Engineer

Library

code packaged for other programs to call

What it is

A library is a collection of ready-made routines that a program borrows instead of writing them: someone solved a problem once, such as parsing dates, compressing files or drawing a QR code, and published the solution with a set of named entry points that any program can use. The plural is libraries. C libraries on Linux are often named with a lib- prefix, Windows calls the shared kind a DLL, and Rust programmers say crate, a word the Rust Book notes is used interchangeably with library. Most languages also ship a standard library, the modules available in every installation without adding anything.

A library is a kind of reusable code, and it is easiest to define against its neighbours. An application runs on its own and has a user; a library has neither and only runs inside a program that calls it. A framework is the closer comparison. With a library, your code is in charge and calls it when it needs something; a framework is in charge and calls your code at the points it defines. A package, finally, is the box a library ships in, with a name, a version and a list of what it needs, and the library is the code inside the box.

What makes a library usable is its interface: the functions, types and constants it promises to keep. Semantic Versioning builds its whole scheme on that promise. A project that follows it must declare a public API, and its major version goes up whenever that API changes incompatibly.

The Program Library HOWTO sorts libraries on Linux by when they join the program. A static library is copied into the executable when it is built. A shared library is loaded when the program starts and is used by every program that needs it, so it can be updated without rebuilding them. A dynamically loaded library is opened while the program is running, which is how plugin systems usually work.

Use the concept in a decision

Put “What is a library?” to work in a real decision. We can help relate the concept to the system you are building or buying. The shape of a system worked out on paper first, so the parts fit and the cost is known before anyone writes code.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. One infrastructure plan for 14 departments — 14 research departments on one infrastructure plan

One infrastructure plan for 14 departments Get in touch →

Why it matters

Each library a project adopts is a decision with three possible answers: take an existing one, write the code in-house, or copy a small, focused piece into the repository and own it from then on, which is called vendoring.

Taking a library is usually right. Its authors have met the edge cases, and other users have already filed the bug reports. The cost comes later and is easy to miss: the library’s own dependencies arrive with it, its security advisories become yours, its release schedule sets the pace of your upgrades, and its licence travels into your product. A library its maintainer has abandoned stays in your build until someone replaces it.

Writing it yourself buys control and costs everything the library would have saved: the edge cases, the tests and the maintenance, indefinitely. It is the right call when the problem is narrow and central, when the available libraries are far larger than the need, or when a project has a reason to keep its dependency count low.

How a library is linked matters as well. A program built with a static library carries a fixed copy, so a security fix in the library reaches users only when the program is rebuilt and shipped again. A shared library can be patched once for the whole system, at the price of the program trusting whatever version the system supplies. Licensing ignores the difference: the GNU licence FAQ states that a GPL-covered work linked with other modules, whether statically or dynamically, forms a combined work, and the GPL’s terms then cover the whole combination.

For anyone commissioning software, the practical questions are which libraries the result depends on, under which licences, and who owns any libraries the supplier brings along.

What it looks like in practice

A document generator needed a QR code pointing at each document's online version. Any of the existing QR libraries would have done the job, but the project had deliberately limited itself to two runtime dependencies, a template engine and a PDF renderer, a limit chosen so the generator would keep installing cleanly long after release. The answer was to vendor a QR encoder in 522 lines: byte-mode encoding, Reed-Solomon error correction over GF(256), the finder, timing and alignment patterns, format and version information, and all eight mask patterns scored against the four penalty rules. A decoder written separately reads every generated code back on every run and checks that it returns the input. The page is careful about its claim: the encoder covers only the modes and versions those documents use, and the reason for it is the dependency budget, not a belief that it beats a mature library.

A native macOS client calls a core library through its C interface, and early on that interface was represented by a hand-written header. Calls to three functions that had since left the core compiled, linked and then crashed at runtime, because the compiler and the linker had only ever seen the hand-written stand-in and never the core's real declarations. The fix was a parser that verifies every call site against the real 7,308-line header and its 268 declarations, failing the build on any call or enum constant the header does not declare. It checks declarations, so a function that keeps its signature while changing its meaning still passes.

The terms of every engagement settle who owns what. Code written for a client becomes the client's on full payment. Libraries and general-purpose tools the company wrote earlier, or wrote for itself, are the exception: ownership stays with the company, and the client gets a licence to use them inside the delivered work that never expires and cannot be withdrawn. Open-source code from third parties stays under the licence its authors chose, and the client is told which of those components ended up in the result.

Where it goes wrong

A routine update breaks the build

A dependency moves from 2.3 to 2.4, and code that compiled yesterday fails or behaves differently. Either the library changed its public interface without raising its major version, or the project relied on something the library never promised, such as an internal function or an undocumented behaviour. Pin exact versions in a lock file, upgrade on purpose with the changelog open, and call only what the library documents as public.

The program trusts a description of the library

A clean build crashes the first time a feature is used. The code was compiled against headers, type stubs or bindings written by hand, and these drifted away from the library actually linked in. Generate bindings from the library's own published interface, or check every call against it as part of the build, so a mismatch fails there rather than in front of a user.

One small feature drags in a large tree

A single function was needed, and the library that provides it came with dozens of dependencies of its own, each one another set of advisories, licences and updates. Before adding a library, look at what it pulls in, not just at its own size. When the need is narrow and well specified, a small piece of vendored code with its own tests can be the cheaper thing to own.

The licence arrives with the code

Late in a project, someone notices that a library inside a closed-source product is under a licence whose terms extend to the whole combined work. Nobody checked licences when the library was added, and linking it dynamically changed nothing. Check the licence of every library at the moment it is added, keep the list with the release, and ask suppliers for theirs.

Further reading

- Program Library HOWTO: Introduction — what a program library is on Linux, and how static, shared and dynamically loaded libraries differ.
- The Rust Programming Language: Packages and Crates — library crates versus binary crates, and how a package holds them.
- Semantic Versioning 2.0.0 — the version-number rules that turn a library's public API into a promise.
- The Python Standard Library — a standard library at full size: everything that ships with every Python installation.
- Frequently Asked Questions about the GNU Licenses — what the GPL means for programs that link a covered library, statically or dynamically.



Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a library?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/library/