Lecture 11 - Libraries (last updated: 2023-10-15)

Why use libraries?

Libraries are often used to package definitions that are reusable across different software projects. For example, a library may contain functions for numerical computing (e.g., a statistics library), manipulating data structures (e.g., a graph library), or accessing system services (e.g., a GUI library).

Though it is technically possible to share libraries exclusively as source code, it is sometimes impractical to do so for a number of reasons:

- the library source code may be proprietary
- the library source code may add to the compilation time of its users
- the library source code may be implemented in a different language
- the library source code may be difficult to compile (e.g., it depends on build tools that are not widely available)

To avoid these issues, a developer may choose to distribute their library as a collection of header (.h) and pre-compiled object (.o) files. Those header files declare the name and types of library symbols, while the object files provide definitions for those symbols.

For example, let's say we are writing a basic library for some numeric computation; let's call this our "numbers" library. This library will provide two functions, whose declarations are exposed using a header file:

```
// numbers.h

/** Compute x to the power of n */
int power(int x, int n);

/** Returns non-zero if x is prime; returns zero otherwise. */
int is_prime(int x);
```

We implement each of these functions in a separate compilation unit:

```
// power.c

#include "numbers.h"

int power(int x, int n) {
    /* ... */
}

// prime.c

#include "numbers.h"

int is_prime(int x) {
    /* ... */
}
```

So the library's source code consists of numbers.h, power.c, and prime.c.

We compile the library's compilation units to object files, and distribute numbers.h (which the library user will include), as well as power.o and prime.o.

Now, the library user will #include the header file, like this:

```
#include <numbers.h>    // library declarations for power() and is_prime()

int main(void) {
    printf("5^3 = %d\n", power(5, 3));
    printf("363 is prime: %d\n", is_prime(363));
    return 0;
}
```

Archive files

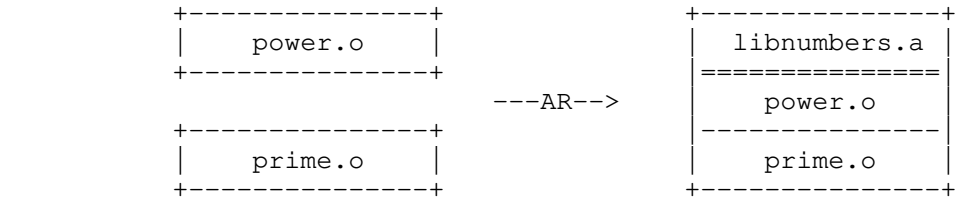
Instead of distributing a (potentially large) collection of object files, it is often more convenient to package them together as a single "archive" (.a) file.

Continuing our previous example: instead of having to distribute both power.o and prime.o, we can bundle them into a single archive file named "libnumbers.a" using the "ar" command:

```
ar rcs libnumbers.a power.o prime.o
```

The 'r' operation code tells ar to add the files power.o and prime.o to the archive (or replace them if they are already in there), the 'c' modifier tells ar to create the archive file if it doesn't already exist, and the 's' operation code tells ar to generate an index to the contents of the archive to speed up linking.

Visually:



Now, the library developer only needs to distribute two files: numbers.h and libnumbers.a (which contains power.o and prime.o). You can link directly with archive files; e.g., instead of running:

```
gcc myprogram.o prime.o power.o -o myprogram    # link with library objects
```

You can run;

```
gcc myprogram.o libnumbers.a -o myprogram       # link with library archive
```

The linker will recognize that libnumbers.a is an archive and use the object files inside it directly.

Using libraries

Library headers and archives are not typically installed in the same place as the source code using that library. For example, on Linux systems, library headers and archives are usually installed in /usr/include/ and /usr/lib/ respectively. For any library files installed in non-standard locations, you

will need to tell the compiler and linker where to look and what to look for, using the -I, -L, and -l flags.

The -I flag tells the compiler where to look for header files. This flag is usually written as -I<path> or -I <path>, which adds <path> to the so-called "include path". For example, if you run:

```
gcc -I/some/path -c foo.c
```

This tells gcc that when foo.c #includes <some-header.h>, it should also look for some-header.h in /some/path.

You may optionally include a space the space after the -I, and you can use relative paths, e.g.,

```
gcc -I /some/path -c foo.c

gcc -Isome/path -c foo.c

gcc -I../some/path -c foo.c
```

These are all valid include paths (though they specify different paths).

The -L flag tells the linker where to look for archive files, and works similarly to the -I flag: it is usually written as -L<path> or -L <path>.

Though -L tells the linker where to look, it doesn't tell the linker what to look for; this is the job of the -l flag. Writing -l<lib-name> will tell the linker to link in lib<lib-name>.a. For example, if you write:

```
gcc  -Lsome/path  foo.o bar.o  -lbaz
```

the linker will look for libbaz.a in some/path and try to link it with foo.o and bar.o. Note that in this example, gcc is acting as a linker, not a C compiler. The -L flags should appear before the object files, while -l flags must appear after.

You can pass multiple -I and -L flags to tell the compiler and linker to look in multiple locations for header and archive files, respectively. Similarly, you can pass multiple -l flags to link in multiple libraries.

The -I, -L, and -l flags actually build on implicit include paths, library paths, and libraries that the compiler/linker already uses by default. For example, -I tells gcc to look for header files in the location you specify in addition to standard locations like /usr/include/ (where it finds header files like stdio.h and stdlib.h). In a similar vein, the -L flag tells gcc to look beyond /usr/lib/ for library archive files, and the -l flag tells gcc to link in more libraries in addition to the standard C library (which defines standard functions like strlen() and malloc()).

Static vs dynamic linking (optional)

The libaries described above are "static" libraries; they are linked into executable before runtime, and work pretty much the same way as object files. However, if the same static library is used by many different executables, then the same code will be duplicated among those executables, taking up more space and memory than needed.

In contrast to static libraries, dynamic libraries are not linked with executables until runtime. Since dynamic libraries are stored separately from the executables that use them, a single copy of each dynamic library may be shared among many executables, reducing its footprint.

Another benefit of dynamic libraries is that they can be upgraded separately from the executables that use them, instead of having to re-link each executable with an updated static library. This is especially useful in scenarios where security is a priority: for example, if a critical vulnerability is discovered in libssl (used for secure networking), those vulnerabilities can be immediately fixed and deployed as a shared library, without waiting for maintainers to rebuild each executable with an updated libssl static library.

There are several downsides to dynamic libraries. The first is that dynamic linking is more complex and system-dependent than static linking, since it relies on the operating system to inject library code into a running process. Dynamic linking also comes with a performance cost, since it is necessarily done at runtime, and may prevent link-time optimizations (LTOs) otherwise available at compile time. Finally, because dynamic libraries are stored separately from executables, they can make testing, versioning, and deployment more complicated.