# Explain include header files, linking library, compilation/building and running?

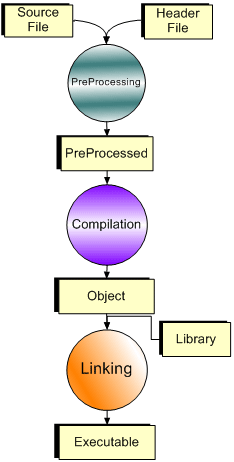
**Header files** - #include preprocesor keyword is used to include a header file in a C program. Header files are simple text file, includes declaration/prototype of functions and variables, macros etc. Header files does not include any function definitions except static or inline type.

**Library** - We often write our program with the help of API or with function calls which are imported from other modules like C Library. Compiler contains several library files are to resolve external symbols during linking process. These library files are nothing but pre-compiled object binaries given by compiler tool chain or by external third party vendors. Library file are in compiled binary format and thus source are protected.

There are two types of libraries: Static and dynamic. Static libraries are like complied object code archived inside a single tar file. Linker takes only the object code during linking time. This is same like using another object code of a source file except the source C file is not given.

Source code for libraries are not given in the compiler. The reason can be the copy-right. Static link has one disadvantage as it increases the code size and code sharing between two modules is not possible. Dynamic link libraries have these advantages. DLL executable comes in a separate binary file and it will be loaded in a common area of the OS and can be shared across many applications.

**Compilation**- Compiler generates object code from source file. Object codes are intermediate code needed by the linker. Compilation has three stages Preprocessing, Compilation and optimization. Preprocessing stage includes header file expansion, macro/preprocessor replacement etc. After preprocessing a single file gets expanded with headers and macros. Now it passes to compiler. Compiler does syntax checking, validation and generates object code. Compiler also works in conjunction with optimizer. The job of optimizer is to arrange the code or steps in such a way to reduce size or execution time. For example to optimize execution time optimizer often takes a frequently used local variable in one of CPU register. This ensures speed optimization. Again optimizer sometimes changes logic flow of the code to make the output size smaller. Here it is clear that optimizer alters program a bit thus line by like stepping info may not be generated when optimization has been applied.



**Linking** Linking is the final step of the build process. Linker takes all object files and combined into one output binary. Linker links external symbols between these object files and also arranges code, rodata, data and BSS segments and generates a binary in the form of formats often known as a.out and ELF . Now it is clear that linker checks the definition all symbols which is there in all object files but what will happen to the symbols which are imported from external libraries. For these external symbols linker needs library files. There are mainly two types of library files - Static and dynamic. Static libraries are same as object files archived in a container. Thus linking to static library is same as linking to an object file. Dynamic libraries are compiled and linked binary file. There is concept of exporting table for symbols. Dynamic link library exports a set of function symbols through an export table. An application or C program imports these external symbols by means of import table. During linking linker checks the symbol if it is imported from a DLL it creates an import table and there it writes the DLL name and function entry point. This way linker links these symbols. It is the job of the loader of load the imported DLL first into the memory and checks the linkage then load the source application and links the import table with the exported function from DLL. This is called dynamic linking. Program loader does this process.

**Running**- We execute a program in shell and we get the output in the console. This seems very simple but a huge process is involved in this. There is a loader for a.out and ELF files. It reads header of the binary and checks and validates code, rodata, data, BSS, import, export table etc. After this it loads corresponding segments in memory. If there are external symbols in import table it also loads all related dynamic link libraries if not loaded earlier. One thing to point here is loader also populates all the export entries of the DLL with proper loaded address. Then it links all the symbols of the external libraries to the loaded application. It links via import table. It fills all the entry address of the external symbols from the export table of the DLL to import table of the application. C runtime library (glibc) is one of such dynamic link library which takes these steps of loading. Now all the linking and loading process are done. So it creates task and copies these loaded address to code, rodata, data and BSS segments. It also creates a stack and heap space. There is a small startup code to do all the initialization. It generally starts with a symbol \_start with is hidden in linker/compiler. Then after all segment initialization it opens three files with descriptors stdin, stdout and stderr. These file descriptors are used for taking inputs and printing outputs and errors. Scanf/Printf etc library calls use these descriptors internally. Next thing is argument parsing and creating argument count(argc) and argument vector(argv). Then last step is calling \_main() with exit i.e. exit(main());. Now execution is inside our source code. We generally call exit(<status value>) to quit from the task or return the status in main which will actually feed to the exit call.