

# Conan – menadżer pakietów, którego potrzebuje projekt C++

Programiści Pythona mają pip, PHP – composer, Ruby ma RubyGems, świat JavaScript ma npm, Rust ma cargo. A C++? Przez dekady odpowiedzią było „jakoś to będzie” – ręczne ściąganie źródeł, system pakietów dystrybucji Linuksa, submoduły git czy kopiowanie plików nagłówkowych wprost do repozytorium. Każde z tych podejść działa, dopóki projekt nie urośnie, nie pojawi się potrzeba budowania na wielu platformach albo aktualizacji jednej z zależności. Nie wspominając już o dołączeniu nowego członka zespołu, który spędzi pierwszy tydzień na walce z zależnościami i konfiguracji środowiska.

Problem zarządzania zależnościami w C++ jest trudniejszy niż w większości języków. Nie istnieje jeden „oficjalny” sposób budowania projektów – mamy CMake, Meson, Make, Ninja, MSBuild i wiele innych. Biblioteki mogą wymagać kompilacji z określonymi flagami, linkowania statycznego lub dynamicznego, a to samo źródło musi często działać na Linuksie, Windowsie i macOS, z różnymi kompilatorami i ich wersjami. Liczba kombinacji rośnie geometrycznie i szybko wymyka się możliwości ręcznego jej zarządzania.

Conan powstał, żeby rozwiązać dokładnie ten problem. Jest to menadżer pakietów zaprojektowany z myślą o specyfice ekosystemu C i C++ – obsługuje dowolny system budowania, pozwala precyzyjnie definiować warianty binarne dla różnych konfiguracji i integruje się z istniejącymi projektami bez wymuszania rewolucji w strukturze kodu. Utworzył go zespół, który wcześniej rozwijał nieistniejący już projekt biicode.

W tym artykule przeprowadzimy czytelnika od instalacji, przez konsumowanie pakietów z publicznego repozytorium ConanCenter, aż po tworzenie własnych pakietów i cross-kompilację. W przykładach użyto CMake, choć Conan współpracuje również z innymi systemami budowania. Czytelnik powinien czuć się swobodnie z podstawami CMake i kompilacji C++.

## I PIERWSZE KROKI

### I Instalacja

Conan jest aplikacją napisaną w Pythonie, więc najprostszą metodą instalacji jest użycie pip:

#### Listing 0. Instalacja Conana za pomocą pip

```
% pip install conan
Collecting conan
  Using cached conan-2.25.1-py3-none-any.whl
Collecting requests<3.0.0,>=2.25 (from conan)
[...]
Using cached six-1.17.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: [...] conan
Successfully installed [...] conan
```

Na nowszych dystrybucjach Linuksa (Ubuntu 24.04+, Fedora 38+) systemowy Python jest oznaczony jako „externally managed” i pip odmówi globalnej instalacji. W takim przypadku można użyć pipx, który automatycznie tworzy izolowane środowisko:

#### Listing 1. Instalacja Conana za pomocą pipx

```
% pipx install conan
installed package conan 2.25.1, installed using Python 3.13.7
These apps are now globally available
- conan
done! 🌟🌟🌟
```

Alternatywnie dostępne są natywne pakiety dla popularnych systemów: Homebrew na macOS (`brew install conan`), pakiet AUR dla Arch Linux, oraz gotowe binaria do pobrania dla Windows, Linux i macOS dostępne w repozytorium projektu w serwisie GitHub [0]. Te ostatnie nie wymagają zainstalowanego Pythona – zawierają własny interpreter.

Po instalacji warto zweryfikować, że wszystko działa:

#### Listing 2. Weryfikacja instalacji i sprawdzenie zainstalowanej wersji Conana wraz z przykładowym wywołaniem

```
# conan --version
Conan version 2.25.1
```

### I Konfiguracja profilu

Zanim zaczniemy korzystać z pakietów, Conan musi wiedzieć, w jakim środowisku pracujemy – jaki mamy system operacyjny, kompilator, architekturę i domyślny typ budowania. Te informacje przechowywane są w pliku profilu. Conan potrafi wykryć te ustawienia automatycznie:

#### Listing 3. Detekcja domyślnego profilu kompilacji dla Conana

```
% conan profile detect
detect_api: Found cc=gcc-15.2.1
detect_api: gcc>=5, using the major as version
detect_api: gcc C++ standard library: libstdc++11

Detected profile:
```

```
[settings]
arch=x86_64
build_type=Release
compiler=gcc
compiler.cppstd=gnu17
compiler.libcxx=libstdc++11
compiler.version=15
os=Linux
```

```
WARN: This profile is a guess of your environment,
      please check it.
WARN: The output of this command is not guaranteed to be
      stable and can change in future Conan versions.
WARN: Use your own profile files for stability.
Saving detected profile to [...]/profiles/default
```

Profil zapisywany jest w katalogu `~/conan2/profiles/default` (lub odpowiedniku na systemie Windows). Można go edytować ręcznie, tworzyć dodatkowe profile dla innych konfiguracji (np. Debug, cross-kompilacja) i przełączać się między nimi flagą `--profile`.

Warto zwrócić uwagę na `compiler.cppstd` – Conan automatycznie ustawia domyślny standard C++ dla wykrytego kompilatora. Jeśli projekt wymaga konkretnej wersji standardu, można ją zmienić bezpośrednio w profilu lub nadpisać z linii poleceń. Na moim systemie domyślnie ustawiam wersję standardu `gnu26`.

## I Anatomia conanfile.txt i conanfile.py

Conan oferuje dwa sposoby definiowania zależności projektu: prosty plik tekstowy *conanfile.txt* lub skrypt Pythona *conanfile.py*.

Plik *conanfile.txt* to deklaracyjny format wystarczający dla większości prostych przypadków:

### Listing 4. Przykładowy plik conanfile.txt

```
[requires]
fmt/10.2.1
spdlog/1.13.0

[generators]
CMakeDeps
CMakeToolchain

[layout]
cmake_layout
```

Gdy potrzebujemy większej elastyczności – warunkowych zależności, dynamicznej konfiguracji opcji czy definiowania layoutu projektu – przechodzimy na *conanfile.py*:

### Listing 5. Przykładowy (minimalny) plik conanfile.py

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

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

    def requirements(self):
        self.requires("fmt/10.2.1")
        self.requires("spdlog/1.13.0")

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

Plik *conanfile.py* nosi nazwę „przepisu” (ang. *recipe*) i służy zarówno do konsumowania pakietów, jak i do ich tworzenia – o czym więcej w dalszej części artykułu. Dla prostego projektu konsumującego kilka bibliotek *conanfile.txt* jest całkowicie wystarczający, ale warto od razu zapoznać się z wersją pythonową – migracja w tę stronę wydaje się nieunikniona, gdy projekt się rozrośnie.

Przepis to zwykła klasa Pythona dziedzicząca po `ConanFile`. Konfigurację definiujemy na dwa sposoby: przez atrybuty klasy (dla wartości statycznych) oraz przez metody (gdy potrzebujemy logiki). Na przykład `settings` i `generators` to atrybuty – ich wartości są znane z góry i nie zależą od żadnych warunków. Z kolei `requirements()` czy `layout()` to metody, bo mogą zawierać dowolny kod Pythona: warunki, pętle, odwołania do innych ustawień.

Przepis – niezależnie, czy w formie *conanfile.txt* czy *conanfile.py* – składa się z kilku sekcji:

**Zależności** (`[requires]` / `requirements()`) definiują biblioteki potrzebne do zbudowania projektu. Wersje można podawać dokładnie (`fmt/10.2.1`) lub jako zakresy w nawiasach kwadratowych – Conan automatycznie wybierze najnowszą pasującą. Conan rozszerza specyfikację `semver` (MAJOR.MINOR.PATCH) o dowolną liczbę segmentów i wspiera kilka operatorów:

- » porównania (`fmt/[>=10.0 <11.0]`),
- » operator tyldy `~`, oznaczający „w przybliżeniu równy”: `fmt/[~10.2]` dopuści `10.2.0` i `10.2.3`, ale nie `10.3`,
- » operator karety `^`, dopuszczający zmiany po pierwszej niezerowej cyfrze: `fmt/[^10.2]` dopuści `10.2.1` i `10.9`, ale nie `11.0` i `10.1` – zmiana może być tylko w górę

Warunki można łączyć operatorem `||`, choć w praktyce rzadko jest to potrzebne.

**Generatory** (`[generators]` / `generators`) określają, jakie pliki integracyjne Conan ma utworzyć. `CMakeDeps` tworzy pliki konfiguracyjne dla `find_package()`, a `CMakeToolchain` generuje preset z ustawieniami toolchaina.

**Layout** (`[layout]` / `layout()`) definiuje strukturę katalogów projektu. Wartość `cmake_layout` ustawia sensowne domyślne ścieżki dla typowych projektów CMake, w tym oddzielne katalogi dla różnych konfiguracji (`build/Release`, `build/Debug`).

**Ustawienia** (`settings` – tylko w *conanfile.py*) deklarują, które parametry z profilu są istotne dla projektu: system operacyjny, kompilator, typ budowania, architektura. Conan używa ich do identyfikacji wariantu binarnego pakietu. W *conanfile.txt* ustawienia są brane bezpośrednio z profilu.

Główna różnica między formatami: *conanfile.txt* jest czysto deklaracyjny, *conanfile.py* pozwala na logikę – warunki, pętle, dynamiczne zależności. Dla prostych projektów wystarczy `.txt`, ale przy rosnącej złożoności migracja do `.py` staje się koniecznością.

## I KONSUMOWANIE PAKIETÓW

### I Repozytorium ConanCenter

Zanim dodamy zależność do projektu, musimy wiedzieć, jakie pakiety są dostępne i w jakich wersjach. ConanCenter to oficjalne, publiczne repozytorium pakietów Conan, zawierające tysiące popularnych

bibliotek C i C++. Można je przeszukiwać przez stronę internetową [1] lub bezpośrednio z linii poleceń:

#### Listing 6. Wyszukiwanie pakietów w repozytorium ConanCenter

```
% conan search "fmt" -r conancenter
Connecting to remote 'conancenter' anonymously
Found 21 pkg/version recipes matching fmt in conancenter
conancenter
fmt
  fmt/5.3.0
  fmt/6.2.1
  fmt/7.1.3
  fmt/8.0.1
  fmt/8.1.1
  fmt/9.0.0
  fmt/9.1.0
  fmt/10.0.0
  fmt/10.1.0
  fmt/10.1.1
  fmt/10.2.0
  fmt/10.2.1
  fmt/11.0.0
  fmt/11.0.1
  fmt/11.0.2
  fmt/11.1.1
  fmt/11.1.3
  fmt/11.1.4
  fmt/11.2.0
  fmt/12.0.0
  fmt/12.1.0
```

## I Praktyczny przykład

Zbudujemy prosty program używający dwóch bibliotek: `fmt` do formatowania tekstu i `spdlog` do logowania. Co istotne, `spdlog` wewnętrznie używa `fmt` – Conan automatycznie rozwiąże tę zależność przechodnią.

Zadaniem tego programu będzie testowanie silnej hipotezy Goldbacha dla liczby 42. Główny plik programu przedstawiony jest w Listingu 7.

#### Listing 7. `main.cpp`

```
#include <spdlog/spdlog.h>
#include <fmt/format.h>

int main()
{
    // Goldbach: 42 = 19 + 23
    auto msg = fmt::format(
        "{} + {} = {}", 19, 23, 19 + 23
    );
    spdlog::info!("{}", msg);
}
```

Definicja zależności `conanfile.txt` znajduje się w Listingu 8. Użyłem najnowszych dostępnych wersji bibliotek `fmt` i `spdlog` na dzień pisania artykułu.

#### Listing 8. `conanfile.txt`

```
[requires]
fmt/12.1.0
spdlog/1.17.0
```

```
[generators]
CMakeDeps
CMakeToolchain
```

```
[layout]
cmake_layout
```

Plik `CMakeLists.txt` (Listing 9) zawiera komendy `find_package` dla dodawanych bibliotek, ale trzeba zauważyć, że nie ma w nim wzmianki o Conan. To celowe – dzięki temu projekt pozostaje niezależny od menedżera pakietów i można go zbudować z bibliotekami zainstalowanymi w dowolny inny sposób.

Integracja odbywa się przez presety CMake (*CMake Presets*). Generator `CMakeToolchain` tworzy plik `CMakeUserPresets.json`, który zawiera wszystkie niezbędne ustawienia: ścieżki do bibliotek, flagi kompilatora, konfigurację `toolchaina`. Kiedy wywołujemy `cmake --preset conan-release`, CMake automatycznie wczytuje te ustawienia i wie, gdzie szukać pakietów zainstalowanych przez Conana.

Z kolei generator `CMakeDeps` tworzy pliki konfiguracyjne dla każdej zależności (np. `fmt-config.cmake`, `spdlog-config.cmake`), które są standardowym mechanizmem CMake do znajdowania bibliotek. Dzięki temu `find_package(fmt REQUIRED)` działa tak, jakby biblioteka była zainstalowana systemowo.

To rozdzielenie odpowiedzialności – Conan zarządza zależnościami, CMake buduje projekt – sprawia, że oba narzędzia robią to, w czym są najlepsze, a nasz `CMakeLists.txt` pozostaje czysty i przenośny.

#### Listing 9. `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.30)
project(intro_txt CXX)

find_package(fmt REQUIRED)
find_package(spdlog REQUIRED)

add_executable(intro main.cpp)
target_link_libraries(intro
    fmt::fmt spdlog::spdlog
)
target_compile_features(intro
    PRIVATE cxx_std_20
)
```

Uwaga: Mechanizm presetów CMake, z którego korzysta Conan, wymaga wersji 3.23 lub nowszej. Dla starszych wersji można użyć bezpośrednio pliku `toolchaina`: `cmake -DCMAKE_TOOLCHAIN_FILE=<ścieżka>` – Conan wypisuje odpowiednią komendę w swoim wyjściu, co widać np. w Listingu F.

Makefile nie jest konieczny dla tego przykładu – równie dobrze można wywołać komendy bezpośrednio:

#### Listing A. Bezpośrednie wywołanie komend, bez użycia `make`

```
# conan install . --build=missing
# cmake --preset conan-release
# cmake --build build/Release
```

Używam jednak `make` w praktycznie każdym projekcie, bo pozwala uzyskać powtarzalność oraz udokumentować sposób budowy niezależnie od użytych technologii. Nowy członek zespołu nie musi wiedzieć, czy projekt używa CMake, Meson, pip, npm, cargo, czy nawet

# programista

1/2026 (122)

Cena 32.90 zł (w tym VAT 8%)

## WŁASNE ŚRODOWISKO DO ĆWICZEŃ RED TEAM



ISSN 2084-9400



9 772084 940602

Conan – menadżer pakietów,  
którego potrzebuje projekt C++

Alternatywne sposoby  
zmiennorodzajowych

MicroPython – lekki Python dla  
mikrokontrolerów i nie tylko

Procedury generacji muzyki  
(na serio)

Nowy numer już w empikach