# B5 - Advanced C++

B-CPP-500

# Babel Boostrap

Overcome the barbarian

1.0

{EPITECH.}

## Conan setup

### What is Conan ?

Conan is a C++ package manager, written in python. It handle download and build of your project's dependencies.
Package are downloaded from Conan repositories, then used as is, or recompiled if they are incompatible with your configuration.

### Exercise 0 - Installing Conan

Install `conan` binary using python3 pip command.

```
Terminal
~/B-CPP-500> pip install --user conan
```

Depending on your distribution, `pip` can be installed either as `pip` or `pip3`. You SHOULD check whether `pip` depend on python2 or python3 using `pip --version`

You MIGHT want to add `conan` location to your PATH variable.

`conan` can also be installed as a system binary using `sudo pip install conan`

Check that conan was properly installed.

```
Terminal
~/B-CPP-500> conan
Consumer commands
  install    Installs the requirements specified in a recipe (conanfile.py or
conanfile.txt).
  config     Manages Conan configuration.
  get        Gets a file or list a directory of a given reference or package.
  info       Gets information about the dependency graph of a recipe.
[...]
Conan commands. Type "conan -h" for help
```

## EXERCISE 1 - CONAN GLOBAL CONFIGURATION (PROFILES)

You should now update your Conan profile.
Profiles are located in `$HOME/.conan/profiles`.

Conan allow multiple profiles. You can either copy the file *default* to a new name, or edit it.
If no profile is available, you can create an empty one using `conan profile new <name>`, or let conan fill it with `conan profile new --detect <name>`.

Edit the following variables:

- compiler: The binary you will use to compile your project. This is necessary as conan might need to recompile your dependencies using this binary.
- compiler.version: This must match your compiler version, as shown by `<compiler> --version`. Usually, only the major version is needed (eg. if your compiler version is 11.1, you can use 11 here.)
- compiler.libcxx: Configure which version of c++ standard will be used to compile dependencies. You MUST set this variable to use c++11 (`libstdc++11` for gcc), as c++11 and later are not compatible with older version of the standard.

Optionally, you can edit `build_type` and set it to Debug. Dependencies compile using Debug should contains debug information, which would help your during your development.

You might want to create two profile: The default one, that will build using Debug profile, and a second one to build your dependencies using Release build.

## Exercise 2 - Conan remotes

When a given package is requested (more on that later), conan looks the package specified in its available remote.
Conan has a default remote: conancenter, with a collection of packages.
However, not all library are available, nor are every libraries versions.
Remote management is made using `conan remote` commands.
To add a new remote, run the following commmand

Terminal

```
~/B-CPP-500> conan remote add <remote_name> <url>
```

Let's add Bincrafters remote:

Terminal

```
~/B-CPP-500> conan remote add bincrafters \
https://bincrafters.jfrog.io/artifactory/api/conan/public-conan
```

Bincrafters repositories need revisions to be enabled to work properly. To enable these, run `conan config set general.revisions_enabled=1`

## Exercise 3 - Conan project configuration

Conan must be configured for each project. To do so, you can either use a file named conanfile.txt, or using python via conanfile.py, which won't be covered here.

### Part 0 - Anatomy of conanfile.txt

The conanfile.txt file is split into sections. Each section begin with the section name between square brackets (`[section_name]`). The section content is then written, with each parameter on a single line.
Sections are defined as follow:

- **requires (mandatory)**: This section contains your dependencies, in the following format `package/version`. Optionally, you can also specify the user and channel from which to retrieve the package using `package/version@user/channel`
- **build_requires**: Same as requires. This section should be used for dependencies used only at build time, and not linked to the binary build.
- **options**: This section enable you to tweak some packages features or dependencies. Each line must be written as `package:option=value`. Most package have the "shared" option, that specify whether the package will be statically link, or a shared object.
- **imports**: Using this section, you can instruct conan to copy files from their package folder to the destination folder of your package. Use this to handle shared libraries, using the following syntax : `<src_folder>, <pattern> -> <dest_folder>`. For example: `lib, *.so -> bin` will move every shared object from the folder called lib in every packages, to the folder bin in your build folder.
- **generators**: This section configure what files conan should generate. It's used to create files used by your build-system. For example, using `cmake` will produce a file called `conanbuildinfo.cmake` that will define variables to use inside cmake.

### Part 1 - Adding dependencies

Begin by adding the header of the "requires" section.
In this section, we'll add a dependency on portaudio. We'll use version `19.7.0`, which is packaged by `bincrafters`
.
Your file should now looks something like this:

```
Terminal
~/B-CPP-500> cat conanfile.txt
[requires]
portaudio/19.7.0@bincrafters/stable
```

Try to add the following dependencies:

- opus (1.3.1) (from conan-center)
- qt (5.15.2) (from conan-center)

You can search a package using `conan search <pattern> -r <remote_name>`

## Part 2 - Configuring dependencies

Next, we need to configure each dependencies options. Most package support the `shared` option, which should be used to compile the dependencies as a shared object.
You can set common options using pattern (e.g.: `*:shared = True` will set all packages to be compiled as shared libraries).

You can get the informations on the package using `conan inspect <package>/<version> [-r <remote>] [-a <attribute>]`. You MIGHT want to use the `options` and `default_options` attributes.

Some options change which dependencies will be used by the configured packages. It's important that you check which options are available, especially if the package cannot be built because of missing dependencies.

## Part 3 - Importing files

Shared object libraries are loaded at runtime, either from a known path (system libraries), from a path contains in the environment (LD_LIBRARY_PATH) or from a path stored inside the binary.
If you don't want the user to be responsible of installing your dependencies, you should ship the shared objects alongside your software, and either use a launcher (such as a script) that will set LD_LIBRARY_PATH, or the rpath variable, stored in the binary.
Both solution require you to know the path in which your libraries will be. The easiest way is to use the import section in your `conanfile.txt`.
Adds the `imports` section to your `conanfile.txt`, and write rules to import any shared object in the folder bin/lib.

Most packages will store their library in a folder lib. However, you can consult the recipe used to build the package to check this. You can also check your local cache in `$HOME/.conan/data/`. Using `find` might be a good idea.

Some libraries needs configuration files to be imported as well.

## Part 4 - CMake Integration

At this point your conanfile.txt should looks as follow:

```
~/B-CPP-500> cat conanfile.txt
[requires]
portaudio/19.7.0@bincrafters/stable
opus/1.3.1
qt/5.15.2

[options]
*:shared=True
qt:AAA=BBB

[import]
lib, *.so* -> lib
```

We now need to setup integration with CMake, using generators. Conan support several generators to use for CMake integration. We'll see two flavor of CMake generators, but many more exists, as well as for other tools.

Take some time to read https://docs.conan.io/en/latest/reference/generators.html#generators-reference, to know more about generators, and how to use these.

**cmake** The `cmake` generator create a file called `cmakebuildinfo.cmake`. When included in your `CMakeLists.txt`, it defines several variable and functions.
Once the file is included, you should call `conan_basic_setup()` to initialize them.
The following global variables are especially noteworthy:

- CONAN_LIBS: A list of all libraries, to be used in CMake function `target_link_libraries*()`
- CONAN_INCLUDE_DIRS: A list of all libraries directories containing headers, to be used in CMake function `target_include_directories()`

Alternatively, these variables are defined per packages, with the following syntax: `<global>_<package>`

Here <package> represent the name of the package, in uppercase

This generator also allow you to add libraries per target. To do so, call `conan_basic_setup(TARGETS)`, then add link libraries as follow:

```
Terminal
~/B-CPP-500> cat CMakeLists.txt
[...]
conan_basic_setup(TARGETS)
[...]
add_executable(<target> ${SRCS})
conan_target_link_libraries(<target> CONAN_PKG::<package>)
```

You should read https://docs.conan.io/en/latest/reference/generators/cmake.html to know more about macros and variables exposed by conanbuildinfo.cmake

**cmake_find_package** If you've already used CMake, you might be familiar with the `find_package()` command.
Another generator allow you to use this syntax, which can be useful when building multitarget command, or to be able to reuse an already existing CMakeLists.txt that depends on this feature.
When using this generator, conan will create `Find<package>.cmake` files, which are used by the `find_package (<package>)` command to define needed variables.

You can learn more here : https://docs.conan.io/en/latest/reference/generators/cmake_find_package.html

Take some time to read https://docs.conan.io/en/latest/integrations/build_system/cmake.html to learn more about conan integration with CMake

## EXERCISE 5 - WRAP IT ALL TOGETHER

At this point, you should be able to setup a conan project by yourself. At the root of a directory, create two files: CMakeLists.txt and conanfile.txt.
Setup conan using the conanfile.txt, then create a simple program that initialize all of your dependencies. It does not need to do anything useful, but it MUST call at least one function in each library.
Use your conanfile.txt to import all shared objects in the bin folder or one of its subfolder (e.g. bin/lib)
Setup your CMakeLists.txt to build the project. You MUST use it to adjust the binary RPATH, so your binary load libraries from the folder they are imported in. The RPATH should be relative to the location of your binary.

https://docs.conan.io/en/latest/howtos/manage_shared_libraries/rpaths.html

Build your binary and run it. If the binary work, check which libraries are loaded, and from which location, using `ldd`.
Then copy or move your bin folder, and check again using `ldd` that the path was properly updated.

Terminal

```
~/B-CPP-500> ldd bin/babel
linux-vdso.so.1 (0x00007ffed6de6000)
libportaudio.so.2 => /home/student/B-CPP-500/bootstrap/build/bin/lib/libportaudio.s
(0x00007fb2db1bc000)
libopus.so.0 => /home/student/B-CPP-500/bootstrap/build/bin/lib/libopus.so.0
(0x00007fb2db13b000)
libasound.so.2 => /home/student/B-CPP-500/bootstrap/build/bin/lib/libasound.so.2
(0x00007fb2db04c000)
[...]
~/B-CPP-500> mv bin test
~/B-CPP-500> ldd test/babel
linux-vdso.so.1 (0x00007ffed6de6000)
libportaudio.so.2 => /home/student/B-CPP-500/bootstrap/build/test/lib/libportaudio.
(0x00007fb2db1bc000)
libopus.so.0 => /home/student/B-CPP-500/bootstrap/build/test/lib/libopus.so.0
(0x00007fb2db13b000)
libasound.so.2 => /home/student/B-CPP-500/bootstrap/build/test/lib/libasound.so.2
(0x00007fb2db04c000)
[...]
```