A.1 — Static and dynamic libraries

A **library** is a package of code that is meant to be reused by many programs. Typically, a C++ library comes in two pieces:

1. A header file that defines the functionality the library is exposing (offering) to the programs using it.
2. A precompiled binary that contains the implementation of that functionality pre-compiled into machine language.

Some libraries may be split into multiple files and/or have multiple header files.

Libraries are precompiled for several reasons. First, since libraries rarely change, they do not need to be recompiled often. It would be a waste of time to recompile the library every time you wrote a program that used them. Second, because precompiled objects are in machine language, it prevents people from accessing or changing the source code, which is important to businesses or people who don't want to make their source code available for intellectual property reasons.

There are two types of libraries: static libraries and dynamic libraries.

A **static library** (also known as an **archive**) consists of routines that are compiled and linked directly into your program. When you compile a program that uses a static library, all the functionality of the static library that your program uses becomes part of your executable. On Windows, static libraries typically have a .lib extension, whereas on Linux, static libraries typically have an .a (archive) extension. One advantage of static libraries is that you only have to distribute the executable in order for users to run your program. Because the library becomes part of your program, this ensures that the right version of the library is always used with your program. Also, because static libraries become part of your program, you can use them just like functionality you've written for your own program. On the downside, because a copy of the library becomes part of every executable that uses it, this can cause a lot of wasted space. Static libraries also can not be upgraded easy -- to update the library, the entire executable needs to be replaced.

A **dynamic library** (also called a **shared library**) consists of routines that are loaded into your application at run time. When you compile a program that uses a dynamic library, the library does not become part of your executable -- it remains as a separate unit. On Windows, dynamic libraries typically have a .dll (dynamic link library) extension, whereas on Linux, dynamic libraries typically have a .so (shared object) extension. One advantage of

dynamic libraries is that many programs can share one copy, which saves space. Perhaps a bigger advantage is that the dynamic library can be upgraded to a newer version without replacing all of the executables that use it.

Because dynamic libraries are not linked into your program, programs using dynamic libraries must explicitly load and interface with the dynamic library. This mechanism can be confusing, and makes interfacing with a dynamic library awkward. To make dynamic libraries easier to use, an import library can be used.

An **import library** is a library that automates the process of loading and using a dynamic library. On Windows, this is typically done via a small static library (.lib) of the same name as the dynamic library (.dll). The static library is linked into the program at compile time, and then the functionality of the dynamic library can effectively be used as if it were a static library. On Linux, the shared object (.so) file works as both a dynamic library and an import library. Most linkers can build an import library for a dynamic library when the dynamic library is created.

Installing and using libraries

Now that you know about the different kinds of libraries, let's talk about how to actually use libraries in your program. Installing a library in C++ typically involves 4 steps:

1. Acquire the library. The best option is to download a precompiled package for your operating system (if it exists) so you do not have to compile the library yourself. If there is not one provided for your operating system, you will have to download a source-code-only package and compile it yourself (which is outside of the scope of this lesson). On Windows, libraries are typically distributed as .zip files. On Linux, libraries are typically distributed as packages (e.g. .RPM). Your package manager may have some of the more popular libraries (e.g. SDL) listed already for easy installation, so check there first.
2. Install the library. On Linux, this typically involves invoking the package manager and letting it do all the work. On Windows, this typically involves unzipping the library to a directory of your choice. We recommend keeping all your libraries in one location for easy access. For example, use a directory called C:\Libs, and put each library in it's own subdirectory.
3. Make sure the compiler knows where to look for the header file(s) for the library. On Windows, typically this is the include subdirectory of the directory you installed the library files to (e.g. if you installed your library to C:\libs\SDL-1.2.11, the header files are probably in C:\libs\SDL-1.2.11\include). On Linux, header files are typically installed to /usr/include, which should already be part of your include file search path. However, if the files are installed elsewhere, you will have to tell the compiler where to find them.

4. Tell the linker where to look for the library file(s). As with step 3, this typically involves adding a directory to the list of places the linker looks for libraries. On Windows, this is typically the /lib subdirectory of the directory you installed the library files to. On Linux, libraries are typically installed to /usr/lib, which should already be a part of your library search path.

Once the library is installed and the IDE knows where to look for it, the following 3 steps typically need to be performed for each project that wants to use the library:

5. If using static libraries or import libraries, tell the linker which library files to link.
6. #include the library's header file(s) in your program. This tells the compiler about all of the functionality the library is offering so that your program will compile properly.
7. If using dynamic libraries, make sure the program knows where to find them. Under Linux, libraries are typically installed to /usr/lib, which is in the default search path after the paths in the `LD_LIBRARY_PATH` environment variable. On Windows, the default search path includes the directory the program is run from, directories set by calling SetDllDirectory(), the Windows, System, and System32 directories, and directories in the PATH environment variable. The easiest way to use a .dll is to copy the .dll to the location of the executable. Since you'll typically distribute the .dll with your executable, it makes sense to keep them together anyway.

Steps 3-5 involve configuring your IDE -- fortunately, almost all IDEs work the same way when it comes to doing these things. Unfortunately, because each IDE has a different interface, the most difficult part of this process is simply locating *where* the proper place to perform each of these steps is. Consequently, in the next few lessons in this section, we'll cover how to do all of these steps for both Visual Studio and Code::Blocks. If you are using another IDE, read both -- by the time you're done, you should have enough information to do the same with your own IDE with a little searching.