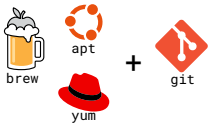
Installing and Building Software in C++/Fortran

Builds for open source software are often **Readme Driven.** In this common approach developers provide a set of instructions for how to install a project's dependencies and the project itself. Common approaches include:

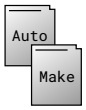
Dependency Management Methods

1) Simply add all dependencies to source tree of your project. Projects with a large number of dependencies can grow to have millions of lines of code, which can have a drastic effect on compilation times. Large C++ projects for example can take between several minutes to several hours to compile from source



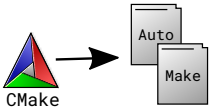
2) Use a system package manager, and source from repositories such as GitHub to install globally. These can then be found by build systems. This makes it difficult to build isolated build environments, and configure builds with different versions, or sets, of dependencies.

Build Methods



1) Developer provided Make and Autotools scripts. lowest barrier to entry but build system will use globally installed dependency libraries, unless alternative provided. Makes multi-platform builds, or those using different software versions challenging.

2) Developer provided CMake scripts, to generate build systems for different environments. Again, reliant on globally installed dependency libraries.



Neither of these methods can check for dependency conflicts, which is left up to the developer to resolve.

Modern Package Management

Modern package managers allow for maximum safety and flexibility. They support multiple operating systems, compilers, build systems, and hardware microarchitetures, and are usually defined by recipes written in a simple scripting language such as Python, and can be used to generate build system scripts such as Makefiles and CMake scripts. The also support dependency tree checking for conflicting requirements, leading to stable builds. Two popular leading tools for HPC in C++ and Fortran are **Spack** and **Conan**. .

Spack

+ Supported on Linux and MacOS.
+ Package recipes specified with Python scripts.
+ Over 5000 commonly used packages available.
+ Supports isolated environments, supporting building packages with from specific constraints, and clean global environment.
+ Can target hardware microarchitectures.
+ Growing support for binary package installation, though not available on all hardware targets.
+ Compatible with all major build systems, such as CMake and SCons
- No Windows support.

Conan

+ Supported on Linux, MacOS and Windows
+ Package recipes specified with Python scripts.
+ Supports isolated environments, supporting building packages with from specific constraints, and clean global environment.
+ Support for binary packages, speeding up installation times.
+ Compatible with all major build systems, such as CMake and SCons.

However, we note that even in 2021 modern package managers are still only used in a tiny fraction of all projects. For example Conan, only accounts for 5% of all C++ projects surveyed by JetBrains[1] in comparison to 21% of projects still relying on the system package manager, 26% simply including a dependencies source code as a part of a project's source tree, 23% using the readme driven build of each specific dependency, and 21% installing non-optimised precompiled binaries from the internet. Indeed only a small minority, 22%, used a package manager of any kind, with no solution taking a majority of even this market share. This is in stark contrast to theuniformity of the situation in Rust, in which all projects use Cargo as a build system and package manager and rustc as a compiler.

1. https://www.jetbrains.com/lp/devecosystem-2021/cpp/