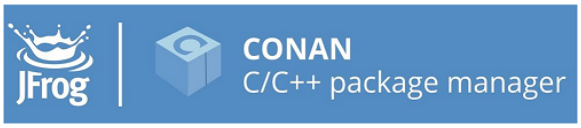


Conan Documentation

Release 2.0.0-alpha

The Conan team

Jun 03, 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.

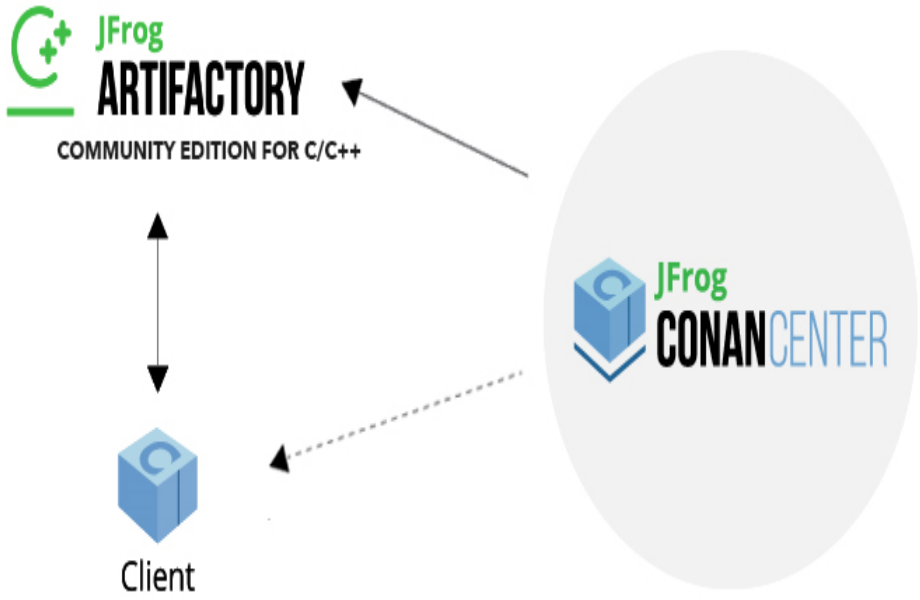
1.1 Open Source

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.



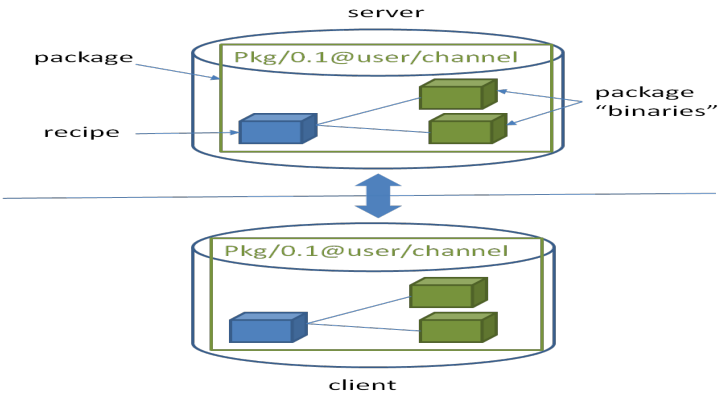
The different applications in the image above are:

- 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

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.

2.1 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.

2.1.1 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
```

2.2 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.

2.3 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

THREE

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.

3.1 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.

3.1.1 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 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`

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/getting_started/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 *cmake-build-release* (Linux/macOS) or in the folder *build* (Windows). To do that, just run:

Listing 4: Windows

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

Listing 5: Linux, macOS

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

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

```
(Windows)
$ conan install . --output-folder=build --build=missing

(Linux, macOS)
$ conan install . --output-folder cmake-build-release --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/getting_started/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
```

(continues on next page)

(continued from previous page)

```
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 **cmake-build-release** 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 6: 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 7: Linux, macOS

```
$ cd cmake-build-release
$ 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 Autotools
- Getting started with Meson
- ...

3.1.2 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/getting_started/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 8: **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 9: **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 in the folder *cmake-build-release* (Linux/macOS) or in the folder *build* (Windows). To do that, just run:

Listing 10: Windows

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

Listing 11: Linux, macOS

```
$ conan install . --output-folder cmake-build-release --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
Downloading conanfile.py
cmake/3.19.8: Downloaded recipe revision 3e3d8f3a848b2a60afafbe7a0955085a
Graph root
    conanfile.txt: /Users/carlosz/Documents/developer/conan/examples2/tutorial/
↪getting_started/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 12: Windows

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

Listing 13: Linux, macOS

```
$ cd cmake-build-release
$ 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 14: 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 15: 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 16: Windows

```
$ deactivate_conanbuild.bat
```

Listing 17: 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

3.1.3 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/getting_started/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
```

(continues on next page)

(continued from previous page)

```
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 18: <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:

Listing 19: Windows

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

Listing 20: Linux, macOS

```
$ conan install . --output-folder cmake-build-release --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 21: 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
$ Debug\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
Debug configuration!
```

Listing 22: Linux, macOS

```
$ cd cmake-build-release
$ 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 23: Windows

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

Listing 24: Linux, macOS

```
$ conan install . --output-folder cmake-build-release --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 25: 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 26: Linux, Macos

```
$ cd cmake-build-release
$ 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 27: 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 28: 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 29: Windows

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

(continues on next page)

(continued from previous page)

```
...
```

Listing 30: 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 31: Windows

```
$ deactivate_conanrun.bat
```

Listing 32: 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

3.1.4 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 configura-

tion (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 33: **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 dynamically, 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/getting_started/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 34: **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 35: **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 36: 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 37: Linux, macOS

```
$ conan install . --output-folder cmake-build-release --build=missing
$ cd cmake-build-release
$ source conanbuild.sh
Capturing current environment in deactivate_conanbuildenv-release-x86_64.sh
```

(continues on next page)

(continued from previous page)

```
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 38: 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 39: **conanfile.py**

```
from conan import ConanFile
```

(continues on next page)

(continued from previous page)

```
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):
        build_type = str(self.settings.build_type)
        compiler = self.settings.get_safe("compiler")

        # We make the assumption that if the compiler is msvc the
        # CMake generator is multi-config
        if compiler == "msvc":
            multi = True
        else:
            multi = False

        if multi:
            # CMake multi-config, just one folder for both builds
            self.folders.build = "build"
            self.folders.generators = "build"
        else:
            self.folders.build = "cmake-build-{}".format(build_type.lower())
            self.folders.generators = self.folders.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 40: 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
```

(continues on next page)

(continued from previous page)

```
...
[100%] Built target compressor

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

Listing 41: Linux, macOS

```
$ conan install . --build=missing
$ cd cmake-build-release
$ 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
```

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 42: **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.

Listing 43: conanfile.py

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

class CompressorRecipe(ConanFile):
    ...

    def validate(self):
        if self.settings.os == "Macos" and self.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()

3.1.5 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 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 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 44: <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 45: <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 46: **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
```

(continues on next page)

(continued from previous page)

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

3.1.6 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 47: **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 .

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 48: **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
```

(continues on next page)

(continued from previous page)

```
    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*

3.2 Creating Packages

This section shows how to create, build and test your packages.

3.2.1 Getting started

This section introduces how to create your own Conan packages, explain *conanfile.py* recipes, and the commands to build packages from sources in your computer.

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.

Using the **conan new** command will create a "Hello World" C++ library example project for us:

```
$ mkdir hellopkg && cd hellopkg
$ conan new cmake_lib --name=hello --version=0.1
File saved: CMakeLists.txt
File saved: conanfile.py
File saved: src/hello.cpp
File saved: src/hello.h
File saved: test_package/CMakeLists.txt
File saved: test_package/conanfile.py
File saved: test_package/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.
- (optional) **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 conans import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake
from conan.tools.layout import cmake_layout


class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"

    # 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 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/*"

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

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

(continues on next page)

(continued from previous page)

```
    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"]
```

Let's explain this recipe a little bit:

- The binary configuration is composed by `settings` and `options`. When something changes in the configuration, the resulting binary built and packaged will be different:
  - `settings` are project-wide configuration that cannot be defaulted in recipes, like the OS or the architecture.
  - `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.
- The `exports_sources` attribute defines which sources are exported together with the recipe, these sources become part of the package recipe (others mechanisms don't do this, will be explained later).
- 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 `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. Although this method implies some potential duplication with the build system output (CMake could generate xxx-config.cmake files), it is important to define this, as Conan packages can be consumed by any other build system, not only CMake.

The content of 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 . demo/testing
...
hello/0.1: Hello World Release!
  hello/0.1: _M_X64 defined
  ...
```

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/0.1@demo/testing` package starts, calling the `generate()`, `build()` and `package()` methods. This creates the binary package in the Conan cache.
- 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/0.1@demo/testing#afa4685e137e7d13f2b9845987c5af77

$ conan list package-ids hello/0.1@demo/testing#afa4685e137e7d13f2b9845987c5af77
Local Cache:
  hello/0.1@demo/testing
↪#afa4685e137e7d13f2b9845987c5af77:e360b62ce00057522e221cfe56714705a46e20e2
    settings:
      arch=x86_64
      build_type=Release
      compiler=apple-clang
      compiler.libcxx=libc++
      compiler.version=12.0
      os=Macos
    options:
      fPIC=True
      shared=False
```

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 those configurations:

```
$ conan create . demo/testing -s build_type=Debug
...
hello/0.1: Hello World Debug!

$ conan create . demo/testing -o hello:shared=True
...
hello/0.1: 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:

```
$ conan list package-ids hello/0.1@demo/testing#afa4685e137e7d13f2b9845987c5af77
Local Cache:
  hello/0.1@demo/testing
↪#afa4685e137e7d13f2b9845987c5af77:842490321f80b0a9e1ba253d04972a72b836aa28
    settings:
      arch=x86_64
      build_type=Release
      compiler=apple-clang
      compiler.libcxx=libc++
      compiler.version=12.0
      os=Macos
    options:
      fPIC=True
      shared=True
  hello/0.1@demo/testing
↪#afa4685e137e7d13f2b9845987c5af77:a5c01fc21d2db712d56189dff69fc10f12b22375
    settings:
      arch=x86_64
      build_type=Debug
      compiler=apple-clang
      compiler.libcxx=libc++
      compiler.version=12.0
      os=Macos
    options:
      fPIC=True
      shared=False
  hello/0.1@demo/testing
↪#afa4685e137e7d13f2b9845987c5af77:e360b62ce00057522e221cfe56714705a46e20e2
    settings:
      arch=x86_64
      build_type=Release
      compiler=apple-clang
      compiler.libcxx=libc++
      compiler.version=12.0
      os=Macos
    options:
      fPIC=True
      shared=False
```

Any doubts? Please check out our *FAQ section* or open a Github issue

3.3 Versioning and Continuous Integration

Intro to versions, semver

Intro to version ranges

Intro to revisions

Intro to lockfiles

Intro to versions conflicts

3.3.1 Version ranges

3.3.2 Revisions

3.3.3 Lockfiles

3.3.4 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

FOUR

INTEGRATIONS

CHAPTER

FIVE

EXAMPLES

5.1 tools.cmake

5.1.1 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!
```

(continues on next page)

(continued from previous page)

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


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

Note that we haven't needed to create the `cmake-build-debug` or `cmake-build-release` 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.

CHAPTER

SIX

REFERENCE

6.1 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:

6.1.1 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 conans 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`.

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.header_only()
```

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 conans 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
        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 conans 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 conans 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 conans import ConanFile
```

(continues on next page)

(continued from previous page)

```
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 conans 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 conans 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 conans import ConanFile

class FooRecipe(ConanFile):
    ...
    win_bash = True
```

It can also be declared as a `property` based on any condition:

```
from conans import ConanFile

class FooRecipe(ConanFile):
    ...


    @property
    def win_bash(self):
        return self.settings.arch == "armv8"
```

6.1.2 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"]
    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
```

(continues on next page)

(continued from previous page)

```
    # 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.

6.1.3 Methods

6.2 Conan commands

6.2.1 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:
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
```

(continues on next page)

(continued from previous page)

```
    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"
            }
        ]
    }
]
```

6.2.2 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"
            }
        ]
    }
]
```

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
```

(continues on next page)

(continued from previous page)

```
    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)
```

6.2.3 Creator commands

6.3 Python API

6.3.1 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
- ...

6.3.2 Remotes API

orphan

Warning: The Conan API is experimental and subject to breaking changes.

class RemotesAPI(*conan_api*)

6.3.3 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

6.3.4 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]

6.3.5 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

6.3.6 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

6.3.7 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

6.3.8 Export API

orphan

> **Warning:** The Conan API is experimental and subject to breaking changes.

class ExportAPI(*conan_api*)

6.3.9 Remove API

orphan

> **Warning:** The Conan API is experimental and subject to breaking changes.

class RemoveAPI(*conan_api*)

6.3.10 Config API

orphan

> **Warning:** The Conan API is experimental and subject to breaking changes.

class ConfigAPI(*conan_api*)

6.3.11 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

6.3.12 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

6.3.13 Download API

orphan

Warning: The Conan API is experimental and subject to breaking changes.

class DownloadAPI(*conan_api*)

6.4 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:

6.4.1 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 conans 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 conans 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 1: **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"]
```

(continues on next page)

(continued from previous page)

```
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()`.
- **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")

    # 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")
    # Automatically include the lib/mypkg.cmake file when calling find_package()
    self.cpp_info.components["mycomponent"].set_property("cmake_build_modules", [os.
↪path.join("lib", "mypkg.cmake")])

    # 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 conans import ConanFile

class Pkg(ConanFile):
    generators = "CMakeToolchain"
```

Or fully instantiated in the `generate()` method:

```
from conans 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
- **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.
- **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.

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 ...)`.

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 conans 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 conans import ConanFile
  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 conans 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"]
```

(continues on next page)

(continued from previous page)

```
    # 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

# 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):
```

(continues on next page)

(continued from previous page)

```
    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.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 conans 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}
```

(continues on next page)

(continued from previous page)

```
    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.

  - `cmake-build-debug` or `cmake-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.
  - `cmake-build-debug-apple-clang-shared_false`: if the cmake generator is single-configuration.
- `conanfile.folders.generators`: `build-apple-clang-shared_false/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.
  - `cmake-build-debug-apple-clang-shared_true`: 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.

Reference

`cmake_layout`(*conanfile*, *generator=None*, *src_folder='.'*)

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.

6.4.2 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 2: conanfile.py

```
class Pkg(ConanFile):
    generators = "AutotoolsDeps"
```

Listing 3: 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 4: conanfile.py

```
class Pkg(ConanFile):
    generators = "AutotoolsToolchain"
```

Listing 5: conanfile.txt

```
[generators]
AutotoolsToolchain
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conans 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 conans 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.
- **cxxflags** (Defaulted to `[]`): Additional cxxflags.
- **cflags** (Defaulted to `[]`): Additional cflags.
- **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`

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"]
    # 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 conans 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 conans 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):
  ...
  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 6: conanfile.py

```
class Pkg(ConanFile):
    generators = "PkgConfigDeps"
```

Listing 7: conanfile.txt

```
[generators]
PkgConfigDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conans import ConanFile
from conan.tools.gnu import PkgConfigDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "zlib/1.2.11"

    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

Returns A dict with file names as keys and contents as values

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`.

6.4.3 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 8: conanfile.py

```
class Pkg(ConanFile):
    generators = "XcodeDeps"
```

Listing 9: conanfile.txt

```
[generators]
XcodeDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 10: 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_libpng.xcconfig
├── conan_libpng_debug_x86_64.xcconfig
├── conan_libpng_release_x86_64.xcconfig
├── conan_libpng_vars_debug_x86_64.xcconfig
├── conan_libpng_vars_release_x86_64.xcconfig
├── conan_zlib.xcconfig
├── conan_zlib_debug_x86_64.xcconfig
├── conan_zlib_release_x86_64.xcconfig
├── conan_zlib_vars_debug_x86_64.xcconfig
├── conan_zlib_vars_release_x86_64.xcconfig
├── conandeps.xcconfig
└── conan_config.xcconfig
```

The first `conan install` with the default *Release* and *x86_64* configuration generates:

- *conan_libpng_vars_release_x86_64.xcconfig*: declares some intermediate variables that are included in *conan_libpng_release_x86_64.xcconfig*
- *conan_libpng_release_x86_64.xcconfig*: includes *conan_libpng_vars_release_x86_64.xcconfig* and 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.xcconfig*: includes *conan_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.xcconfig* in this case).
- 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_vars_release_x86_64.xcconfig*, *conan_zlib_release_x86_64.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_vars_debug_x86_64.xcconfig*: same variables as the one below for *Debug* configuration.
- *conan_libpng_debug_x86_64.xcconfig*: same variables as the one below for *Debug* configuration.
- *conan_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*.
- 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_vars_debug_x86_64.xcconfig*, *conan_zlib_debug_x86_64.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 11: conanfile.py

```
class Pkg(ConanFile):
    generators = "XcodeToolchain"
```

Listing 12: 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)
```

6.4.4 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 conans 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:

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 13: conanfile.py

```
class Pkg(ConanFile):
    generators = "VirtualBuildEnv"
```

Listing 14: conanfile.txt

```
[generators]
VirtualBuildEnv
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 15: conanfile.py

```
from conans 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 16: conanfile.py

```
class Pkg(ConanFile):
    generators = "VirtualRunEnv"
```

Listing 17: conanfile.txt

```
[generators]
VirtualRunEnv
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 18: conanfile.py

```
from conans 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.

6.4.5 conan.tools.build

Building

conan.tools.build.build_jobs()

build_jobs(*conanfile*)

Get the number of jobs to use while building. Read from the conf `tools.build:jobs` or autodetected, by

default it will return the number of CPUs available.

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.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.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.

6.4.6 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.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*)

Utility functions to remove a directory. The existence of the specified directory is checked, so rmdir() will do nothing if the directory doesn't exists.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **path** – Path to the folder to be removed.

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"]  # Deafult value is 'lib'
    self.cpp_info.libs = collect_libs(self)
```

conan.tools.files downloads

conan.tools.files.get()

get(*conanfile*, *url*, *md5=''*, *sha1=''*, *sha256=''*, *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=''*, *sha1=''*, *sha256=''*)

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.
- **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)
```

(continues on next page)

(continued from previous page)

```
# 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
```

(continues on next page)

(continued from previous page)

```
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);
        +     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)
```

(continues on next page)

(continued from previous page)

```
        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`.

6.4.7 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

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 19: **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
```

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 20: **android_host_profile**

```
[settings]
os = Android
os.api_level = 21
arch = armv8

[conf]
tools.android:ndk_path=/my/path/to/NDK
```

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"]`

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.

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.

6.4.8 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 21: 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*
- *Yum* for **Linux** with distribution names: *pidora*, *scientific*, *xenserver*, *amazon*, *oracle*, *amzn*, *almalinux*
- *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 manager 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"`: 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 22: 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",
                    "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.

6.4.9 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=<conf> /p:Platform=
↪<platform>
```

Where:

- `<vcvars-cmd>` calls the Visual Studio prompt that matches the current recipe `settings`.
- `<conf>` is the configuration, typically `Release`, or `Debug`, which is obtained from `settings.build_type`, but this is configurable.
- `<platform>` is the architecture, a mapping from the `settings.arch` to the common 'x86', 'x64', 'ARM', 'ARM64'.

Customization

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*)
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

Returns str msbuild command line.

build(*sln*)
Runs the msbuild command line obtained from self.command(sln).

Parameters **sln** – str name of Visual Studio *.sln file

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 23: conanfile.py

```
class Pkg(ConanFile):
    generators = "MSBuildDeps"
```

Listing 24: conanfile.txt

```
[generators]
MSBuildDeps
```

And it can also be fully instantiated in the conanfile generate() method:

Listing 25: 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()
```

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.

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

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 26: **conanfile.py**

```
class Pkg(ConanFile):
    generators = "MSBuildToolchain"
```

Listing 27: **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 28: conanfile.py

```
class Pkg(ConanFile):
    generators = "VCVars"
```

Listing 29: conanfile.txt

```
[generators]
VCVars
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 30: 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*)

Validates if the current compiler is `msvc`.

Parameters **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.

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

SEVEN

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!

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z