I tire of this one ... so let's see if we can agree on how things work.

First, let's agree that the steps of creating a running program are

i) **Translation** -- source code gets translated into an object file that is targeted at a particular machine. (I ignore here the issue of interpreters.) The object file contains instructions that implement the algorithmic intent of the source code plus additional information that will allow object files to be woven together into an executable.

N.B. An executable can be created from source code written in different languages -- thus having separate object files for each such source file is useful, apart from the fact that one does not need to recompile all source files if only one piece has been modified.

ii**) Linking / Loading** -- in this step, object files are 'woven together' i.e. linked/loaded to create an executable file (called a.out by default in Unix). References to symbols (one module calls the other as a function) are resolved, and the way the instructions will lay in memory as a stored program is determined. An interesting observation is that the command to create the executable (EX % gcc obj1.o obj2.o vs %gcc obj2.o obj1.o) can cause the same object files to be organized differently, although the way the program executes should not be sensitive to this matter).

iiii) **Execution** -- the executable becomes a running program (process) with execution beginning at 'main' (there can be only 1). The shell is provided the name of the file (including path) unless the file is already on the search path for the shell. Thus, frequently the path ./ (current directory) must be supplied to execute a program.

So what does this have to do with ‘include files’ typically named with a .h suffix ?

Reconsider the translation step. With C programs te #include directive is processed by cpp the ‘C pre processor’. This literally modifies the source code before translation. (If you wish to see the modifications there is a –E switch to gcc. ) A common modification is the inclusion of *prototypes* for functions that will be used in step ii) . Many of these are in the so-called ‘standard library’ which is so standard that you don’t even need to mention it when you do step ii.

A good example of this is printf() the function typically used for output.

#include <stdio.h> provides a prototype for printf(). If you do not include stdio.h you will receive a warning from the gcc translator. (Try It !)

Sometimes you must explicitly supply a library at step ii) in addition to providing a prototype in step i) . For example, I know that when we write threaded programs in CS 224 that both <pthread.h> and –lpthread must be supplied for steps i) and ii) respectively. The math library is supplied with a –lm switch at step ii) if needed while <math.h> is included in step i).

SO … headers do not supply the actual source code --- only the prototype. OK ?