Static library

Installed into the executable before the program can be run

Library contains a collection of ordinary object files (each with a .o suffix → archive with a .a suffix)

Used sometimes when a provider doesn't want to supply the source code
If there are many libraries/functions this can create a very large executable file
Link to library using the -l option
    Be sure to use the correct order for compilation -- file, then flag

Shared Libraries (Also known as dynamically linked libraries)

Loaded at program start-up and shared between programs
A large program will start and load several libraries
    These will be shared automatically by any operations in the program
You can update libraries with optimal support
Override specific libraries or even functions in a library when executing a program

soname: contains directory to library→ lib(libname).so.(version number)

Real name: filename containing the library's real code
    This is the same as the soname plus a .(release number) // This is optional
Linker name: used with -L, just the soname without the version number
The libraries don't become part of the executable
The object file contains dynamic library references, assuming that the files will be there
This allows for a change to the library without a necessity for recompilation
Command "ldd" will list the dependencies of the file within the shared library
Has the extension .so

Dynamic library

Can be loaded and used at times other than the startup of a program
Both Static and shared can be dynamic

Library structure

All have src, lib, and include files
src=cpp files
include=header files, contains info for the references to the library files
lib=compiled binary files
    For static the extension becomes .a
    For dynamic the extension become .so (shared object)
The name of a library is lib(libraryName).so or lib(libraryName).a in the lib folder
Usr directory just means that it is local to the computer accessible to all users
    System files and libraries are downloaded to the usr/lib directory
    Third part/downloaded libraries should go to the usr/local/lib directory
        It doesn't appear that this computer follows these rules
    An /etc/ld.so.conf file contains the directories searched for libraries
-L tells the compiler where to find the library (directory)
-l tells the compiler the name of the library
    When using these, dont put a space between the -L/-l and the next argument
        -l/path/to/library
When compiling and linking at the same time (not just linking to .o files), you must use the flag -Wl, -L… -Wl,-l…. These are both -W(ell)
#include will work if the file is within the same directory

If not use -I/path/to/file (dash eye) to include the file

Paths can be absolute or relative

Absolute: with reference to the root directory

Relative: with reference to the current directory

Linking two .o files creates a single executable effectively the same as a single long file

Using the <> notation searches for the library in the includes sections

Using the “” notation searches for the library in the local directory and also the includes section (dependant on the compiler)

Hands on example

C++ Compiling

Major steps to creating executable

Preprocessor

Takes care of preprocessor directives

Compiler

Compiler generates the .o file

Linker

Linker generates the .exe file

C++ compiles/looks at only one file at a time

Keep this in mind with definitions etc..

https://www.youtube.com/watch?v=Wa6irrxFG88&list=PLRwVmtr-pp05Bl0j6lwXd8tU754nUEB5P&index=1

When do you need to link to Libraries and when do you just include

http://stackoverflow.com/questions/7096152/c-difference-between-linking-library-and-adding-include-directories

Including a header file will replace the header reference with add definitions so that your code will run

If multiple files include the same header files

Recursive definitions can be prevented with header guards

Include files contain the declarations and prototypes of variables and functions

It contains all that a compiler will need

The linkable libraries contain the actual object code of the functions of the library

#include "file" vs #include <file>

The difference is in the location where the preprocessor searches for the included file.

#include "filename" the preprocessor searches in the same directory as the file and any included directories containing the directive. This method is normally used to include programmer-defined header files.

#include <filename> the preprocessor searches in an implementation dependent manner, normally in search directories pre-designated by the compiler/IDE. This method is normally used to include standard library header files.

Linking to libraries:

Static:

The executable will include chunks of library code within it

In the linking step the required sections of code in the (.a) library will be added to the executable
Increases the size of the executable
Updates in the .a will require recompilation of all executables using that library
When you -l you are including everything in the executable/compilation stage

Dynamic libraries (Shared object libraries):
the executable will contain references to the library address in memory
Library is necessary in compile and runtime
No problem of size, updates easier
(.so)
All C++ STL (standard libraries) are .so format

Static:

- Compile: `cc -Wall -c ctest1.c ctest2.c`
  Compiler options:
  - -Wall: include warnings. See man page for warnings specified.
- Create library "libctest.a": `ar -cvq libctest.a ctest1.o ctest2.o`
- List files in library: `ar -t libctest.a`
- Linking with the library:
  - `cc -o executable-name prog.c libctest.a`
  - `cc -o executable-name prog.c -L/path/to/library-directory -lctest`

Link by including the library name in the compilation or using the -L -l format
Shared object libraries only use the -L -l format
If you wish to do compiling and linking at the same time, use the -WL, -Wl flags instead

Shared Object Library:
With a shared object library there is often a symbolic link in the filesystem to the library
This is done so that you can use the syntax -L PATH -l name and disregard the version numbers specified at the end of the soname. That way if the library is updated, the symbolic links can stay in tack

```
ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so.1
ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so
```

Library Links:

- The link to `/opt/lib/libctest.so` allows the naming convention for the compile flag `-lctest` to work.
- The link to `/opt/lib/libctest.so.1` allows the run time binding to work. See dependency below.

See what other libraries a library depends on with the ldd commna

```
[prompt]$ ldd libname-of-lib.so
    libglut.so.3 => /usr/lib64/libglut.so.3 (0x00007fb582b74000)
    libGL.so.1 => /usr/lib64/libGL.so.1 (0x00007fb582857000)
    libX11.so.6 => /usr/lib64/libX11.so.6 (0x00007fb582518000)
    libIL.so.1 (0x00007fa0f2c0f000)
    libcudart.so.4 => not found
```

Produces error...
Fix:

- Add the unresolved library path in `/etc/ld.so.conf.d/name-of-lib-x86_64.conf` and/or `/etc/ld.so.conf.d/name-of-lib-i686.conf`
  Reload the library cache (`/etc/ld.so.cache`) with the command: `sudo ldconfig`
  or
- Add library and path explicitly to the compiler/linker command: `-lname-of-lib -L/path/to/lib`
  or
- Add the library path to the environment variable to fix runtime dependency:
  `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/lib`

The /etc/ld.so.conf file
This file will have a list of directories to be searched for shared libraries
With ldconfig this is run and all symbols and specifies all dynamic link libraries etc.
    This is then stored in /etc/ld.so.cache
This may mean that if a new library is downloaded you may need to run ldconfig

LD_LIBRARY_PATH
Is a smaller set of paths which is an environmental variable
Order of library search:
    -L, -l
    LD_LIBRARY_PATH
    /etc/ld.so.conf file (from the cache, refreshed with ldconfig)

Makefiles
Variables (macros) are defined at the top
    Just using the “CC=g++” style
    CC is used to denote the compiler
    CFLAGS is used to denote flags for the code
    LFLAGS is used to denote library flags
    DEBUG used for debugging ex. -g
    DEPS is used to define the dependencies for a file ex. the header file for a .cpp
    % is used as a wildcard ex. “%.o” refers to all object files
    -@ is the name of the file being generated
        -o $@ (allows for easy renaming of the output file)
    $< refers to the first file to the right of the output file
        Ex. hello.o: hello.c hello.h
            Gcc -c $< -o $@
            $< is replaced by hello.c
    $^ is replaced by all files to the right of the output file
        hello.c, hello.h

Cmake
Cmake list creates a makefile
    CMakeLists.txt --- contains makefile instructions
    To run:
    cmake .
    (in the current directory)

Shell scripting
Just executes the commands listed so that you don’t have to do them repeatedly
Pipe output to a file (both cout and cerr)
    ./runtests.sh 2>&1 | tee output.txt

Path variable

The ~/.bashrc file is executed upon each new instance of the terminal
In it the $PATH variable can be altered
This variable is used to define directories searched when a command is entered to the command line. The binary files must be specified

Further Reading

http://tldp.org/HOWTO/Program-Library-HOWTO/index.html