Review of Header Files and Separate Compilation:

It's valuable in large projects to be able compile separate modules separately, i.e., to be able to recompile one module without recompiling all the rest.

For example, when you recompile your program, you don't want to have to recompile all the other code that you use (iostream, cmath, vector, Myro.h). These have already been compiled and are stored in "linkable" object form in subdirectories on the disk.

Another reason for separate compilation is so that libraries can be released in object form without revealing the source code. This is to protect intellectual property (IP).

Separate compilation doesn't change the fact that C++ (and most other "statically typed" languages) has to know the names and types of things in order to be able to compile. In other words, things still have to be declared before they are used.

This is the purpose of header files. They contain the declarations (the interface) to a module, but not the implementation. The interface is the only thing that both the implementer and the user of module need to know, so we put it in a header file.

In somewhat complex programs there is a danger of accidentally including the same header twice, which would redeclare stuff, which is an error. How can this happen? Suppose you #included iostream.h (because you do console I/O) and Myro.h (because you use the robots). Further suppose that Myro.h #included iostream.h (because it does console I/O). That would be an error, because the things in iostream are declared twice. With a large program it would be a nightmare trying to keep track of this. So to avoid it we use preprocessor directives that make work OK to include a header more than once.

Vectors of Objects (in particular Cards):

The first question: How are you going to represent a playing card? What is the most efficient and convenient way to represent cards? It depends on what you are planning to do with them.

One simple answer is to encode the suit and rank as integers. "Encode" here means translating some high-level, application oriented idea (e.g., suit, rank) into terms the machine understands (i.e., bits, but ints are pretty close to bits). The machine representation could be any already defined type. We're translating between human and machine understandable representations.

For example, encode the suits Clubs, Diamonds, Hearts, Spades, as the integers 0, 1, 2, 3 respectively. Why this order? It's arbitrary, but it would work good for bridge. It also happens to be alphabetical order so it's easy to remember.

What about rank. The ranks in their normal order are Ace, Two, Three, …, Ten, Jack, Queen, King. It's more intuitive to start with Ace = 1, Two = 2, etc., rather than Ace = 0, Two = 1, etc.

So I define a struct (or class) called Card, with two fields (member variables, instance variables), suit and rank.

The natural way to represent ordered sets of things (e.g., Cards) is as vectors of those things. Therefore the natural way to represent a deck of cards (of any size) is as a vector of Cards.