Quick glossary for the following explanation:

- Foo, Bar, and Baz are the primary fake-names I'll be using to explain includes.
- A *declaration* is something like `class Foo;` - this lets the compiler know that this class exists and isn't a typo or something. You can have many declarations for a class across many files, since they only let the compiler know something is intentional and does exist, even if the compiler doesn't know the details yet.
- A *definition* is something like `class Foo { ... };` - note the braces. This lets the compiler know that this is the one and only definition of a class and all its methods/variables/etc. You can only have one definition for a class that gets compiled since your program can only use one of them. The compiler cannot just check to see if the two compiled definitions are the same and discard one of them due to a lot of reasons.

There's a fun thing with C++ where only things that are declared above wherever the compiler is working at are available, which is why #includes are stuck at the top of the file. However, there can be issues with circular or double includes (i.e. Foo.h includes Bar.h, and the Bar class includes a Foo member variable, or Baz includes both Foo and Bar), which leads to multiple definitions or an include-cycle. This is what includeguards (#ifndef HEADER_NAME; #define HEADER_NAME; ... #endif) do - they make things only get defined once so that there's no compilation errors.

Let's say that we have the following classes:

- Foo: contains a Bar and a Baz
- Bar: contains a pointer to the Foo holding it
- Baz: contains a pointer to the Foo holding it

When you #include a file, the C++ compiler will literally just copy the contents of the included file over the #include directive. It'll then keep parsing down through the file, expanding #includes as they get to them. Let's look at how this would work in the above scenario if each .h file just simply includes the others it needs without any forward declarations:

foo.h

```
#ifndef foo_h
#define foo_h
#include "bar.h"
#include "baz.h"
class Foo {
 private:
 Bar _bar;
 Baz _bar;
};
#endif
```

bar.h

```
#ifndef bar_h
#define bar_h
#include "foo.h"
class Bar {
 private:
 Foo* foo;
}
#endif
```

baz.h

```
#ifndef baz_h
#define baz_h
#include "foo.h"
class Baz {
 private:
 Foo* foo;
}
#endif;
```

The include guards are necessary here to prevent circular includes. However, let's look at what this actually turns into when the C++ compiler actually tries to parse the foo.h header:

```
#ifndef foo_h
#define foo_h
#include "bar.h"
| #ifndef bar_h
| #define bar_h
| #include "foo.h"
|
| class Bar {
|   private:
|   Foo* foo;
| };
| #endif
#include "baz.h"
| #ifndef baz_h
| #define baz_h
| #include "foo.h"
|
| class Baz {
|   private:
|   Foo* foo;
| };
| #endif

class Foo {
  private:
  Bar _bar;
  Baz _bar;
};
#endif
```

foo.h will include bar.h, but then the foo.h inside that won't do anything - the includeguard will prevent that. This means that the bar.h won't actually have the foo.h include that it needs! When the compiler reaches the line in bar.h with the `Foo* foo;` it won't know what `Foo` is (since there hasn't been a matching declaration or definition yet!) and it will error out.

The way to fix these types of errors is to forward-declare all the classes in a header file (say, foo.h) before any #include directives, so that any include below it will have all the declarations from foo.h, since it wouldn't normally be able to compile it! Here's a fixed version of the above example:

foo.h

```
#ifndef foo_h
#define foo_h
class Foo;
#include "bar.h"
#include "baz.h"
class Foo {
 private:
 Bar _bar;
 Baz _bar;
};
#endif
```

bar.h

```
#ifndef bar_h
#define bar_h
class Bar;
#include "foo.h"
class Bar {
 private:
 Foo* foo;
}
#endif
```

baz.h

```
#ifndef baz_h
#define baz_h
class Baz;
#include "foo.h"
class Baz {
 private:
 Foo* foo;
}
#endif;
```

This would parse out as the following when the compiler reads through foo.h:

```
#ifndef foo_h
#define foo_h
class Foo;

#include "bar.h"
| #ifndef bar_h
| #define bar_h
| class Bar;
|
| #include "foo.h"
|
| class Bar {
|   private:
|   Foo* foo;
| };
| #endif
#include "baz.h"
| #ifndef baz_h
| #define baz_h
| class Baz;
|
| #include "foo.h"
|
| class Baz {
|   private:
|   Foo* foo;
| };
| #endif

class Foo {
  private:
  Bar _bar;
  Baz _bar;
};
#endif
```

Now, when the compiler parses through foo.h, it has the Foo declaration before parsing bar.h and baz.h. It can then successfully parse the Bar and Baz class definitions, which lets the Foo class definition complete successfully.

In a lot of cases, it may be simply sufficient to have just a forward declaration for a class, and the full definition wouldn't be necessary. However, setting up the classes/headers like above makes it a lot easier to just #include and get the full definition (which is needed to access any member variables or functions).

One other tidbit piece of advice when dealing with including multiple headers: always check the first compile error, since something as small as a missing semicolon can cascade pretty hard. C++ compiler errors usually quickly descend into an unreadable mess, unfortunately.