain generator for Xcode. It will generate *.xcconfig* configuration files that can be added to Xcode projects. This generator translates the current package configuration, settings, and options, into Xcode *.xcconfig* files syntax.

The `XcodeToolchain` generator can be used by name in conanfiles:

Listing 14: conanfile.py

```
class Pkg(ConanFile):
    generators = "XcodeToolchain"
```

Listing 15: conanfile.txt

```
[generators]
XcodeToolchain
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.apple import XcodeToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = XcodeToolchain(self)
        tc.generate()
```

The `XcodeToolchain` will generate three files after a `conan install` command. As explained above for the XcodeDeps generator, each different configuration will create a set of files with different names. For example, running `conan install` for *Release* first and then *Debug* configuration:

```
$ conan install conanfile.py # default is Release
$ conan install conanfile.py -s build_type=Debug
```

Will create these files:

```
.
├── conan_config.xcconfig
├── conantoolchain_release_x86_64.xcconfig
├── conantoolchain_debug_x86_64.xcconfig
├── conantoolchain.xcconfig
└── conan_global_flags.xcconfig
```

Those files are:

- The main *conan_config.xcconfig* file, to be added to the project. Includes both the files from this generator and the generated by the *XcodeDeps* in case it was also set.
- *conantoolchain_<debug/release>_x86_64.xcconfig*: declares `CLANG_CXX_LIBRARY`, `CLANG_CXX_LANGUAGE_STANDARD` and `MACOSX_DEPLOYMENT_TARGET` variables with conditional logic depending on the build configuration, architecture and sdk set.
- *conantoolchain.xcconfig*: aggregates all the *conantoolchain_<config>_<arch>.xcconfig* files for the different installed configurations.
- *conan_global_flags.xcconfig*: this file will only be generated in case of any configuration variables related to compiler or linker flags are set. Check *the configuration section* below for more details.

Every invocation to `conan install` with different configuration will create a new *conantoolchain_<config>_<arch>.xcconfig* file that is aggregated in the *conantoolchain.xcconfig*, so you can have different configurations included in your Xcode project.

The XcodeToolchain files can declare the following Xcode build settings based on Conan settings values:

- `MACOSX_DEPLOYMENT_TARGET` is based on the value of the `os.version` setting and will make the build system to pass the flag `-mmacosx-version-min` with that value (if set). It defines the operating system version the binary should run into.
- `CLANG_CXX_LANGUAGE_STANDARD` is based on the value of the `compiler.cppstd` setting that sets the C++ language standard.
- `CLANG_CXX_LIBRARY` is based on the value of the compiler.libcxx setting and sets the version of the C++ standard library to use.

One of the advantages of using toolchains is that they can help to achieve the exact same build with local development flows, than when the package is created in the cache.

conf

This toolchain is also affected by these **[conf]** variables:

- `tools.build:cxxflags` list of C++ flags.
- `tools.build:cflags` list of pure C flags.
- `tools.build:sharedlinkflags` list of flags that will be used by the linker when creating a shared library.
- `tools.build:exelinkflags` list of flags that will be used by the linker when creating an executable.
- `tools.build:defines` list of preprocessor definitions.

If you set any of these variables, the toolchain will use them to generate the `conan_global_flags.xcconfig` file that will be included from the `conan_config.xcconfig` file.

XcodeBuild

The `XcodeBuild` build helper is a wrapper around the command line invocation of Xcode. It will abstract the calls like `xcodebuild -project app.xcodeproj -configuration <config> -arch <arch> ...`

The `XcodeBuild` helper can be used like:

```
from conan import conanfile
from conan.tools.apple import XcodeBuild

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def build(self):
        xcodebuild = XcodeBuild(self)
        xcodebuild.build("app.xcodeproj")
```

Reference

class XcodeBuild(*conanfile*)

__init__(*conanfile*)
Initialize self. See help(type(self)) for accurate signature.

XcodeBuild.**build**(*xcodeproj*, *target=None*)
Call to `xcodebuild` to build a Xcode project.

Parameters

- **xcodeproj** – the *xcodeproj* file to build.
- **target** – the target to build, in case this argument is passed to the `build()` method it will add the `-target` argument to the build system call. If not passed, it will build all the targets passing the `-alltargets` argument instead.

Returns the return code for the launched `xcodebuild` command.

The `Xcode.build()` method internally implements a call to `xcodebuild` like:

```
$ xcodebuild -project app.xcodeproj -configuration <configuration> -arch
↪<architecture> <sdk> <verbosity> -target <target>/-alltargets
```

Where:

- `configuration` is the configuration, typically *Release* or *Debug*, which will be obtained from `settings.build_type`.
- `architecture` is the build architecture, a mapping from the `settings.arch` to the common architectures defined by Apple 'i386', 'x86_64', 'armv7', 'arm64', etc.
- `sdk` is set based on the values of the `os.sdk` and `os.sdk_version` defining the `SDKROOT` Xcode build setting according to them. For example, setting `os.sdk=iOS` and *os.sdk_version=8.3*' will pass `SDKROOT=iOS8.3` to the build system. In case you defined the `tools.apple:sdk_path` in your **[conf]** this value will take preference and will directly pass `SDKROOT=<tools.apple:sdk_path>` so **take into account** that for this case the skd located in that path should set your `os.sdk` and `os.sdk_version` settings values.
- `verbosity` is the verbosity level for the build and can take value 'verbose' or 'quiet' if set by `tools.apple.xcodebuild:verbosity` in your **[conf]**

conf

- `tools.apple.xcodebuild:verbosity` verbosity value for the build, can be 'verbose' or 'quiet'
- `tools.apple:sdk_path` path for the sdk location, will set the `SDKROOT` value with preference over composing the value from the `os.sdk` and `os.sdk_version` settings.

conan.tools.apple.fix_apple_shared_install_name()

fix_apple_shared_install_name(*conanfile*)

Search for all the *dylib* files in the conanfile's *package_folder* and fix both the `LC_ID_DYLIB` and `LC_LOAD_DYLIB` fields on those files using the *install_name_tool* utility available in macOS to set `@rpath`.

This tool will search for all the *dylib* files in the *conanfile.package_folder* and fix both the `LC_ID_DYLIB` and `LC_LOAD_DYLIB` fields on those files using the *install_name_tool* utility available in macOS.

- For `LC_ID_DYLIB` which is the field containing the install name of the library, it will change the install name to one that uses the `@rpath`. For example, if the install name is `/path/to/lib/libname.dylib`, the new install name will be `@rpath/libname.dylib`. This is done by executing internally something like:

```
install_name_tool /path/to/lib/libname.dylib -id @rpath/libname.dylib
```

- For `LC_LOAD_DYLIB` which is the field containing the path to the library dependencies, it will change the path of the dependencies to one that uses the `@rpath`. For example, if the path is `/path/to/lib/dependency.dylib`, the new path will be `@rpath/dependency.dylib`. This is done by executing internally something like:

```
install_name_tool /path/to/lib/libname.dylib -change /path/to/lib/dependency.dylib␣
↪@rpath/dependency.dylib
```

This tool is typically needed by recipes that use Autotools as the build system and in the case that the correct install names are not fixed in the library being packaged. Use this tool, if needed, in the conanfile's `package()` method like:

```
from conan.tools.apple import fix_apple_shared_install_name
class HelloConan(ConanFile):
  ...
  def package(self):
      autotools = Autotools(self)
      autotools.install()
      fix_apple_shared_install_name(self)
```

7.4.4 conan.tools.env

Environment

`Environment` is a generic class that helps to define modifications to the environment variables. This class is used by other tools like the *conan.tools.gnu* autotools helpers and the *VirtualBuildEnv* and *VirtualRunEnv* generator. It is important to highlight that this is a generic class, to be able to use it, a specialization for the current context (shell script, bat file, path separators, etc), a `EnvVars` object needs to be obtained from it.

Variable declaration

```
from conan.tools.env import Environment

def generate(self):
    env = Environment()
    env.define("MYVAR1", "MyValue1")  # Overwrite previously existing MYVAR1 with new␣
↪value
    env.append("MYVAR2", "MyValue2")  # Append to existing MYVAR2 the new value
    env.prepend("MYVAR3", "MyValue3") # Prepend to existing MYVAR3 the new value
    env.remove("MYVAR3", "MyValue3")  # Remove the MyValue3 from MYVAR3
    env.unset("MYVAR4")               # Remove MYVAR4 definition from environment

    # And the equivalent with paths
    env.define_path("MYPATH1", "path/one")  # Overwrite previously existing MYPATH1␣
↪with new value
    env.append_path("MYPATH2", "path/two")  # Append to existing MYPATH2 the new value
    env.prepend_path("MYPATH3", "path/three") # Prepend to existing MYPATH3 the new␣
↪value
```

The "normal" variables (the ones declared with `define`, `append` and `prepend`) will be appended with a space, by default, but the `separator` argument can be provided to define a custom one.

The "path" variables (the ones declared with `define_path`, `append_path` and `prepend_path`) will be appended with the default system path separator, either `:` or `;`, but it also allows defining which one.

Composition

Environments can be composed:

```
from conan.tools.env import Environment

env1 = Environment()
env1.define(...)
env2 = Environment()
env2.append(...)

env1.compose_env(env2) # env1 has priority, and its modifications will prevail
```

Obtaining environment variables

You can obtain an `EnvVars` object with the `vars()` method like this:

```
from conan.tools.env import Environment

def generate(self):
    env = Environment()
    env.define("MYVAR1", "MyValue1")
```

(continues on next page)

(continued from previous page)

```
    envvars = env.vars(self, scope="build")
    # use the envvars object
```

The default `scope` is equal `"build"`, which means that if this `envvars` generate a script to activate the variables, such script will be automatically added to the `conanbuild.sh|bat` one, for users and recipes convenience. Conan generators use `build` and `run` scope, but it might be possible to manage other scopes too.

Environment definition

There are some other places where `Environment` can be defined and used:

- In recipes `package_info()` method, in new `self.buildenv_info` and `self.runenv_info`, this environment will be propagated via `VirtualBuildEnv` and `VirtualRunEnv` respectively to packages depending on this recipe.
- In generators like `AutootoolsDeps`, `AutotoolsToolchain`, that need to define environment for the current recipe.
- In profiles new `[buildenv]` section.

The definition in `package_info()` is as follow, taking into account that both `self.buildenv_info` and `self.runenv_info` are objects of `Environment()` class.

```
from conan import ConanFile

class App(ConanFile):
    name = "mypkg"
    version = "1.0"
    settings = "os", "arch", "compiler", "build_type"

    def package_info(self):
        # This is information needed by consumers to build using this package
        self.buildenv_info.append("MYVAR", "MyValue")
        self.buildenv_info.prepend_path("MYPATH", "some/path/folder")

        # This is information needed by consumers to run apps that depends on this␣
↪package
        # at runtime
        self.runenv_info.define("MYPKG_DATA_DIR", os.path.join(self.package_folder,
                                                               "datadir"))
```

Reference

class Environment

Generic class that helps to define modifications to the environment variables.

dumps()

Returns A string with a profile-like original definition, not the full environment values

define(*name*, *value*, *separator=' '*)

Define *name* environment variable with value *value*

Parameters

- **name** – Name of the variable
- **value** – Value that the environment variable will take
- **separator** – The character to separate appended or prepended values

unset(*name*)
clears the variable, equivalent to a unset or set XXX=

Parameters **name** – Name of the variable to unset

append(*name*, *value*, *separator=None*)
Append the *value* to an environment variable *name*

Parameters

- **name** – Name of the variable to append a new value
- **value** – New value
- **separator** – The character to separate the appended value with the previous value. By default it will use a blank space.

append_path(*name*, *value*)
Similar to "append" method but indicating that the variable is a filesystem path. It will automatically handle the path separators depending on the operating system.

Parameters

- **name** – Name of the variable to append a new value
- **value** – New value

prepend(*name*, *value*, *separator=None*)
Prepend the *value* to an environment variable *name*

Parameters

- **name** – Name of the variable to prepend a new value
- **value** – New value
- **separator** – The character to separate the prepended value with the previous value

prepend_path(*name*, *value*)
Similar to "prepend" method but indicating that the variable is a filesystem path. It will automatically handle the path separators depending on the operating system.

Parameters

- **name** – Name of the variable to prepend a new value
- **value** – New value

remove(*name*, *value*)
Removes the *value* from the variable *name*.

Parameters

- **name** – Name of the variable
- **value** – Value to be removed.

compose_env(*other*)
Compose an Environment object with another one. `self` has precedence, the "other" will add/append if possible and not conflicting, but `self` mandates what to do. If `self` has `define()`, without placeholder, that will remain.

Parameters **other** (class:*Environment*) – the "other" Environment

vars(*conanfile*, *scope='build'*)
Return an EnvVars object from the current Environment object :param conanfile: Instance of a conanfile, usually `self` in a recipe :param scope: Determine the scope of the declared variables. :return:

deploy_base_folder(*package_folder*, *deploy_folder*)
Make the paths relative to the deploy_folder

EnvVars

`EnvVars` is a class that represents an instance of environment variables for a given system. It is obtained from the generic *Environment* class.

This class is used by other tools like the *conan.tools.gnu* autotools helpers and the *VirtualBuildEnv* and *VirtualRunEnv* generator.

Creating environment files

`EnvVars` object can generate environment files (shell, bat or powershell scripts):

```
def generate(self):
    env1 = Environment()
    env1.define("foo", "var")
    envvars = env1.vars(self)
    envvars.save_script("my_env_file")
```

Although it potentially could be used in other methods, this functionality is intended to work in the `generate()` method.

It will generate automatically a `my_env_file.bat` for Windows systems or `my_env_file.sh` otherwise.

In Windows, it is possible to opt-in to generate Powershell `.ps1` scripts instead of `.bat` ones, using the conf `tools.env.virtualenv:powershell=True`.

Also, by default, Conan will automatically append that launcher file path to a list that will be used to create a `conanbuild.bat|sh|ps1` file aggregating all the launchers in order. The `conanbuild.sh|bat|ps1` launcher will be created after the execution of the `generate()` method.

The `scope` argument (`"build"` by default) can be used to define different scope of environment files, to aggregate them separately. For example, using a `scope="run"`, like the `VirtualRunEnv` generator does, will aggregate and create a `conanrun.bat|sh|ps1` script:

```
def generate(self):
    env1 = Environment(self)
    env1.define("foo", "var")
    envvars = env1.vars(self, scope="run")
    # Will append "my_env_file" to "conanrun.bat|sh|ps1"
    envvars.save_script("my_env_file")
```

You can also use `scope=None` argument to avoid appending the script to the aggregated `conanbuild.bat|sh|ps1`:

```
env1 = Environment(self)
env1.define("foo", "var")
# Will not append "my_env_file" to "conanbuild.bat|sh|ps1"
envvars = env1.vars(self, scope=None)
envvars.save_script("my_env_file")
```

Running with environment files

The `conanbuild.bat|sh|ps1` launcher will be executed by default before calling every `self.run()` command. This would be typically done in the `build()` method.

You can change the default launcher with the `env` argument of `self.run()`:

```
...
def build(self):
    # This will automatically wrap the "foo" command with the correct environment:
    # source my_env_file.sh && foo
    # my_env_file.bat && foo
    # powershell my_env_file.ps1 ; cmd c/ foo
    self.run("foo", env=["my_env_file"])
```

Applying the environment variables

As an alternative to running a command, environments can be applied in the python environment:

```
from conan.tools.env import Environment

env1 = Environment(self)
env1.define("foo", "var")
envvars = env1.vars(self)
with envvars.apply():
   # Here os.getenv("foo") == "var"
   ...
```

Iterating the variables

You can iterate the environment variables of an `EnvVars` object like this:

```
env1 = Environment()
env1.append("foo", "var")
env1.append("foo", "var2")
envvars = env1.vars(self)
for name, value in envvars.items():
    assert name == "foo":
    assert value == "var var2"
```

Warning: In Windows, there is a limit to the size of environment variables, a total of 32K for the whole environment, but spcifically the PATH variable has a limit of 2048 characters. That means that the above utils could hit that limit, for example for large dependency graphs where all packages contribute to the PATH env-var.

This can be mitigated by:

- Putting the Conan cache closer to C:/ for shorter paths
- Better definition of what dependencies can contribute to the PATH env-var
- Other mechanisms for things like running with many shared libraries dependencies with too many .dlls, like `deployers`

Reference

class EnvVars(*conanfile*, *env*, *scope*)

Represents an instance of environment variables for a given system. It is obtained from the generic Environment class.

items()

Returns a dict with variable name as keys and variable values as values

Returns `{str: str}` (varname: value)

apply()
Context manager to apply the declared variables to the current `os.environ` restoring the original environment when the context ends.

save_script(*filename*)
Saves a script file (bat, sh, ps1) with a launcher to set the environment. If the conf "tools.env.virtualenv:powershell" is set to True it will generate powershell launchers if Windows.

Parameters **filename** – Name of the file to generate. If the extension is provided, it will generate the launcher script for that extension, otherwise the format will be deduced checking if we are running inside Windows (checking also the subsystem) or not.

VirtualBuildEnv

VirtualBuildEnv is a generator that produces a *conanbuildenv* .bat or .sh script containing the environment variables of the build time context:

- From the `self.buildenv_info` of the direct `tool_requires` in "build" context.
- From the `self.runenv_info` of the transitive dependencies of those `tool_requires`.

It can be used by name in conanfiles:

Listing 16: conanfile.py

```
class Pkg(ConanFile):
    generators = "VirtualBuildEnv"
```

Listing 17: conanfile.txt

```
[generators]
VirtualBuildEnv
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 18: conanfile.py

```
from conan import ConanFile
from conan.tools.env import VirtualBuildEnv

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = VirtualBuildEnv(self)
        ms.generate()
```

Generated files

This generator (for example the invocation of `conan install --tool-require=cmake/3.20.0@ -g VirtualBuildEnv`) will create the following files:

- conanbuildenv-release-x86_64.(bat|sh): This file contains the actual definition of environment variables like PATH, LD_LIBRARY_PATH, etc, and any other variable defined in the dependencies `buildenv_info` corresponding to the `build` context, and to the current installed configuration. If a repeated call is done with other settings, a different file will be created. After the execution or sourcing of this file, a new deactivation script will be generated, capturing the current environment, so the environment can

be restored when desired. The file will be named also following the current active configuration, like `deactivate_conanbuildenv-release-x86_64.bat`.

- conanbuild.(bat|sh): Accumulates the calls to one or more other scripts, in case there are multiple tools in the generate process that create files, to give one single convenient file for all. This only calls the latest specific configuration one, that is, if `conan install` is called first for Release build type, and then for Debug, `conanbuild.(bat|sh)` script will call the Debug one.
- deactivate_conanbuild.(bat|sh): Accumulates the deactivation calls defined in the above `conanbuild.(bat|sh)`. This file should only be called after the accumulated activate has been called first.

Reference

class VirtualBuildEnv(*conanfile*)

Calculates the environment variables of the build time context and produces a conanbuildenv .bat or .sh script

environment()

Returns an `Environment` object containing the environment variables of the build context.

Returns an `Environment` object instance containing the obtained variables.

vars(*scope='build'*)

Parameters **scope** – Scope to be used.

Returns An `EnvVars` instance containing the computed environment variables.

generate(*scope='build'*)

Produces the launcher scripts activating the variables for the build context.

Parameters **scope** – Scope to be used.

VirtualRunEnv

`VirtualRunEnv` is a generator that produces a launcher *conanrunenv* .bat or .sh script containing environment variables of the run time environment.

The launcher contains the runtime environment information, anything that is necessary in the environment to actually run the compiled executables and applications. The information is obtained from:

- The `self.runenv_info` of the dependencies corresponding to the `host` context.
- Also automatically deduced from the `self.cpp_info` definition of the package, to define `PATH`, `LD_LIBRARY_PATH`, `DYLD_LIBRARY_PATH` and `DYLD_FRAMEWORK_PATH` environment variables.

It can be used by name in conanfiles:

Listing 19: conanfile.py

```
class Pkg(ConanFile):
    generators = "VirtualRunEnv"
```

Listing 20: conanfile.txt

```
[generators]
VirtualRunEnv
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 21: conanfile.py

```
from conan import ConanFile
from conan.tools.env import VirtualRunEnv

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = VirtualRunEnv(self)
        ms.generate()
```

Generated files

- conanrunenv-release-x86_64.(bat|sh): This file contains the actual definition of environment variables like PATH, LD_LIBRARY_PATH, etc, and `runenv_info` of dependencies corresponding to the `host` context, and to the current installed configuration. If a repeated call is done with other settings, a different file will be created.
- conanrun.(bat|sh): Accumulates the calls to one or more other scripts to give one single convenient file for all. This only calls the latest specific configuration one, that is, if `conan install` is called first for Release build type, and then for Debug, `conanrun.(bat|sh)` script will call the Debug one.

After the execution of one of those files, a new deactivation script will be generated, capturing the current environment, so the environment can be restored when desired. The file will be named also following the current active configuration, like `deactivate_conanrunenv-release-x86_64.bat`.

Reference

class VirtualRunEnv(*conanfile*)

Calculates the environment variables of the runtime context and produces a conanrunenv .bat or .sh script

Parameters **conanfile** – The current recipe object. Always use `self`.

environment()

Returns an `Environment` object containing the environment variables of the run context.

Returns an `Environment` object instance containing the obtained variables.

vars(*scope='run'*)

Parameters **scope** – Scope to be used.

Returns An `EnvVars` instance containing the computed environment variables.

generate(*scope='run'*)

Produces the launcher scripts activating the variables for the run context.

Parameters **scope** – Scope to be used.

7.4.5 conan.tools.build

Building

conan.tools.build.build_jobs()

build_jobs(*conanfile*)

Returns the number of CPUs available for parallel builds. It returns the configuration value for `tools.`

`build:jobs` if exists, otherwise, it defaults to the helper function `_cpu_count()`. `_cpu_count()` reads cgroup to detect the configured number of CPUs. Currently, there are two versions of cgroup available.

In the case of cgroup v1, if the data in cgroup is invalid, processor detection comes into play. Whenever processor detection is not enabled, `build_jobs()` will safely return 1.

In the case of cgroup v2, if no limit is set, processor detection is used. When the limit is set, the behavior is as described in cgroup v1.

Parameters **conanfile** – The current recipe object. Always use `self`.

Returns `int` with the number of jobs

conan.tools.build.cross_building()

cross_building(*conanfile=None*, *skip_x64_x86=False*)

Check if we are cross building comparing the *build* and *host* settings. Returns `True` in the case that we are cross-building.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **skip_x64_x86** – Do not consider cross building when building to 32 bits from 64 bits: x86_64 to x86, sparcv9 to sparc or ppc64 to ppc32

Returns `True` if we are cross building, `False` otherwise.

conan.tools.build.can_run()

can_run(*conanfile*)

Validates whether is possible to run a non-native app on the same architecture. It's an useful feature for the case your architecture can run more than one target. For instance, Mac M1 machines can run both *armv8* and *x86_64*.

Parameters **conanfile** – The current recipe object. Always use `self`.

Returns `bool` value from `tools.build.cross_building:can_run` if exists, otherwise, it returns `False` if we are cross-building, else, `True`.

Cppstd

conan.tools.build.check_min_cppstd()

check_min_cppstd(*conanfile*, *cppstd*, *gnu_extensions=False*)

Check if current cppstd fits the minimal version required.

In case the current cppstd doesn't fit the minimal version required by cppstd, a ConanInvalidConfiguration exception will be raised.

1. If settings.compiler.cppstd, the tool will use settings.compiler.cppstd to compare
2. It not settings.compiler.cppstd, the tool will use compiler to compare (reading the default from cppstd_default)
3. If not settings.compiler is present (not declared in settings) will raise because it cannot compare.
4. If can not detect the default cppstd for settings.compiler, a exception will be raised.

Parameters

- **conanfile** – The current recipe object. Always use `self`.

- **cppstd** – Minimal cppstd version required
- **gnu_extensions** – GNU extension is required (e.g gnu17)

conan.tools.build.check_max_cppstd()

check_max_cppstd(*conanfile*, *cppstd*, *gnu_extensions=False*)

Check if current cppstd fits the maximum version required.

In case the current cppstd doesn't fit the maximum version required by cppstd, a ConanInvalidConfiguration exception will be raised.

1. If settings.compiler.cppstd, the tool will use settings.compiler.cppstd to compare
2. It not settings.compiler.cppstd, the tool will use compiler to compare (reading the default from cppstd_default)
3. If not settings.compiler is present (not declared in settings) will raise because it cannot compare.
4. If can not detect the default cppstd for settings.compiler, a exception will be raised.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **cppstd** – Maximum cppstd version required
- **gnu_extensions** – GNU extension is required (e.g gnu17)

conan.tools.build.valid_min_cppstd()

valid_min_cppstd(*conanfile*, *cppstd*, *gnu_extensions=False*)

Validate if current cppstd fits the minimal version required.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **cppstd** – Minimal cppstd version required
- **gnu_extensions** – GNU extension is required (e.g gnu17). This option ONLY works on Linux.

Returns True, if current cppstd matches the required cppstd version. Otherwise, False.

conan.tools.build.valid_max_cppstd()

valid_max_cppstd(*conanfile*, *cppstd*, *gnu_extensions=False*)

Validate if current cppstd fits the maximum version required.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **cppstd** – Maximum cppstd version required
- **gnu_extensions** – GNU extension is required (e.g gnu17). This option ONLY works on Linux.

Returns True, if current cppstd matches the required cppstd version. Otherwise, False.

conan.tools.build.default_cppstd()

default_cppstd(*conanfile*, *compiler=None*, *compiler_version=None*)

Get the default `compiler.cppstd` for the "conanfile.settings.compiler" and "conanfile settings.compiler_version" or for the parameters "compiler" and "compiler_version" if specified.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **compiler** – Name of the compiler e.g. gcc
- **compiler_version** – Version of the compiler e.g. 12

Returns The default `compiler.cppstd` for the specified compiler

conan.tools.build.supported_cppstd()

supported_cppstd(*conanfile*, *compiler=None*, *compiler_version=None*)

Get the a list of supported `compiler.cppstd` for the "conanfile.settings.compiler" and "conanfile.settings.compiler_version" or for the parameters "compiler" and "compiler_version" if specified.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **compiler** – Name of the compiler e.g: gcc
- **compiler_version** – Version of the compiler e.g: 12

Returns a list of supported `cppstd` values.

7.4.6 conan.tools.files

conan.tools.files basic operations

conan.tools.files.copy()

copy(*conanfile*, *pattern*, *src*, *dst*, *keep_path=True*, *excludes=None*, *ignore_case=True*)

Copy the files matching the pattern (fnmatch) at the src folder to a dst folder.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **pattern** – (Required) An fnmatch file pattern of the files that should be copied. It must not start with `..` relative path or an exception will be raised.
- **src** – (Required) Source folder in which those files will be searched. This folder will be stripped from the dst parameter. E.g., lib/Debug/x86.
- **dst** – (Required) Destination local folder. It must be different from src value or an exception will be raised.
- **keep_path** – (Optional, defaulted to `True`) Means if you want to keep the relative path when you copy the files from the src folder to the dst one.
- **excludes** – (Optional, defaulted to `None`) A tuple/list of fnmatch patterns or even a single one to be excluded from the copy.
- **ignore_case** – (Optional, defaulted to `True`) If enabled, it will do a case-insensitive pattern matching. will do a case-insensitive pattern matching when `True`

Returns list of copied files

Usage:

```
def package(self):
    copy(self, "*.h", self.source_folder, os.path.join(self.package_folder, "include
↪"))
    copy(self, "*.lib", self.build_folder, os.path.join(self.package_folder, "lib"))
```

Note: The files that are **symlinks to files** or **symlinks to folders** with be treated like any other file, so they will only be copied if the specified pattern matches with the file.

At the destination folder, the symlinks will be created pointing to the exact same file or folder, absolute or relative, being the responsibility of the user to manipulate the symlink to, for example, transform the symlink into a relative path before copying it so it points to the destination folder.

Check *here* the reference of tools to manage symlinks.

---

conan.tools.files.load()

load(*conanfile*, *path*, *encoding='utf-8'*)

Utility function to load files in one line. It will manage the open and close of the file, and load binary encodings. Returns the content of the file.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **path** – Path to the file to read
- **encoding** – (Optional, Defaulted to `utf-8`): Specifies the input file text encoding.

Returns The contents of the file

Usage:

```
from conan.tools.files import load

content = load(self, "myfile.txt")
```

conan.tools.files.save()

save(*conanfile*, *path*, *content*, *append=False*, *encoding='utf-8'*)

Utility function to save files in one line. It will manage the open and close of the file and creating directories if necessary.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **path** – Path of the file to be created.
- **content** – Content (str or bytes) to be write to the file.
- **append** – (Optional, Defaulted to False): If `True` the contents will be appended to the existing one.
- **encoding** – (Optional, Defaulted to utf-8): Specifies the output file text encoding.

Usage:

```
from conan.tools.files import save

save(self, "path/to/otherfile.txt", "contents of the file")
```

conan.tools.files.rename()

rename(*conanfile*, *src*, *dst*)

Utility functions to rename a file or folder src to dst with retrying. `os.rename()` frequently raises "Access is denied" exception on Windows. This function renames file or folder using robocopy to avoid the exception on Windows.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **src** – Path to be renamed.
- **dst** – Path to be renamed to.

Usage:

```
from conan.tools.files import rename

def source(self):
    rename(self, "lib-sources-abe2h9fe", "sources")  # renaming a folder
```

conan.tools.files.replace_in_file()

replace_in_file(*conanfile*, *file_path*, *search*, *replace*, *strict=True*, *encoding='utf-8'*)

Replace a string `search` in the contents of the file `file_path` with the string replace.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **file_path** – File path of the file to perform the replacing.
- **search** – String you want to be replaced.
- **replace** – String to replace the searched string.
- **strict** – (Optional, Defaulted to `True`) If `True`, it raises an error if the searched string is not found, so nothing is actually replaced.
- **encoding** – (Optional, Defaulted to utf-8): Specifies the input and output files text encoding.

Usage:

```
from conan.tools.files import replace_in_file

replace_in_file(self, os.path.join(self.source_folder, "folder", "file.txt"), "foo",
↪"bar")
```

conan.tools.files.rm()

rm(*conanfile*, *pattern*, *folder*, *recursive=False*)

Utility functions to remove files matching a `pattern` in a `folder`.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **pattern** – Pattern that the files to be removed have to match (fnmatch).
- **folder** – Folder to search/remove the files.
- **recursive** – If `recursive` is specified it will search in the subfolders.

Usage:

```
from conan.tools.files import rm

rm(self, "*.tmp", self.build_folder, recursive=True)
```

conan.tools.files.mkdir()

mkdir(*conanfile*, *path*)

Utility functions to create a directory. The existence of the specified directory is checked, so mkdir() will do nothing if the directory already exists.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **path** – Path to the folder to be created.

Usage:

```
from conan.tools.files import mkdir

mkdir(self, "mydir") # Creates mydir if it does not already exist
mkdir(self, "mydir") # Does nothing
```

conan.tools.files.rmdir()

rmdir(*conanfile*, *path*)

Usage:

```
from conan.tools.files import rmdir

rmdir(self, "mydir") # Remove mydir if it exist
rmdir(self, "mydir") # Does nothing
```

conan.tools.files.chdir()

chdir(*conanfile*, *newdir*)

This is a context manager that allows to temporary change the current directory in your conanfile

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **newdir** – Directory path name to change the current directory.

Usage:

```
from conan.tools.files import chdir

def build(self):
```

(continues on next page)

(continued from previous page)

```
    with chdir(self, "./subdir"):
        do_something()
```

conan.tools.files.unzip()

This function extract different compressed formats (`.tar.gz`, `.tar`, `.tzb2`, `.tar.bz2`, `.tgz`, `.txz`, `tar.xz`, and `.zip`) into the given destination folder.

It also accepts gzipped files, with extension `.gz` (not matching any of the above), and it will unzip them into a file with the same name but without the extension, or to a filename defined by the `destination` argument.

```
from conan.tools.files import unzip

unzip(self, "myfile.zip")
# or to extract in "myfolder" sub-folder
unzip(self, "myfile.zip", "myfolder")
```

You can keep the permissions of the files using the `keep_permissions=True` parameter.

```
from conan.tools.files import unzip

unzip(self, "myfile.zip", "myfolder", keep_permissions=True)
```

Use the `pattern` argument if you want to filter specific files and paths to decompress from the archive.

```
from conan.tools.files import unzip

# Extract only files inside relative folder "small"
unzip(self, "bigfile.zip", pattern="small/*")
# Extract only txt files
unzip(self, "bigfile.zip", pattern="*.txt")
```

unzip(*conanfile*, *filename*, *destination='.'*, *keep_permissions=False*, *pattern=None*, *strip_root=False*)

Extract different compressed formats

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **filename** – Path to the compressed file.
- **destination** – (Optional, Defaulted to `.`) Destination folder (or file for .gz files)
- **keep_permissions** – (Optional, Defaulted to `False`) Keep the zip permissions. WARNING: Can be dangerous if the zip was not created in a NIX system, the bits could produce undefined permission schema. Use this option only if you are sure that the zip was created correctly.
- **pattern** – (Optional, Defaulted to `None`) Extract only paths matching the pattern. This should be a Unix shell-style wildcard, see fnmatch documentation for more details.
- **strip_root** – (Optional, Defaulted to False) If True, and all the unzipped contents are in a single folder it will flat the folder moving all the contents to the parent folder.

conan.tools.files.update_conandata()

This function reads the `conandata.yml` inside the exported folder in the conan cache, if it exists. If the `conandata.yml` does not exist, it will create it. Then, it updates the conandata dictionary with the provided

`data` one, which is updated recursively, prioritizing the `data` values, but keeping other existing ones. Finally the `conandata.yml` is saved in the same place.

This helper can only be used within the `export()` method, it can raise otherwise. One application is to capture in the `conandata.yml` the scm coordinates (like Git remote url and commit), to be able to recover it later in the `source()` method and have reproducible recipes that can build from sources without actually storing the sources in the recipe.

Usage:

```
from conan import ConanFile
from conan.tools.files import update_conandata

class Pkg(ConanFile):
    name = "pkg"
    version = "0.1"

    def export(self):
        # This is an example, doesn't make sense to have static data, instead you
        # could put the data directly in a conandata.yml file.
        # This would be useful for storing dynamic data, obtained at export() time␣
↪from elsewhere
        update_conandata(self, {"mydata": {"value": {"nested1": 123, "nested2": "some-
↪string"}}})

    def source(self):
        data = self.conan_data["sources"]["mydata"]
```

update_conandata(*conanfile*, *data*)

Tool to modify the `conandata.yml` once it is exported. It can be used, for example:

- To add additional data like the "commit" and "url" for the scm.
- To modify the contents cleaning the data that belong to other versions (different from the exported) to avoid changing the recipe revision when the changed data doesn't belong to the current version.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **data** – (Required) A dictionary (can be nested), of values to update

conan.tools.files.collect_libs()

collect_libs(*conanfile*, *folder=None*)

Returns a sorted list of library names from the libraries (files with extensions *.so*, *.lib*, *.a* and *.dylib*) located inside the `conanfile.cpp_info.libdirs` (by default) or the **folder** directory relative to the package folder. Useful to collect not inter-dependent libraries or with complex names like `libmylib-x86-debug-en.lib`.

For UNIX libraries staring with **lib**, like *libmath.a*, this tool will collect the library name **math**.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **(Optional, Defaulted to None)** (*folder*) – String indicating the subfolder name inside `conanfile.package_folder` where the library files are.

Returns A list with the library names

Warning: This tool collects the libraries searching directly inside the package folder and returns them in no specific order. If libraries are inter-dependent, then `package_info()` method should order them to achieve correct linking order.

Usage:

```
from conan.tools.files import collect_libs

def package_info(self):
    self.cpp_info.libdirs = ["lib", "other_libdir"]  # Default value is 'lib'
    self.cpp_info.libs = tools.collect_libs(self)
```

For UNIX libraries starting with **lib**, like *libmath.a*, this tool will collect the library name **math**. Regarding symlinks, this tool will keep only the "most generic" file among the resolved real file and all symlinks pointing to this real file. For example among files below, this tool will select *libmath.dylib* file and therefore only append *math* in the returned list:

```
-rwxr-xr-x libmath.1.0.0.dylib lrwxr-xr-x libmath.1.dylib -> libmath.1.0.0.dylib
lrwxr-xr-x libmath.dylib -> libmath.1.dylib
```

conan.tools.files downloads

conan.tools.files.get()

get(*conanfile*, *url*, *md5=None*, *sha1=None*, *sha256=None*, *destination='.'*, *filename=''*, *keep_permissions=False*, *pattern=None*, *verify=True*, *retry=None*, *retry_wait=None*, *auth=None*, *headers=None*, *strip_root=False*)

High level download and decompressing of a tgz, zip or other compressed format file. Just a high level wrapper for download, unzip, and remove the temporary zip file once unzipped. You can pass hash checking parameters: `md5`, `sha1`, `sha256`. All the specified algorithms will be checked. If any of them doesn't match, it will raise a `ConanException`.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **destination** – (Optional defaulted to `.`) Destination folder
- **filename** – (Optional defaulted to '') If provided, the saved file will have the specified name, otherwise it is deduced from the URL
- **url** – forwarded to `tools.file.download()`.
- **md5** – forwarded to `tools.file.download()`.
- **sha1** – forwarded to `tools.file.download()`.
- **sha256** – forwarded to `tools.file.download()`.
- **keep_permissions** – forwarded to `tools.file.unzip()`.
- **pattern** – forwarded to `tools.file.unzip()`.
- **verify** – forwarded to `tools.file.download()`.
- **retry** – forwarded to `tools.file.download()`.
- **retry_wait** – S forwarded to `tools.file.download()`.
- **auth** – forwarded to `tools.file.download()`.

- **headers** – forwarded to `tools.file.download()`.
- **strip_root** – forwarded to `tools.file.unzip()`.

conan.tools.files.ftp_download()

ftp_download(*conanfile*, *host*, *filename*, *login=''*, *password=''*)

Ftp download of a file. Retrieves a file from an FTP server. This doesn't support SSL, but you might implement it yourself using the standard Python FTP library.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **host** – IP or host of the FTP server
- **filename** – Path to the file to be downloaded
- **login** – Authentication login
- **password** – Authentication password

Usage:

```
from conan.tools.files import ftp_download

def source(self):
    ftp_download(self, 'ftp.debian.org', "debian/README")
    self.output.info(load("README"))
```

conan.tools.files.download()

download(*conanfile*, *url*, *filename*, *verify=True*, *retry=None*, *retry_wait=None*, *auth=None*, *headers=None*, *md5=None*, *sha1=None*, *sha256=None*)

Retrieves a file from a given URL into a file with a given filename. It uses certificates from a list of known verifiers for https downloads, but this can be optionally disabled.

You can pass hash checking parameters: `md5`, `sha1`, `sha256`. All the specified algorithms will be checked. If any of them doesn't match, the downloaded file will be removed and it will raise a `ConanException`.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **url** – URL to download. It can be a list, which only the first one will be downloaded, and the follow URLs will be used as mirror in case of download error. Files accessible in the local filesystem can be referenced with a URL starting with `file:///` followed by an absolute path to a file (where the third `/` implies `localhost`).
- **filename** – Name of the file to be created in the local storage
- **verify** – When False, disables https certificate validation
- **retry** – Number of retries in case of failure. Default is overridden by "tools.files.download:retry" conf
- **retry_wait** – Seconds to wait between download attempts. Default is overriden by "tools.files.download:retry_wait" conf.
- **auth** – A tuple of user and password to use HTTPBasic authentication
- **headers** – A dictionary with additional headers
- **md5** – MD5 hash code to check the downloaded file

- **sha1** – SHA-1 hash code to check the downloaded file
- **sha256** – SHA-256 hash code to check the downloaded file

Usage:

```
download(self, "http://someurl/somefile.zip", "myfilename.zip")

# to disable verification:
download(self, "http://someurl/somefile.zip", "myfilename.zip", verify=False)

# to retry the download 2 times waiting 5 seconds between them
download(self, "http://someurl/somefile.zip", "myfilename.zip", retry=2, retry_wait=5)

# Use https basic authentication
download(self, "http://someurl/somefile.zip", "myfilename.zip", auth=("user",
↪"password"))

# Pass some header
download(self, "http://someurl/somefile.zip", "myfilename.zip", headers={"Myheader":
↪"My value"})

# Download and check file checksum
download(self, "http://someurl/somefile.zip", "myfilename.zip", md5=
↪"e5d695597e9fa520209d1b41edad2a27")

# to add mirrors
download(self, ["https://ftp.gnu.org/gnu/gcc/gcc-9.3.0/gcc-9.3.0.tar.gz",
                "http://mirror.linux-ia64.org/gnu/gcc/releases/gcc-9.3.0/gcc-9.3.0.
↪tar.gz"],
                "gcc-9.3.0.tar.gz",
               sha256=
↪"5258a9b6afe9463c2e56b9e8355b1a4bee125ca828b8078f910303bc2ef91fa6")
```

conf

- `tools.files.download:retry`: number of retries in case some error occurs.
- `tools.files.download:retry_wait`: seconds to wait between retries.

conan.tools.files patches

conan.tools.files.patch()

patch(*conanfile*, *base_path=None*, *patch_file=None*, *patch_string=None*, *strip=0*, *fuzz=False*, ***kwargs*)

Applies a diff from file (patch_file) or string (patch_string) in the conanfile.source_folder directory. The folder containing the sources can be customized with the self.folders attribute in the layout(self) method.

Parameters

- **base_path** – The path is a relative path to conanfile.export_sources_folder unless an absolute path is provided.
- **patch_file** – Patch file that should be applied. The path is relative to the conanfile.source_folder unless an absolute path is provided.
- **patch_string** – Patch string that should be applied.
- **strip** – Number of folders to be stripped from the path.
- **output** – Stream object.

- **fuzz** – Should accept fuzzy patches.
- **kwargs** – Extra parameters that can be added and will contribute to output information

Usage:

```
from conan.tools.files import patch

def build(self):
    for it in self.conan_data.get("patches", {}).get(self.version, []):
        patch(self, **it)
```

conan.tools.files.apply_conandata_patches()

apply_conandata_patches(*conanfile*)

Applies patches stored in `conanfile.conan_data` (read from `conandata.yml` file). It will apply all the patches under `patches` entry that matches the given `conanfile.version`. If versions are not defined in `conandata.yml` it will apply all the patches directly under `patches` keyword.

The key entries will be passed as kwargs to the `patch` function.

Usage:

```
from conan.tools.files import apply_conandata_patches

def build(self):
    apply_conandata_patches(self)
```

Examples of `conandata.yml`:

```
patches:
- patch_file: "patches/0001-buildflatbuffers-cmake.patch"
- patch_file: "patches/0002-implicit-copy-constructor.patch"
  base_path: "subfolder"
  patch_type: backport
  patch_source: https://github.com/google/flatbuffers/pull/5650
  patch_description: Needed to build with modern clang compilers.
```

With different patches for different versions:

```
patches:
  "1.11.0":
    - patch_file: "patches/0001-buildflatbuffers-cmake.patch"
    - patch_file: "patches/0002-implicit-copy-constructor.patch"
      base_path: "subfolder"
      patch_type: backport
      patch_source: https://github.com/google/flatbuffers/pull/5650
      patch_description: Needed to build with modern clang compilers.
  "1.12.0":
    - patch_file: "patches/0001-buildflatbuffers-cmake.patch"
    - patch_string: |
        --- a/tests/misc-test.c
        +++ b/tests/misc-test.c
        @@ -1232,6 +1292,8 @@ main (int argc, char **argv)
              g_test_add_func ("/misc/pause-cancel", do_pause_cancel_test);
              g_test_add_data_func ("/misc/stealing/async", GINT_TO_POINTER (FALSE),␣
↪do_stealing_test);
              g_test_add_data_func ("/misc/stealing/sync", GINT_TO_POINTER (TRUE), do_
↪stealing_test);
```

(continues on next page)

(continued from previous page)

```
        +     g_test_add_func ("/misc/response/informational/content-length", do_
↪response_informational_content_length_test);
        +
        ret = g_test_run ();
    - patch_file: "patches/0003-fix-content-length-calculation.patch"
```

conan.tools.files checksums

conan.tools.files.check_md5()

check_md5 (*conanfile*, *file_path*, *signature*)

Check that the specified `md5sum` of the `file_path` matches with `signature`. If doesn't match it will raise a `ConanException`.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **file_path** – Path of the file to check.
- **signature** – Expected md5sum.

conan.tools.files.check_sha1()

check_sha1 (*conanfile*, *file_path*, *signature*)

Check that the specified `sha1` of the `file_path` matches with signature. If doesn't match it will raise a `ConanException`.

Parameters

- **conanfile** – Conanfile object.
- **file_path** – Path of the file to check.
- **signature** – Expected sha1sum

conan.tools.files.check_sha256()

check_sha256 (*conanfile*, *file_path*, *signature*)

Check that the specified `sha256` of the `file_path` matches with signature. If doesn't match it will raise a `ConanException`.

Parameters

- **conanfile** – Conanfile object.
- **file_path** – Path of the file to check.
- **signature** – Expected sha256sum

conan.tools.files.symlinks

conan.tools.files.symlinks.absolute_to_relative_symlinks()

absolute_to_relative_symlinks (*conanfile*, *base_folder*)

Convert the symlinks with absolute paths into relative ones if they are pointing to a file or directory inside the `base_folder`. Any absolute symlink pointing outside the `base_folder` will be ignored.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **base_folder** – Folder to be scanned.

conan.tools.files.symlinks.remove_external_symlinks()

remove_external_symlinks(*conanfile*, *base_folder*)

Remove the symlinks to files that point outside the `base_folder`, no matter if relative or absolute.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **base_folder** – Folder to be scanned.

conan.tools.files.symlinks.remove_broken_symlinks()

remove_broken_symlinks(*conanfile*, *base_folder=None*)

Remove the broken symlinks, no matter if relative or absolute.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **base_folder** – Folder to be scanned.

conan.tools.files AutoPackager

The `AutoPackager` together with the layout() feature, allow to automatically package the files following the declared information in the `layout()` method:

It will copy:

- Files from `self.cpp.local.includedirs` to `self.cpp.package.includedirs`
- Files from `self.cpp.local.libdirs` to `self.cpp.package.libdirs`
- Files from `self.cpp.local.bindirs` to `self.cpp.package.bindirs`
- Files from `self.cpp.local.srcdirs` to `self.cpp.package.srcdirs`
- Files from `self.cpp.local.builddirs` to `self.cpp.package.builddirs`
- Files from `self.cpp.local.resdirs` to `self.cpp.package.resdirs`
- Files from `self.cpp.local.frameworkdirs` to `self.cpp.package.frameworkdirs`

The patterns of the files to be copied can be defined with the *.patterns* property of the `AutoPackager` instance. The default patterns are:

```
packager = AutoPackager(self)
packager.patterns.include == ["*.h", "*.hpp", "*.hxx"]
packager.patterns.lib == ["*.so", "*.so.*", "*.a", "*.lib", "*.dylib"]
packager.patterns.bin == ["*.exe", "*.dll"]
packager.patterns.src == []
packager.patterns.build == []
packager.patterns.res == []
packager.patterns.framework == []
```

Usage:

```
from conan import ConanFile
from conan.tools.files import AutoPackager

class Pkg(ConanFile):

    def layout(self):
        ...

    def package(self):
        packager = AutoPackager(self)
        packager.patterns.include = ["*.hpp", "*.h", "include3.h"]
        packager.patterns.lib = ["*.a"]
        packager.patterns.bin = ["*.exe"]
        packager.patterns.src = ["*.cpp"]
        packager.patterns.framework = ["sframe*", "bframe*"]
        packager.run()
```

class AutoPackager(*conanfile*)

Parameters **conanfile** – The current recipe object. Always use `self`.

7.4.7 conan.tools.meson

MesonToolchain

The `MesonToolchain` is the toolchain generator for Meson and it can be used in the `generate()` method as follows:

```
from conan import ConanFile
from conan.tools.meson import MesonToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    options = {"shared": [True, False]}
    default_options = {"shared": False}

    def generate(self):
        tc = MesonToolchain(self)
        tc.preprocessor_definitions["MYDEFINE"] = "MYDEF_VALUE"
        tc.generate()
```

Important: When your recipe has dependencies `MesonToolchain` only works with the `PkgConfigDeps` generator. Please, do not use other generators, as they can have overlapping definitions that can conflict.

Generated files

The `MesonToolchain` generates the following files after a **conan install** (or when building the package in the cache) with the information provided in the `generate()` method as well as information translated from the current `settings`, `conf`, etc.:

- *conan_meson_native.ini*: if doing a native build.
- *conan_meson_cross.ini*: if doing a cross-build (*conan.tools.build*).

conan_meson_native.ini

This file contains the definitions of all the Meson properties related to the Conan options and settings for the current package, platform, etc. This includes but is not limited to the following:

- Detection of `default_library` from Conan settings.
  - Based on existence/value of an option named `shared`.
- Detection of `buildtype` from Conan settings.
- Definition of the C++ standard as necessary.
- The Visual Studio runtime (`b_vscrt`), obtained from Conan input settings.

conan_meson_cross.ini

This file contains the same information as the previous *conan_meson_native.ini*, but with additional information to describe host, target, and build machines (such as the processor architecture).

Check out the meson documentation for more details on native and cross files:

- Machine files
- Native environments
- Cross compilation

Default directories

`MesonToolchain` manages some of the directories used by Meson. These are variables declared under the `[project options]` section of the files *conan_meson_native.ini* and *conan_meson_cross.ini* (see more information about Meson directories):

`bindir`: value coming from `self.cpp.package.bindirs`. Defaulted to None. `sbindir`: value coming from `self.cpp.package.bindirs`. Defaulted to None. `libexecdir`: value coming from `self.cpp.package.bindirs`. Defaulted to None. `datadir`: value coming from `self.cpp.package.resdirs`. Defaulted to None. `localedir`: value coming from `self.cpp.package.resdirs`. Defaulted to None. `mandir`: value coming from `self.cpp.package.resdirs`. Defaulted to None. `infodir`: value coming from `self.cpp.package.resdirs`. Defaulted to None. `includedir`: value coming from `self.cpp.package.includedirs`. Defaulted to None. `libdir`: value coming from `self.cpp.package.libdirs`. Defaulted to None.

Notice that it needs a `layout` to be able to initialize those `self.cpp.package.xxxxx` variables. For instance:

```
from conan import ConanFile
from conan.tools.meson import MesonToolchain
class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    def layout(self):
        self.folders.build = "build"
        self.cpp.package.resdirs = ["res"]
    def generate(self):
        tc = MesonToolchain(self)
        self.output.info(tc.project_options["datadir"])  # Will print '["res"]'
        tc.generate()
```

Note: All of them are saved only if they have any value. If the values are‘‘None‘‘, they won’t be mentioned in `[project options]` section.

Customization

Attributes

definitions

This attribute allows defining Meson project options:

```
def generate(self):
    tc = MesonToolchain(self)
    tc.definitions["MYVAR"] = "MyValue"
    tc.generate()
```

This is translated to:

- One project options definition for `MYVAR` in `conan_meson_native.ini` or `conan_meson_cross.ini` file.

preprocessor_definitions

This attribute allows defining compiler preprocessor definitions, for multiple configurations (Debug, Release, etc).

```
def generate(self):
    tc = MesonToolchain(self)
    tc.preprocessor_definitions["MYDEF"] = "MyValue"
    tc.generate()
```

This is translated to:

- One preprocessor definition for `MYDEF` in `conan_meson_native.ini` or `conan_meson_cross.ini` file.

conf

`MesonToolchain` is affected by these `[conf]` variables:

- `tools.meson.mesontoolchain:backend`. the meson backend to use. Possible values: `ninja`, `vs`, `vs2010`, `vs2015`, `vs2017`, `vs2019`, `xcode`.
- `tools.apple:sdk_path` argument for SDK path in case of Apple cross-compilation. It is used as value of the flag `-isysroot`.
- `tools.android:ndk_path` argument for NDK path in case of Android cross-compilation. It is used to get some binaries like `c`, `cpp` and `ar` used in `[binaries]` section from *conan_meson_cross.ini*.
- `tools.build:cxxflags` list of extra C++ flags that is used by `cpp_args`.
- `tools.build:cflags` list of extra of pure C flags that is used by `c_args`.
- `tools.build:sharedlinkflags` list of extra linker flags that is used by `c_link_args` and `cpp_link_args`.
- `tools.build:exelinkflags` list of extra linker flags that is used by `c_link_args` and `cpp_link_args`.

Cross-building for Apple and Android

The `MesonToolchain` adds all the flags required to cross-compile for Apple (MacOS M1, iOS, etc.) and Android.

Apple

It adds link flags `-arch XXX`, `-isysroot [SDK_PATH]` and the minimum deployment target flag, e.g., `-mios-version-min=8.0` to the `MesonToolchain` `c_args`, `c_link_args`, `cpp_args`, and `cpp_link_args` attributes, given the Conan settings for any Apple OS (iOS, watchOS, etc.) and the `tools.apple:sdk_path` configuration value like it's shown in this example of host profile:

Listing 22: **ios_host_profile**

```
[settings]
os = iOS
os.version = 10.0
os.sdk = iphoneos
arch = armv8
compiler = apple-clang
compiler.version = 12.0
compiler.libcxx = libc++

[conf]
tools.apple:sdk_path=/my/path/to/iPhoneOS.sdk
```

Objective-C arguments

In Apple OS's there are also specific Objective-C/Objective-C++ arguments: `objc`, `objcpp`, `objc_args`, `objc_link_args`, `objcpp_args`, and `objcpp_link_args`, as public attributes of the `MesonToolchain` class, where the variables `objc` and `objcpp` are initialized as `clang` and `clang++` respectively by default.

Android

It initializes the `MesonToolchain` `c`, `cpp`, and `ar` attributes, which are needed to cross-compile for Android, given the Conan settings for Android and the `tools.android:ndk_path` configuration value like it's shown in this example of host profile:

Listing 23: **android_host_profile**

```
[settings]
os = Android
os.api_level = 21
arch = armv8

[conf]
tools.android:ndk_path=/my/path/to/NDK
```

Read more

- *Getting started with Meson*

Reference

class MesonToolchain(*conanfile*, *backend=None*)

MesonToolchain generator

Parameters

- **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.
- **backend** – `str` `backend` Meson variable value. By default, `ninja`.

properties = None

Dict-like object that defines Meson''properties'' with `key=value` format

project_options = None
Dict-like object that defines Meson `project options` with `key=value` format

preprocessor_definitions = None
Dict-like object that defines Meson `preprocessor definitions`

pkg_config_path = None
Defines the Meson `pkg_config_path` variable

cross_build = None
Dict-like object with the build, host, and target as the Meson machine context

c = None
Defines the Meson `c` variable. Defaulted to `CC` build environment value

cpp = None
Defines the Meson `cpp` variable. Defaulted to `CXX` build environment value

c_ld = None
Defines the Meson `c_ld` variable. Defaulted to `CC_LD` or `LD` build environment value

cpp_ld = None
Defines the Meson `cpp_ld` variable. Defaulted to `CXX_LD` or `LD` build environment value

ar = None
Defines the Meson `ar` variable. Defaulted to `AR` build environment value

strip = None
Defines the Meson `strip` variable. Defaulted to `STRIP` build environment value

as_ = None
Defines the Meson `as` variable. Defaulted to `AS` build environment value

windres = None
Defines the Meson `windres` variable. Defaulted to `WINDRES` build environment value

pkgconfig = None
Defines the Meson `pkgconfig` variable. Defaulted to `PKG_CONFIG` build environment value

c_args = None
Defines the Meson `c_args` variable. Defaulted to `CFLAGS` build environment value

c_link_args = None
Defines the Meson `c_link_args` variable. Defaulted to `LDFLAGS` build environment value

cpp_args = None
Defines the Meson `cpp_args` variable. Defaulted to `CXXFLAGS` build environment value

cpp_link_args = None
Defines the Meson `cpp_link_args` variable. Defaulted to `LDFLAGS` build environment value

apple_arch_flag = None
Apple arch flag as a list, e.g., `["-arch", "i386"]`

apple_isysroot_flag = None
Apple sysroot flag as a list, e.g., `["-isysroot", "./Platforms/MacOSX.platform"]`

apple_min_version_flag = None
Apple minimum binary version flag as a list, e.g., `["-mios-version-min", "10.8"]`

objc = None
Defines the Meson `objc` variable. Defaulted to `None`, if if any Apple OS `clang`

objcpp = None
Defines the Meson `objcpp` variable. Defaulted to `None`, if if any Apple OS `clang++`

objc_args = None
Defines the Meson `objc_args` variable. Defaulted to `OBJCFLAGS` build environment value

objc_link_args = None
Defines the Meson `objc_link_args` variable. Defaulted to `LDFLAGS` build environment value

objcpp_args = None
Defines the Meson `objcpp_args` variable. Defaulted to `OBJCXXFLAGS` build environment value

objcpp_link_args = None
Defines the Meson `objcpp_link_args` variable. Defaulted to `LDFLAGS` build environment value

generate()
Creates a `conan_meson_native.ini` (if native builds) or a `conan_meson_cross.ini` (if cross builds) with the proper content. If Windows OS, it will be created a `conanvcvars.bat` as well.

MesonDeps

MesonToolchain normally works together with *PkgConfigDeps* to manage all the dependencies, but sometimes we need to gather some flags coming from `Autotools` tool so that's what `MesonDeps` is meant for. In other words, it is typically used when Meson cannot find a dependency using the already known detection mechanisms like: *pkg-config*, *cmake*, *config-tool*, etc. For instance, if we'd have these lines in your *meson.build* file, you might need `MesonDeps` to find that dependency and inject the correct flags to the compiler:

Listing 24: **meson.build**

```
project('tutorial', 'cpp')
cxx = meson.get_compiler('cpp')
mylib = cxx.find_library('mylib', required: true)
executable('app', 'main.cpp', dependencies: mylib)
```

In a nutshell, the `MesonDeps` generator is the dependencies generator for Meson and GNU flags. It creates a *conan_meson_deps_flags.ini* file with all those flags collected by each dependency.

Important: At this moment, this generator must be used along with `MesonToolchain` one to make it work correctly.

Important: This class will require very soon to define both the "host" and "build" profiles. It is very recommended to start defining both profiles immediately to avoid future breaking. Furthermore, some features, like trying to cross-compile might not work at all if the "build" profile is not provided.

The `MesonDeps` generator can be used by name in conanfiles:

Listing 25: conanfile.py

```
class Pkg(ConanFile):
    generators = "MesonDeps"
```

Listing 26: conanfile.txt

```
[generators]
MesonDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.meson import MesonDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = MesonDeps(self)
        tc.generate()
```

The `MesonDeps` generates after a `conan install` command a *conan_meson_deps_flags.ini* file:

```
[constants]
deps_c_args = []
deps_c_link_args = []
deps_cpp_args = []
deps_cpp_link_args = []
```

This generator defines a Meson constants: `deps_c_args`, `deps_c_link_args`, `deps_cpp_args`, `deps_cpp_link_args`, that accumulate all dependencies information, including transitive dependencies, with flags like `-I<path>`, `-L<path>`, etc.

Important: Those variables are added automatically as part of the built-in options declared by `MesonToolchain` generator: `c_args`, `c_link_args`, `cpp_args`, `cpp_link_args`.

Note: For now, only the `requires` information is generated, the `tool_requires` one is not managed by this generator yet.

Attributes

- `c_args`, `c_link_args`, `cpp_args`, `cpp_link_args`: list of flags that accumulate all dependencies information. Each one is saved as `deps_c_args`, `deps_c_link_args`, `deps_cpp_args`, and `deps_cpp_link_args`, respectively in the *conan_meson_deps_flags.ini* file.

```
from conan import ConanFile
from conan.tools.meson import MesonDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = MesonDeps(self)
        tc.c_args.append("-val1")
        tc.c_link_args.append("-val2")
        tc.cpp_args.append("-val3")
```

(continues on next page)

(continued from previous page)

```
        tc.cpp_link_args.append("-val4")
        tc.generate()
```

Reference

class MesonDeps(*conanfile*)
Generator that manages all the GNU flags from all the dependencies

Meson

The `Meson()` build helper is intended to be used in the `build()` and `package()` methods, to call Meson commands automatically.

```
from conan import ConanFile
from conan.tools.meson import Meson

class PkgConan(ConanFile):

    def build(self):
        meson = Meson(self)
        meson.configure()
        meson.build()

    def package(self):
        meson = Meson(self)
        meson.install()
```

Reference

class Meson(*conanfile*)
This class calls Meson commands when a package is being built. Notice that this one should be used together with the `MesonToolchain` generator.

Parameters **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.

configure(*reconfigure=False*)
Runs `meson setup [FILE] "BUILD_FOLDER" "SOURCE_FOLDER" [-Dprefix=PACKAGE_FOLDER]` command, where `FILE` could be `--native-file conan_meson_native.ini` (if native builds) or `--cross-file conan_meson_cross.ini` (if cross builds).

Parameters **reconfigure** – `bool` value that adds `--reconfigure` param to the final command.

build(*target=None*)
Runs `meson compile -C . -j[N_JOBS] [TARGET]` in the build folder. You can specify `N_JOBS` through the configuration line `tools.build:jobs=N_JOBS` in your profile `[conf]` section.

Parameters **target** – `str` Specifies the target to be executed.

install()
Runs `meson install -C "."` in the build folder. Notice that it will execute `self.configure(reconfigure=True)` at first.

test()
Runs `meson test -v -C "."` in the build folder.

7.4.8 conan.tools.system

conan.tools.system.package_manager

The tools under *conan.tools.system.package_manager* are wrappers around some of the most popular system package managers for different platforms. You can use them to invoke system package managers in recipes and perform the most typical operations, like installing a package, updating the package manager database or checking if a package is installed. By default, when you invoke them they will not try to install anything on the system, to change this behavior you can set the value of the `tools.system.package_manager:mode` *configuration*.

You can use these tools inside the `system_requirements()` method of your recipe, like:

Listing 27: conanfile.py

```
from conan.tools.system.package_manager import Apt, Yum, PacMan, Zypper

def system_requirements(self):
    # depending on the platform or the tools.system.package_manager:tool configuration
    # only one of these will be executed
    Apt(self).install(["libgl-dev"])
    Yum(self).install(["libglvnd-devel"])
    PacMan(self).install(["libglvnd"])
    Zypper(self).install(["Mesa-libGL-devel"])
```

Conan will automatically choose which package manager to use by looking at the Operating System name. In the example above, if we are running on Ubuntu Linux, Conan will ignore all the calls except for the `Apt()` one and will only try to install the packages using the `apt-get` tool. Conan uses the following mapping by default:

- *Apt* for **Linux** with distribution names: *ubuntu*, *debian* or *raspbian*
- *Yum* for **Linux** with distribution names: *pidora*, *scientific*, *xenserver*, *amazon*, *oracle*, *amzn*, *almalinux* or *rocky*
- *Dnf* for **Linux** with distribution names: *fedora*, *rhel*, *centos*, *mageia*
- *Brew* for **macOS**
- *PacMan* for **Linux** with distribution names: *arch*, *manjaro* and when using **Windows** with *msys2*
- *Chocolatey* for **Windows**
- *Zypper* for **Linux** with distribution names: *opensuse*, *sles*
- *Pkg* for **Linux** with distribution names: *freebsd*
- *PkgUtil* for **Solaris**

You can override this default mapping and set the package manager tool you want to use by default setting the configuration property *tools.system.package_manager:tool*.

Methods available for system package manager tools

All these wrappers share three methods that represent the most common operations with a system package manager. They take the same form for all of the package managers except for *Apt* that also accepts the *recommends* argument for the *install method*.

- `install(self, packages, update=False, check=False)`: try to install the list of packages passed as a parameter. If the parameter `check` is `True` it will check if those packages are already installed before installing them. If the parameter `update` is `True` it will try to update the package man-

ager database before checking and installing. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*. It will return the return code of the executed commands.

- `install_substitutes(packages_substitutes, update=False, check=True)`: try to install the list of lists of substitutes packages passed as a parameter, e.g., `[["pkg1", "pkg2"], ["pkg3"]]`. It succeeds if one of the substitutes list is completely installed, so it's intended to be used when you have different packages for different distros. Internally, it's calling the previous `install(packages, update=update, check=check)` method, so `update` and `check` have the same purpose as above.
- `update()` update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.
- `check(packages)` check if the list of packages passed as parameter are already installed. It will return a list with the packages that are missing.

Configuration properties that affect how system package managers are invoked

As explained above there are several `[conf]` that affect how these tools are invoked:

- `tools.system.package_manager:tool`: to choose which package manager tool you want to use by default: `"apt-get"`, `"yum"`, `"dnf"`, `"brew"`, `"pacman"`, `"choco"`, `"zypper"`, `"pkg"` or `"pkgutil"`
- `tools.system.package_manager:mode`: mode to use when invoking the package manager tool. There are two possible values:
  - `"check"`: it will just check for missing packages at most and will not try to update the package manager database or install any packages in any case. This is the default value.
  - `"install"`: it will allow Conan to perform update or install operations.
- `tools.system.package_manager:sudo`: Use *sudo* when invoking the package manager tools in Linux (`False` by default)
- `tools.system.package_manager:sudo_askpass`: Use the `-A` argument if using sudo in Linux to invoke the system package manager (`False` by default)

There are some specific arguments for each of these tools. Here is the complete reference:

conan.tools.system.package_manager.Apt

Will invoke the *apt-get* command. Enabled by default for **Linux** with distribution names: *ubuntu* and *debian*.

Reference

class Apt(*conanfile*, *arch_names=None*)

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **arch_names** – This argument maps the Conan architecture setting with the package manager tool architecture names. It is `None` by default, which means that it will use a default mapping for the most common architectures. For example, if you are using `x86_64` Conan architecture setting, it will map this value to `amd64` for *Apt* and try to install the `<package_name>:amd64` package.

install(*packages*, *update=False*, *check=False*, *recommends=False*)

Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.
- **recommends** – if the parameter `recommends` is `False` it will add the `'--no-install-recommends'` argument to the *apt-get* command call.

Returns the return code of the executed apt command.

check(**args*, ***kwargs*)

Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install_substitutes(**args*, ***kwargs*)

Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)

Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

You can pass the `arch_names` argument to override the default Conan mapping like this:

Listing 28: conanfile.py

```
...
def system_requirements(self):
    apt = Apt(self, arch_names={"<conan_arch_setting>": "apt_arch_setting"})
    apt.install(["libgl-dev"])
```

The default mapping that Conan uses for *APT* packages architecture is:

```
self._arch_names = {"x86_64": "x86_64",
                    "x86": "i?86",
                    "ppc32": "powerpc",
                    "ppc64le": "ppc64le",
                    "armv7": "armv7",
```

(continues on next page)

(continued from previous page)

```
                    "armv7hf": "armv7hl",
                    "armv8": "aarch64",
                    "s390x": "s390x"} if arch_names is None else arch_names
```

conan.tools.system.package_manager.Yum

Will invoke the *yum* command. Enabled by default for **Linux** with distribution names: *pidora*, *scientific*, *xenserver*, *amazon*, *oracle*, *amzn* and *almalinux*.

Reference

class Yum(*conanfile*, *arch_names=None*)

Parameters

- **conanfile** – the current recipe object. Always use `self`.
- **arch_names** – this argument maps the Conan architecture setting with the package manager tool architecture names. It is `None` by default, which means that it will use a default mapping for the most common architectures. For example, if you are using `x86` Conan architecture setting, it will map this value to `i?86` for *Yum* and try to install the `<package_name>.i?86` package.

check(**args*, ***kwargs*)

Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install(**args*, ***kwargs*)

Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes(**args*, ***kwargs*)

Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.

- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)
Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

The default mapping Conan uses for *Yum* packages architecture is:

```
self._arch_names = {"x86_64": "x86_64",
                    "x86": "i?86",
                    "ppc32": "powerpc",
                    "ppc64le": "ppc64le",
                    "armv7": "armv7",
                    "armv7hf": "armv7hl",
                    "armv8": "aarch64",
                    "s390x": "s390x"} if arch_names is None else arch_names
```

conan.tools.system.package_manager.Dnf

Will invoke the *dnf* command. Enabled by default for **Linux** with distribution names: *fedora*, *rhel*, *centos* and *mageia*. This tool has exactly the same default values, constructor and methods than the *Yum* tool.

conan.tools.system.package_manager.PacMan

Will invoke the *pacman* command. Enabled by default for **Linux** with distribution names: *arch*, *manjaro* and when using **Windows** with *msys2*

Reference

class PacMan(*conanfile*, *arch_names=None*)

Parameters

- **conanfile** – the current recipe object. Always use `self`.
- **arch_names** – this argument maps the Conan architecture setting with the package manager tool architecture names. It is `None` by default, which means that it will use a default mapping for the most common architectures. If you are using `x86` Conan architecture setting, it will map this value to `lib32` for *PacMan* and try to install the `<package_name>-lib32` package.

check(**args*, ***kwargs*)
Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install(**args*, ***kwargs*)
Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.

- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes(**args*, ***kwargs*)
Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)
Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

The default mapping Conan uses for *PacMan* packages architecture is:

```
self._arch_names = {"x86": "lib32"} if arch_names is None else arch_names
```

conan.tools.system.package_manager.Zypper

Will invoke the *zypper* command. Enabled by default for **Linux** with distribution names: *opensuse*, *sles*.

Reference

class Zypper(*conanfile*)

Parameters **conanfile** – The current recipe object. Always use `self`.

check(**args*, ***kwargs*)
Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install(**args*, ***kwargs*)
Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes(**args*, ***kwargs*)
Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)
Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

conan.tools.system.package_manager.Brew

Will invoke the *brew* command. Enabled by default for **macOS**.

Reference

class Brew(*conanfile*)

Parameters **conanfile** – The current recipe object. Always use `self`.

check(**args*, ***kwargs*)
Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install(**args*, ***kwargs*)
Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes(**args*, ***kwargs*)
Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro

version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)

Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

conan.tools.system.package_manager.Pkg

Will invoke the *pkg* command. Enabled by default for **Linux** with distribution names: *freebsd*.

Reference

class Pkg(*conanfile*)

Parameters **conanfile** – The current recipe object. Always use `self`.

check(**args*, ***kwargs*)

Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install(**args*, ***kwargs*)

Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes(**args*, ***kwargs*)

Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)
Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

conan.tools.system.package_manager.PkgUtil

Will invoke the *pkgutil* command. Enabled by default for **Solaris**.

Reference

class PkgUtil(*conanfile*)

Parameters **conanfile** – The current recipe object. Always use `self`.

check(**args*, ***kwargs*)
Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install(**args*, ***kwargs*)
Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes(**args*, ***kwargs*)
Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)
Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

conan.tools.system.package_manager.Chocolatey

Will invoke the *choco* command. Enabled by default for **Windows**.

Reference

class Chocolatey(*conanfile*)

Parameters **conanfile** – The current recipe object. Always use `self`.

check(**args*, ***kwargs*)
Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install(**args*, ***kwargs*)
Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes(**args*, ***kwargs*)
Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.

- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)
Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

7.4.9 conan.tools.microsoft

MSBuild

The `MSBuild` build helper is a wrapper around the command line invocation of MSBuild. It abstracts the calls like `msbuild "MyProject.sln" /p:Configuration=<conf> /p:Platform=<platform>` into Python method ones.

This helper can be used like:

```
from conan import ConanFile
from conan.tools.microsoft import MSBuild

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build("MyProject.sln")
```

The `MSBuild.build()` method internally implements a call to `msbuild` like:

```
$ <vcvars-cmd> && msbuild "MyProject.sln" /p:Configuration=<configuration> /
↪p:Platform=<platform>
```

Where:

- `<vcvars-cmd>` calls the Visual Studio prompt that matches the current recipe `settings`.
- `configuration`, typically Release, Debug, which will be obtained from `settings.build_type` but this can be customized with the `build_type` attribute.
- `<platform>` is the architecture, a mapping from the `settings.arch` to the common 'x86', 'x64', 'ARM', 'ARM64'. This can be customized with the `platform` attribute.

Customization

attributes

You can customize the following attributes in case you need to change them:

- **build_type** (default `settings.build_type`): Value for the `/p:Configuration`.
- **platform** (default based on `settings.arch` to select one of these values: `('x86', 'x64', 'ARM', 'ARM64')`: Value for the `/p:Platform`.

Example:

```
from conan import ConanFile
from conan.tools.microsoft import MSBuild
class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    def build(self):
        msbuild = MSBuild(self)
        msbuild.build_type = "MyRelease"
        msbuild.platform = "MyPlatform"
        msbuild.build("MyProject.sln")
```

conf

`MSBuild` is affected by these `[conf]` variables:

- `tools.microsoft.msbuild:verbosity` accepts one of `"Quiet"`, `"Minimal"`, `"Normal"`, `"Detailed"`, `"Diagnostic"` to be passed to the `MSBuild.build()` call as `msbuild .... /verbosity:XXX`.
- `tools.microsoft.msbuild:max_cpu_count` maximum number of CPUs to be passed to the `MSBuild.build()` call as `msbuild .... /m:N`.

Reference

class MSBuild(*conanfile*)

MSBuild build helper class

Parameters **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.

command(*sln*, *targets=None*)

Gets the `msbuild` command line. For instance, **msbuild "MyProject.sln" /p:Configuration=<conf> /p:Platform=<platform>**.

Parameters

- **sln** – `str` name of Visual Studio `*.sln` file
- **targets** – `targets` is an optional argument, defaults to `None`, and otherwise it is a list of targets to build

Returns `str` msbuild command line.

build(*sln*, *targets=None*)

Runs the `msbuild` command line obtained from `self.command(sln)`.

Parameters

- **sln** – `str` name of Visual Studio `*.sln` file
- **targets** – `targets` is an optional argument, defaults to `None`, and otherwise it is a list of targets to build

MSBuildDeps

The `MSBuildDeps` is the dependency information generator for Microsoft MSBuild build system. It will generate multiple *xxxx.props* properties files, one per dependency of a package, to be used by consumers using MSBuild or Visual Studio, just adding the generated properties files to the solution and projects.

The `MSBuildDeps` generator can be used by name in conanfiles:

Listing 29: conanfile.py

```
class Pkg(ConanFile):
    generators = "MSBuildDeps"
```

Listing 30: conanfile.txt

```
[generators]
MSBuildDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 31: conanfile.py

```
from conan import ConanFile
from conan.tools.microsoft import MSBuildDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = MSBuildDeps(self)
        ms.generate()
```

Generated files

The `MSBuildDeps` generator is a multi-configuration generator, and generates different files for any different Debug/Release configuration. For instance, running these commands:

```
$ conan install .  # default is Release
$ conan install . -s build_type=Debug
```

It generates the next files:

- *conan_zlib_vars_release_x64.props*: `Conanzlibxxxx` variables definitions for the `zlib` dependency, Release config, like `ConanzlibIncludeDirs`, `ConanzlibLibs`, etc.
- *conan_zlib_vars_debug_x64.props*: Same `Conanzlib``variables for ``zlib` dependency, Debug config
- *conan_zlib_release_x64.props*: Activation of `Conanzlibxxxx` variables in the current build as standard C/C++ build configuration, Release config. This file contains also the transitive dependencies definitions.
- *conan_zlib_debug_x64.props*: Same activation of `Conanzlibxxxx` variables, Debug config, also inclusion of transitive dependencies.
- *conan_zlib.props*: Properties file for `zlib`. It conditionally includes, depending on the configuration, one of the two immediately above Release/Debug properties files.
- Same 5 files are generated for every dependency in the graph, in this case `conan_bzip.props` too, which conditionally includes the Release/Debug bzip properties files.
- *conandeps.props*: Properties files that includes all direct dependencies, for this case `conan_zlib.props` and `conan_bzip2.props`

Add the *conandeps.props* to your solution project files if you want to depend on all the declared dependencies. For single project solutions, this is probably the way to go. For multi-project solutions, you might be more efficient

and add properties files per project. You could add *conan_zlib.props* properties to "project1" in the solution and *conan_bzip2.props* to "project2" in the solution for example.

The above files are generated when the package doesn't have components. If the package has defined components, the following files will be generated:

- *conan_pkgname_compname_vars_release_x64.props*: Definition of variables for the component `compname` of the package `pkgname`
- *conan_pkgname_compname_release_x64.props*: Activation of the above variables into VS effective variables to be used in the build
- *conan_pkgname_compname.props*: Properties file for component `compname` of package `pkgname`. It conditionally includes, depending on the configuration, the specific activation property files.
- *conan_pkgname.props*: Properties file for package `pkgname`. It includes and aggregates all the components of the package.
- *conandeps.props*: Same as above, aggregates all the direct dependencies property files for the packages (like `conan_pkgname.props`)

If your project depends only on certain components, the specific `conan_pkgname_compname.props` files can be added to the project instead of the global or the package ones.

Requirement traits support

The above generated files, more specifically the files containing the variables (`conan_pkgname_vars_release_x64.props`/`conan_pkgname_compname_vars_release_x64.props`), will not contain all the information if the requirement traits have excluded them. For example, by default, the `includedirs` of transitive dependencies will be empty, as those headers shouldn't be included by the user unless a specific `requires` to that package is defined.

Configurations

If your Visual Studio project defines custom configurations, like `ReleaseShared`, or `MyCustomConfig`, it is possible to define it into the `MSBuildDeps` generator, so different project configurations can use different set of dependencies. Let's say that our current project can be built as a shared library, with the custom configuration `ReleaseShared`, and the package also controls this with the `shared` option:

```
from conan import ConanFile
from conan.tools.microsoft import MSBuildDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    options = {"shared": [True, False]}
    default_options = {"shared": False}
    requires = "zlib/1.2.11"

    def generate(self):
        ms = MSBuildDeps(self)
        # We assume that -o *:shared=True is used to install all shared deps too
        if self.options.shared:
            ms.configuration = str(self.settings.build_type) + "Shared"
        ms.generate()
```

This generates new properties files for this custom configuration, and switching it in the IDE allows to gather dependencies configuration like Debug/Release, and even static and/or shared libraries.

Dependencies

`MSBuildDeps` uses the `self.dependencies` to access to the dependencies information. The following dependencies are translated to properties files:

- All the direct dependencies, which are the ones declared by the current `conanfile`, live in the `host` context: all regular `requires`, plus the `tool_requires`, that are in the host context, e.g. test frameworks like `gtest` or `catch`.
- All transitive `requires` of those direct dependencies (all in the host context)
- Tool requires, in the build context, that is, application and executables that run in the build machine irrespective of the destination platform, are added exclusively to the `<ExecutablePath>` property, taking the value from `$(Conan{{name}}BinaryDirectories)` defined properties. This allows to define custom build commands, invoke code generation tools, with the `<CustomBuild>` and `<Command>` elements.

Customization

conf

`MSBuildDeps` is affected by these `[conf]` variables:

- `tools.microsoft.msbuilddeps:exclude_code_analysis` list of packages names patterns to be added to the Visual Studio `CAExcludePath` property.

Reference

class MSBuildDeps(*conanfile*)

MSBuildDeps class generator conandeps.props: unconditional import of all *direct* dependencies only

Parameters **conanfile** – < ConanFile object > The current recipe object. Always use `self`.

generate()

Generates `conan_<pkg>_<config>_vars.props`, `conan_<pkg>_<config>.props`, and `conan_<pkg>.props` files into the `conanfile.generators_folder`.

MSBuildToolchain

The `MSBuildToolchain` is the toolchain generator for MSBuild. It will generate MSBuild properties files that can be added to the Visual Studio solution projects. This generator translates the current package configuration, settings, and options, into MSBuild properties files syntax.

This generator can be used by name in conanfiles:

Listing 32: **conanfile.py**

```
class Pkg(ConanFile):
    generators = "MSBuildToolchain"
```

Listing 33: **conanfile.txt**

```
[generators]
MSBuildToolchain
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.microsoft import MSBuildToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = MSBuildToolchain(self)
        tc.generate()
```

The `MSBuildToolchain` will generate three files after a `conan install` command:

```
$ conan install . # default is Release
$ conan install . -s build_type=Debug
```

- The main *conantoolchain.props* file, to be added to the project.
- A *conantoolchain_<config>.props* file, that will be conditionally included from the previous *conantoolchain.props* file based on the configuration and platform, e.g., *conantoolchain_release_x86.props*.
- A *conanvcvars.bat* file with the `vcvars` invocation to define the build environment from the command line, or any other automated tools (might not be required if opening the IDE). This file will be automatically called by the `MSBuild.build()` method.

Every invocation with different configuration creates a new properties `.props` file, that is also conditionally included. That allows to install different configurations, then switch among them directly from the Visual Studio IDE.

The MSBuildToolchain files can configure:

- The Visual Studio runtime (*MT/MD/MTd/MDd*), obtained from Conan input settings.
- The C++ standard, obtained from Conan input settings.

One of the advantages of using toolchains is that they help to achieve the exact same build with local development flows, than when the package is created in the cache.

Customization

conf

`MSBuildToolchain` is affected by these `[conf]` variables:

- `tools.microsoft.msbuildtoolchain:compile_options` dict-like object of extra compile options to be added to `<ClCompile>` section. The dict will be translated as follows: `<[KEY]>[VALUE]</[KEY]>`.
- `tools.build:cxxflags` list of extra C++ flags that will be appended to `<AdditionalOptions>` section from `<ClCompile>` and `<ResourceCompile>` one.
- `tools.build:cflags` list of extra of pure C flags that will be appended to `<AdditionalOptions>` section from `<ClCompile>` and `<ResourceCompile>` one.
- `tools.build:sharedlinkflags` list of extra linker flags that will be appended to `<AdditionalOptions>` section from `<Link>` one.
- `tools.build:exelinkflags` list of extra linker flags that will be appended to `<AdditionalOptions>` section from `<Link>` one.
- `tools.build:defines` list of preprocessor definitions that will be appended to `<PreprocessorDefinitions>` section from `<ResourceCompile>` one.

Reference

class MSBuildToolchain(*conanfile*)

MSBuildToolchain class generator

Parameters **conanfile** – < ConanFile object > The current recipe object. Always use self.

generate()

Generates a conantoolchain.props, a conantoolchain_<config>.props, and, if compiler=msvc, a conanvcvars.bat files. In the first two cases, they'll have the valid XML format with all the good settings like any other VS project *.props file. The last one emulates the vcvarsall.bat env script. See also *VCVars*.

VCVars

Generates a file called conanvcvars.bat that activates the Visual Studio developer command prompt according to the current settings by wrapping the vcvarsall Microsoft bash script.

The VCVars generator can be used by name in conanfiles:

Listing 34: conanfile.py

```
class Pkg(ConanFile):
    generators = "VCVars"
```

Listing 35: conanfile.txt

```
[generators]
VCVars
```

And it can also be fully instantiated in the conanfile generate() method:

Listing 36: conanfile.py

```
from conan import ConanFile
from conan.tools.microsoft import VCVars

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = VCVars(self)
        ms.generate()
```

Customization

conf

VCVars is affected by these [conf] variables:

- tools.microsoft.msbuild:installation_path indicates the path to Visual Studio installation folder. For instance: C:\Program Files (x86)\Microsoft Visual Studio\2019\Community, C:\Program Files (x86)\Microsoft Visual Studio 14.0, etc.

Reference

class VCVars(*conanfile*)

VCVars class generator

Parameters **conanfile** – < ConanFile object > The current recipe object. Always use self.

generate(*scope='build'*)

Creates a conanvcvars.bat file with the good args from settings to set environment variables to configure the command line for native 32-bit or 64-bit compilation.

Parameters **scope** – str Launcher to be used to run all the variables. For instance, if build, then it'll be used the conanbuild launcher.

vs_layout

vs_layout(*conanfile*)

Initialize a layout for a typical Visual Studio project.

Parameters **conanfile** – < ConanFile object > The current recipe object. Always use self.

conan.tools.microsoft.visual

check_min_vs

check_min_vs(*conanfile*, *version*)

This is a helper method to allow the migration of 1.X -> 2.0 and VisualStudio -> msvc settings without breaking recipes. The legacy "Visual Studio" with different toolset is not managed, not worth the complexity.

Parameters

- **conanfile** – < ConanFile object > The current recipe object. Always use self.
- **version** – str Visual Studio or msvc version number.

Example:

```
def validate(self):
    check_min_vs(self, "192")
```

msvc_runtime_flag

msvc_runtime_flag(*conanfile*)

Gets the MSVC runtime flag given the compiler.runtime value from the settings.

Parameters **conanfile** – < ConanFile object > The current recipe object. Always use self.

Returns str runtime flag.

is_msvc

is_msvc(*conanfile*, *build_context=False*)

Validates if the current compiler is msvc.

Parameters

- **conanfile** – < ConanFile object > The current recipe object. Always use self.

- **build_context** – If True, will use the settings from the build context, not host ones

Returns `bool` True, if the host compiler is `msvc`, otherwise, False.

is_msvc_static_runtime

is_msvc_static_runtime(*conanfile*)

Validates when building with Visual Studio or msvc and MT on runtime.

Parameters **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.

Returns `bool` True, if `msvc + runtime MT`. Otherwise, False.

conan.tools.microsoft.subsystems

unix_path

unix_path(*conanfile*, *path*)

Transforms the specified path into the correct one according to the subsystem. To determine the subsystem:

- The `settings_build.os` is checked to verify that we are running on "Windows", otherwise, the path is returned without changes.
- If `settings_build.os.subsystem` is specified (meaning we are running Conan under that subsystem) it will be returned.
- If `conanfile.win_bash==True` (meaning we have to run the commands inside the subsystem), the conf `tools.microsoft.bash:subsystem` has to be declared or it will raise an Exception.
- Otherwise the path is returned without changes.

For instance:

```
from conan.tools.microsoft import unix_path

def build(self):
    adjusted_path = unix_path(self, "C:\path\to\stuff")
```

In the example above, `adjusted_path` will be:

- `/c/path/to/stuff` if msys2 or msys.
- `/cygdrive/c/path/to/stuff` if cygwin.
- `/mnt/c/path/to/stuff` if wsl.
- `/dev/fs/C/path/to/stuff` if sfu.

Parameters

- **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.
- **path** – `str` any folder path.

Returns `str` the proper UNIX path.

CHAPTER

EIGHT

FAQ

See also:

There is a great community behind Conan with users helping each other in Cpplang Slack. Please join us in the `#conan` channel!

CHAPTER

NINE

CHANGELOG

For a more detailed description of the major changes that Conan 2.0 brings, compared with Conan 1.X, please read *What's new in Conan 2.0*

9.1 2.0.0-beta3 (12-Sept-2022)

- Feature: Decouple test_package from create. #12046
- Feature: Warn if special chars in exported refs. #12053
- Feature: Improvements in MSBuildDeps traits. #12032
- Feature: Added support for CLICOLOR_FORCE env var, that will activate the colors in the output if the value is declared and different to 0. #12028
- Fix: Call source() just once for all configurations. #12050
- Fix: Fix deployers not creating output_folder. #11977
- Fix: Fix build_id() removal of require. #12019
- Fix: If Conan fails to load a custom command now it fails with a useful error message. #11720
- Bugfix: If the 'os' is not specified in the build profile and a recipe, in Windows, wanted to run a command. #11728

9.2 2.0.0-beta2 (27-Jul-2022)

- Feature: Add traits support in MSBuildDeps. #11680
- Feature: Add traits support in XcodeDeps. #11615
- Feature: Let dependency define package_id modes. #
- Feature: Add `conan.conanrc` file to setup the conan user home. #11675
- Feature: Add `core.cache:storage_path` to declare the absolute path where you want to store the Conan packages. #11672
- Feature: Add tools for checking max cppstd version. #11610
- Feature: Add a `post_build_fail` hook that is called when a build fails. #11593
- Feature: Add `pre_generate` and `post_generate` hook, covering the generation of files around the `generate()` method call. #11593
- Feature: Brought `conan config list` command back and other conf improvements. #11575

- Feature: Added two new arguments for all commands -v for controlling the verbosity of the output and –logger to output the contents in a json log format for log processors. #11522

9.3 2.0.0-beta1 (20-Jun-2022)

- Feature: New graph model to better support C and C++ binaries relationships, compilation, and linkage.
- Feature: New documented public Python API, for user automation
- Feature: New build system integrations, more flexible and powerful, and providing transparent integration when possible, like `CMakeDeps` and `CMakeToolchain`
- Feature: New custom user commands, that can be built using the public PythonAPI and can be shared and installed with `conan config install`
- Feature: New CLI interface, with cleaner commands and more structured output
- Feature: New deployers mechanism to copy artifacts from the cache to user folders, and consume those copies while building.
- Feature: Improved `package_id` computation, taking into account the new more detailed graph model.
- Feature: Added `compatibility.py` extension mechanism to allow users to define binary compatibility globally.
- Feature: Simpler and more powerful `lockfiles` to provide reproducibility over time.
- Feature: Better configuration with `[conf]` and better environment management with the new `conan.tools.env` tools.
- Feature: Conan cache now can store multiple revisions simultaneously.
- Feature: New extensions plugins to implement profile checking, package signing, and build commands wrapping.
- Feature: Used the package immutability for an improved update, install and upload flows.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

V

W

X

Y

Z