Library (computing)

Not to be confused with Integrated library system or Library computers.

In computer science, a **library** is a collection of non-

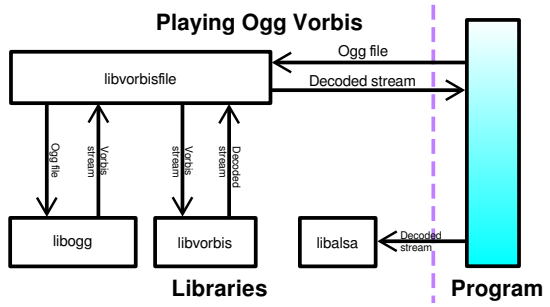

Illustration of an application which uses libvorbisfile to play an Ogg Vorbis file

volatile resources used by computer programs, often to develop software. These may include configuration data, documentation, help data, message templates, pre-written code and subroutines, classes, values or type specifications. In IBM's OS/360 and its successors they are referred to as partitioned data sets.

In computer science, a **library** is a collection of implementations of behavior, written in terms of a language, that has a well-defined interface by which the behavior is invoked. This means that as long as a higher level program uses a library to make system calls, it does not need to be re-written to implement those system calls over and over again. In addition, the behavior is provided for reuse by multiple independent programs. A program invokes the library-provided behavior via a mechanism of the language. For example, in a simple imperative language such as C, the behavior in a library is invoked by using C's normal function-call. What distinguishes the call as being to a library, versus being to another function in the same program, is the way that the code is organized in the system.

Library code is organized in such a way that it can be used by multiple programs that have no connection to each other, while code that is part of a program is organized to only be used within that one program. This distinction can gain a hierarchical notion when a program grows large, such as a multi-million-line program. In that case, there may be internal libraries that are reused by independent sub-portions of the large program. The distinguishing feature is that a library is organized for the purposes of being reused by independent programs or sub-programs, and the user only needs to know the interface, and not the internal details of the library.

The value of a library is the reuse of the behavior. When a program invokes a library, it gains the behavior implemented inside that library without having to implement that behavior itself. Libraries encourage the sharing of code in a modular fashion, and ease the distribution of the code.

The behavior implemented by a library can be connected to the invoking program at different program lifecycle phases. If the code of the library is accessed during the build of the invoking program, then the library is called a static library. An alternative is to build the executable of the invoking program and distribute that, independently from the library implementation. The library behavior is connected after the executable has been invoked to be executed, either as part of the process of starting the execution, or in the middle of execution. In this case the library is called a dynamic library. A dynamic library can be loaded and linked as part of preparing a program for execution, by the linker. Alternatively, in the middle of execution, an application may explicitly request that a module be loaded.

Most compiled languages have a standard library although programmers can also create their own custom libraries. Most modern software systems provide libraries that implement the majority of system services. Such libraries have commoditized the services which a modern application requires. As such, most code used by modern applications is provided in these system libraries.

1 History

The earliest programming concepts analogous to libraries were intended to separate data definitions from the program implementation. JOVIAL brought the "COMPOOL" (Communication Pool) concept to popular attention in 1959, although it adopted the idea from the large-system SAGE software. Following the computer science principles of separation of concerns and information hiding, "Comm Pool's purpose was to permit the sharing of System Data among many programs by providing a centralized data description."[1]

COBOL also included "primitive capabilities for a library system" in 1959,[2] but Jean Sammet described them as "inadequate library facilities" in retrospect.[3]

Another major contributor to the modern library concept came in the form of the subprogram innovation

of FORTRAN. FORTRAN subprograms can be compiled independently of each other, but the compiler lacks a linker. So prior to the introduction of modules in Fortran-90, type checking between FORTRAN[NB 1] subprograms was impossible.[4]

Finally, historians of the concept should remember the influential Simula 67. Simula was the first object-oriented programming language, and its classes are nearly identical to the modern concept as used in Java, C++, and C#. The *class* concept of Simula was also a progenitor of the *package* in Ada and the *module* of Modula-2.[5] Even when developed originally in 1965, Simula classes could be included in library files and added at compile time.[6]

2 Linking

Main articles: Link time and Linker (computing)

Libraries are important in the program *linking* or *binding* process, which resolves references known as *links* or *symbols* to library modules. The linking process is usually automatically done by a *linker* or *binder* program that searches a set of libraries and other modules in a given order. Usually it is not considered an error if a link target can be found multiple times in a given set of libraries. Linking may be done when an executable file is created, or whenever the program is used at run time.

The references being resolved may be addresses for jumps and other routine calls. They may be in the main program, or in one module depending upon another. They are resolved into fixed or relocatable addresses (from a common base) by allocating runtime memory for the memory segments of each module referenced.

Some programming languages may use a feature called *smart linking* wherein the linker is aware of or integrated with the compiler, such that the linker knows how external references are used, and code in a library that is never actually *used*, even though internally referenced, can be discarded from the compiled application. For example, a program that only uses integers for arithmetic, or does no arithmetic operations at all, can exclude floating-point library routines. This smart-linking feature can lead to smaller application file sizes and reduced memory usage.

3 Relocation

Main article: Relocation (computer science)

Some references in a program or library module are stored in a relative or symbolic form which cannot be resolved until all code and libraries are assigned final static addresses. *Relocation* is the process of adjusting these references, and is done either by the linker or the loader. In general, relocation cannot be done to individual libraries themselves because the addresses in memory may vary depending on the program using them and other libraries they are combined with. Position-independent code avoids references to absolute addresses and therefore does not require relocation.

4 Static libraries

Main article: Static library

When linking is performed during the creation of an executable or another object file, it is known as *static linking* or *early binding*. In this case, the linking is usually done by a linker, but may also be done by the compiler. A *static library*, also known as an *archive*, is one intended to be statically linked. Originally, only static libraries existed. Static linking must be performed when any modules are recompiled.

All of the modules required by a program are sometimes statically linked and copied into the executable file. This process, and the resulting stand-alone file, is known as a static build of the program. A static build may not need any further relocation if virtual memory is used and no address space layout randomization is desired.

5 Shared libraries

"Shared object" redirects here. For the synchronization mechanism, see Monitor (synchronization).

A **shared library** or **shared object** is a file that is intended to be shared by executable files and further shared object files. Modules used by a program are loaded from individual shared objects into memory at load time or run time, rather than being copied by a linker when it creates a single monolithic executable file for the program.

Shared libraries can be statically linked, meaning that references to the library modules are resolved and the modules are allocated memory when the executable file is created. But often linking of shared libraries is postponed until they are loaded.

Most modern operating systems[NB 2] can have shared library files of the same format as the executable files. This offers two main advantages: first, it requires making only one loader for both of them, rather than two (having the single loader is considered well worth its added complexity). Secondly, it allows the executables also to be used as shared libraries, if they have a symbol table. Typical combined executable and shared library formats are ELF and Mach-O (both in Unix) and PE (Windows).

In some older environments such as 16-bit Windows or

MPE for the HP 3000 only stack based data (local) was allowed in shared library code, or other significant restrictions were placed on shared library code.

5.1 Memory sharing

Main article: Shared memory (interprocess communication)

Library code may be shared in memory by multiple processes as well as on disk. If virtual memory is used, processes execute the same physical page of RAM, mapped into the different address spaces of each process. This has advantages. For instance on the OpenStep system, applications were often only a few hundred kilobytes in size and loaded quickly; the majority of their code was located in libraries that had already been loaded for other purposes by the operating system.

Programs can accomplish RAM sharing by using position independent code as in Unix, which leads to a complex but flexible architecture, or by using common virtual addresses as in Windows and OS/2. These systems make sure, by various tricks like pre-mapping the address space and reserving slots for each shared library, that code has a great probability of being shared. A third alternative is single-level store, as used by the IBM System/38 and its successors. This allows position-dependent code but places no significant restrictions on where code can be placed or how it can be shared.

In some cases different versions of shared libraries can cause problems, especially when libraries of different versions have the same file name, and different applications installed on a system each require a specific version. Such a scenario is known as DLL hell, named after the Windows and OS/2 DLL file. Most modern operating systems after 2001 have clean-up methods to eliminate such situations or use application specific "private" libraries.[7]

5.2 Dynamic linking

Main article: Dynamic linker

Dynamic linking or late binding is linking performed while a program is being loaded (load time) or executed (run time), rather than when the executable file is created. A dynamically linked library (dynamic-link library or DLL under Windows and OS/2; dynamic shared object or DSO under Unix-like systems) is a library intended for dynamic linking. Only a minimum amount of work is done by the linker when the executable file is created; it only records what library routines the program needs and the index names or numbers of the routines in the library. The majority of the work of linking is done at the time the application is loaded (load time) or during execution (run time). Usually, the necessary linking program, called a "dynamic linker" or "linking loader", is actually part of the underlying operating system. (However, it is possible, and not exceedingly difficult, to write a program that uses dynamic linking and includes its own dynamic linker, even for an operating system that itself provides no support for dynamic linking.)

Programmers originally developed dynamic linking in the Multics operating system, starting in 1964, and the MTS (Michigan Terminal System), built in the late 1960s.[8]

5.3 Optimizations

Since shared libraries on most systems do not change often, systems can compute a likely load address for each shared library on the system before it is needed, and store that information in the libraries and executables. If every shared library that is loaded has undergone this process, then each will load at its predetermined address, which speeds up the process of dynamic linking. This optimization is known as prebinding in OS X and prelinking in Linux. Disadvantages of this technique include the time required to precompute these addresses every time the shared libraries change, the inability to use address space layout randomization, and the requirement of sufficient virtual address space for use (a problem that will be alleviated by the adoption of 64-bit architectures, at least for the time being).

5.4 Locating libraries at run time

Loaders for shared libraries vary widely in functionality. Some depend on the executable storing explicit paths to the libraries. Any change to the library naming or layout of the file system will cause these systems to fail. More commonly, only the name of the library (and not the path) is stored in the executable, with the operating system supplying a method to find the library on-disk based on some algorithm.

If a shared library that an executable depends on is deleted, moved, or renamed, or if an incompatible version of the library is copied to a place that is earlier in the search, the executable would fail to load. This is called *Dependency hell* existing on many platforms. The (infamous) Windows variant is commonly known as DLL hell. This problem cannot occur if each version of each library is uniquely identified and each program references libraries only by their full unique identifiers. The "DLL hell" problems with earlier Windows versions arose from using only the names of libraries, which were not guaranteed to be unique, to resolve dynamic links in programs. (To avoid "DLL hell", later versions of Windows rely largely on options for programs to install private DLLs—essentially a partial retreat from the use of shared libraries—along with mechanisms to prevent replacement of shared system DLLs with earlier versions

of them.)

5.4.1 Microsoft Windows

Microsoft Windows checks the registry to determine the proper place to load DLLs that implement COM objects, but for other DLLs it will check the directories in a defined order. First, Windows checks the directory where it loaded the program (*private DLL*[7]); any directories set by calling the SetDllDirectory() function; the System32, System, and Windows directories; then the current working directory; and finally the directories specified by the PATH environment variable.[9] Applications written for the .NET Framework framework (since 2002), also check the Global Assembly Cache as the primary store of shared dll files to remove the issue of DLL hell.

5.4.2 OpenStep

OpenStep used a more flexible system, collecting a list of libraries from a number of known locations (similar to the PATH concept) when the system first starts. Moving libraries around causes no problems at all, although users incur a time cost when first starting the system.

5.4.3 Unix-like systems

Most Unix-like systems have a "search path" specifying file system directories in which to look for dynamic libraries. Some systems specify the default path in a configuration file; others hard-code it into the dynamic loader. Some executable file formats can specify additional directories in which to search for libraries for a particular program. This can usually be overridden with an environment variable, although it is disabled for setuid and setgid programs, so that a user can't force such a program to run arbitrary code with root permissions. Developers of libraries are encouraged to place their dynamic libraries in places in the default search path. On the downside, this can make installation of new libraries problematic, and these "known" locations quickly become home to an increasing number of library files, making management more complex.

5.5 Dynamic loading

Main article: Dynamic loading

Dynamic loading, a subset of dynamic linking, involves a dynamically linked library loading and unloading at run time on request. Such a request may be made implicitly at compile time or explicitly at run time. Implicit requests are made at compile time when a linker adds library references that include file paths or simply file names. Explicit requests are made when applications make direct calls to an operating system's API at run time.

Most operating systems that support dynamically linked libraries also support dynamically loading such libraries via a run-time linker API. For instance, Microsoft Windows uses the API functions LoadLibrary, LoadLibraryEx, FreeLibrary and GetProcAddress with Microsoft Dynamic Link Libraries; POSIX based systems, including most UNIX and UNIX-like systems, use dlopen, dlclose and dlsym. Some development systems automate this process.

6 Object and class libraries

Although originally pioneered in the 1960s, dynamic linking did not reach operating systems used by consumers until the late 1980s. It was generally available in some form in most operating systems by the early 1990s. During this same period, object-oriented programming (OOP) was becoming a significant part of the programming landscape. OOP with runtime binding requires additional information that traditional libraries don't supply. In addition to the names and entry points of the code located within, they also require a list of the objects they depend on. This is a side-effect of one of OOP's main advantages, inheritance, which means that parts of the complete definition of any method may be in different places. This is more than simply listing that one library requires the services of another: in a true OOP system, the libraries themselves may not be known at compile time, and vary from system to system.

At the same time many developers worked on the idea of multi-tier programs, in which a "display" running on a desktop computer would use the services of a mainframe or minicomputer for data storage or processing. For instance, a program on a GUI-based computer would send messages to a minicomputer to return small samples of a huge dataset for display. Remote procedure calls already handled these tasks, but there was no standard RPC system.

Soon the majority of the minicomputer and mainframe vendors instigated projects to combine the two, producing an OOP library format that could be used anywhere. Such systems were known as **object libraries**, or **distributed objects**, if they supported remote access (not all did). Microsoft's COM is an example of such a system for local use, DCOM a modified version that supports remote access.

For some time object libraries held the status of the "next big thing" in the programming world. There were a number of efforts to create systems that would run across platforms, and companies competed to try to get developers locked into their own system. Examples include IBM's System Object Model (SOM/DSOM), Sun Microsystems' Distributed Objects Everywhere (DOE),

NeXT's Portable Distributed Objects (PDO), Digital's ObjectBroker, Microsoft's Component Object Model (COM/DCOM), and any number of CORBA-based systems.

After the inevitable cooling of marketing hype, object libraries continue to be used in both object-oriented programming and distributed information systems. **Class libraries** are the rough OOP equivalent of older types of code libraries. They contain classes, which describe characteristics and define actions (methods) that involve objects. Class libraries are used to create instances, or objects with their characteristics set to specific values. In some OOP languages, like Java, the distinction is clear, with the classes often contained in library files (like Java's JAR file format) and the instantiated objects residing only in memory (although potentially able to be made persistent in separate files). In others, like Smalltalk, the class libraries are merely the starting point for a system image that includes the entire state of the environment, classes and all instantiated objects.

7 Remote libraries

Another solution to the library issue comes from using completely separate executables (often in some lightweight form) and calling them using a remote procedure call (RPC) over a network to another computer. This approach maximizes operating system re-use: the code needed to support the library is the same code being used to provide application support and security for every other program. Additionally, such systems do not require the library to exist on the same machine, but can forward the requests over the network.

However, such an approach means that every library call requires a considerable amount of overhead. RPC calls are much more expensive than calling a shared library that has already been loaded on the same machine. This approach is commonly used in a distributed architecture that makes heavy use of such remote calls, notably client-server systems and application servers such as Enterprise JavaBeans.

8 Code generation libraries

Code generation libraries are high-level APIs that can generate or transform byte code for Java. They are used by aspect-oriented programming, some data access frameworks, and for testing to generate dynamic proxy objects. They also are used to intercept field access.[10]

9 File naming

- Most modern Unix-like systems

  The system stores libfoo.a and libfoo.so files in directories such as /lib, /usr/lib or /usr/local/lib. The filenames always start with lib, and end with a suffix of .a (archive, static library) or of .so (shared object, dynamically linked library). Some systems might have multiple names for the dynamically linked library, with most of the names being names for symbolic links to the remaining name; those names might include the major version of the library, or the full version number; for example, on some systems libfoo.so.2 would be the filename for the second major interface revision of the dynamically linked library libfoo. The .la files sometimes found in the library directories are libtool archives, not usable by the system as such.

- OS X

  The system inherits static library conventions from BSD, with the library stored in a .a file, and can use .so-style dynamically linked libraries (with the .dylib suffix instead). Most libraries in OS X, however, consist of "frameworks", placed inside special directories called "bundles" which wrap the library's required files and metadata. For example, a framework called MyFramework would be implemented in a bundle called MyFramework.framework, with MyFramework.framework/MyFramework being either the dynamically linked library file or being a symlink to the dynamically linked library file in MyFramework.framework/Versions/Current/MyFramework.

- Microsoft Windows

  Dynamic-link libraries usually have the suffix *.DLL,[11] although other file name extensions may identify specific-purpose dynamically linked libraries, e.g. *.OCX for OLE libraries. The interface revisions are either encoded in the file names, or abstracted away using COM-object interfaces. Depending on how they are compiled, *.LIB files can be either static libraries or representations of dynamically linkable libraries needed only during compilation, known as "import libraries". Unlike in the UNIX world, which uses different file extensions, when linking against .LIB file in Windows one must first know if it is a regular static library or an import library. In the latter case, a .DLL file must be present at run time.

10 See also

- Code reuse
- Linker (computing)
- Loader (computing)
- Dynamic-link library
- Object file
- Plug-in
- Prebinding
- Static library
- Runtime library
- Visual Component Library (VCL)
- Component Library for Cross Platform (CLX)
- Lazarus Component Library (LCL)
- C standard library
- Java Class Library
- Framework Class Library
- Generic programming (used by the C++ standard library)
- soname
- Method stub

11 Notes

[1] It was possible earlier between, e.g., Ada subprograms.

[2] Some older systems, e.g., Burroughs MCP, Multics, also have only a single format for executable files, regardless of whether they are shared.

12 References

[1] Wexelblat, Richard (1981). *History of Programming Languages*. ACM Monograph Series. New York, NY: Academic Press (A subsidiary of Harcourt Brace). p. 369. ISBN 0-12-745040-8.

[2] Wexelblat, *op. cit.*, p. 274

[3] Wexelblat, *op. cit.*, p. 258

[4] Wilson, Leslie B.; Clark, Robert G. (1988). *Comparative Programming Languages*. Wokingham, England: Addison-Wesley. p. 126. ISBN 0-201-18483-4.

[5] Wilson and Clark, *op. cit.*, p. 52

[6] Wexelblat, *op. cit.*, p. 716

[7] Anderson, Rick (2000-01-11). "The End of DLL Hell". microsoft.com. Archived from the original on 2001-06-05. Retrieved 2012-01-15. *Private DLLs are DLLs that are installed with a specific application and used only by that application.*

[8] "A History of MTS". *Information Technology Digest* **5** (5).

[9] "Dynamic-Link Library Search Order". *Microsoft Developer Network Library*. Microsoft. 2012-03-06. Retrieved 2012-05-20.

[10] "Code Generation Library". http://sourceforge.net/: Source Forge. Retrieved 2010-03-03. Byte Code Generation Library is high level API to generate and transform JAVA byte code. It is used by AOP, testing, data access frameworks to generate dynamic proxy objects and intercept field access.

[11] Bresnahan, Christine; Blum, Richard (2015). *LPIC-1 Linux Professional Institute Certification Study Guide: Exam 101-400 and Exam 102-400*. John Wiley & Sons. p. 82. ISBN 9781119021186. Retrieved 2015-09-03. Linux shared libraries are similar to the dynamic link libraries (DLLs) of Windows. Windows DLLs are usually identified by .dll filename extensions.

13 External links

- Shared Libraries - 'Linkers and Loaders' by John R. Levine
- Dynamic Linking and Loading - 'Linkers and Loaders' by John R. Levine
- Article *Beginner's Guide to Linkers* by David Drysdale
- Article *Faster C++ program startups by improving runtime linking efficiency* by Léon Bottou and John Ryland
- How to Create Program Libraries by Baris Simsek
- LIB BFD - the Binary File Descriptor Library
- 1st Library-Centric Software Design Workshop LCSD'05 at OOPSLA'05
- 2nd Library-Centric Software Design Workshop LCSD'06 at OOPSLA'06
- How to create shared library(with much background info)
- Anatomy of Linux dynamic libraries

14 Text and image sources, contributors, and licenses

14.1 Text

- **Library (computing)** *Source:* https://en.wikipedia.org/wiki/Library_(computing)?oldid=680650831 *Contributors:* Damian Yerrick, Zundark, Stephen Gilbert, XJaM, Maury Markowitz, B4hand, KF, Frecklefoot, Edward, K.lee, Lir, Nixdorf, Pnm, Danhicks, TakuyaMurata, Minesweeper, Mac, Nanshu, Julesd, IMSoP, Emperorbma, Dysprosia, Greenrd, Zoicon5, Wernher, Khym Chanur, Mignon~enwiki, Jeffq, Robbot, Mountain, Nurg, Wjhonson, Rfc1394, Rholton, Jor, Salty-horse, Tobias Bergemann, ManuelGR, Endx7, SamB, Haeleth, Art Carlson, Alterego, CyborgTosser, Mboverload, Prosfilaes, Rchandra, Madoka, Vadmium, Beland, Am088, Secfan, Maximaximax, Icairns, Simoneau, Sam Hocevar, Urhixidur, Bluefoxicy, Abdull, Omassey, Mormegil, Poccil, Jkl, Jordancpeterson, Jamadagni, Gronky, CanisRufus, Kop, Art LaPella, Bobo192, R. S. Shaw, Pentap101, Minghong, Philipdl71, Officiallyover, Melah Hashamaim, Resipsa, Guy Harris, Sligocki, Gbeeker, Emvee~enwiki, Cburnett, Jakek101, MIT Trekkie, Alai, Mikenolte, Forderud, Kbolino, WayneMokane, Unixxx, Feezo, Brianwc, Madmardigan53, Ae-a, TheNightFly, Ruud Koot, Sega381, Btyner, Tlroche, Raffaele Megabyte, Justizin, Vegaswikian, Fred Bradstadt, Margosbot~enwiki, Felixdakat, Intgr, Tardis, Glenn L, Chobot, DVdm, YurikBot, Wavelength, Borgx, TexasAndroid, RobotE, FrenchIsAwesome, Piet Delport, Skydot, Manop, SteveLoughran, Mipadi, ZacBowling, Jpbowen, JulesH, Voidxor, Snarius, Bota47, JECompton, Lzyiii, Mugunth Kumar, CWenger, ArielGold, GrinBot~enwiki, SmackBot, IanVaughan, Sam Pointon, Aij, El Cubano, Thumperward, Stevage, Jerome Charles Potts, Nbarth, Torzsmokus, Ian Burnet~enwiki, Frap, Jsmethers, Juancnuno, Baris simsek, Azio, Emre D., Wonderstruck, Cybercobra, Harvestman, Daniel.Cardenas, Digana, Harryboyles, Gang65, Antonielly, NongBot~enwiki, Waggers, Nialsh, Norm mit, SimonD, Nemonemo~enwiki, CmdrObot, Fumblebruschi, AlbertSM, Wws, Tmn, Phatom87, Alanbly, Indeterminate, Msreeharsha, Christian75, Danogo, ColdShine, PamD, Poorleno, Pstanton, Hervegirod, DmitTrix, RobotG, MetaManFromTomorrow, Prolog, Medinoc, JAnDbot, DirtY iCE, Swpb, Gwern, Ian Bailey, Sigmundg, Ahzahraee, Hans Dunkelberg, Plasticup, Merceris, Ale2006, VolkovBot, Oleh Kernytskyi, Rei-bot, Jozue, Anna Lincoln, Onion Bulb, Fogbank74, Insanity Incarnate, Michael Safyan, Pjoef, Biasoli, S.Örvarr.S, SieBot, Prcr, BotMultichill, Jerryobject, YvesEau, OKBot, Svick, Jkonline, ClueBot, Drmies, Justin545, GlasGhost, DanielPharos, 1ForTheMoney, Crb136, PeterFisk, Galzigler, Addbot, Proofreader77, AkhtaBot, Karl gregory jones, IOLJeff, Tenth Plague, Zorrobot, Fiftyquid, Jarble, Legobot, Jobinj00770, Luckas-bot, Yobot, Ptbotgourou, Vanished user rt41as76lk, Golftheman, AnomieBOT, Materialscientist, Nhantdn, ArthurBot, Xqbot, 4twenty42o, Gtfjbl, Br77rino, Almabot, Chatul, FrescoBot, Ashishtanwer, Sae1962, Umawera, Citation bot 1, Winterst, RedBot, Jandalhandler, Aoidh, Limited Atonement, Epsota24, EmausBot, WikitanvirBot, Ajraddatz, Noloader, NoisyJinx, Klbrain, Moswento, JadAizarani, Josve05a, ClueBot NG, Jack Greenmaven, Shaddim, Yttrill, O.Koslowski, Helpful Pixie Bot, Faus, Tom Pippens, Alex.Cham, Codename Lisa, Andyhowlett, 123abc12, OhioGuy814, Seanhalle, My name is not dave, RichSaunders, Lagoset, KasparBot, XJohnSanderson and Anonymous: 202

14.2 Images

- **File:Ogg_vorbis_libs_and_application_dia.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/d/df/Ogg_vorbis_libs_and_application_dia.svg *License:* CC BY-SA 3.0 *Contributors:* self-made, based on file:Libs_dia.png *Original artist:* Kővágó Zoltán (DirtY iCE)

14.3 Content license

- Creative Commons Attribution-Share Alike 3.0