By now, we've established that most real-world software projects involve multiple files of source code. And we've also established that Linux offers plenty of tools to support software development with multiple files. (Of course, nearly every other software development environment supports this, too!)

We've talked a little about *why* software is organized this way, and we've played with some of the tools that support it, but we haven't really talked about *how* to organize software projects.

Let's start by reviewing the why.

How many lines of source code are in your favorite piece of software? It's not always easy to tell, especially if the source code isn't public, but if your favorite piece of software is the Linux kernel, then the answer is: over 20 million lines of code, distributed across more than 40 thousand files. (According to wikipedia and this older report). And several thousand people are working on it right now, and many many more have worked on it in the past.

How is it even possible to organize a project on this scale? How do you organize source code across that many files? What design techniques can we employ to make this possible? Even if our project only has a handful of source files, these questions still apply.

Let's ask the question a different way: why isn't the source code for the Linux kernel in one big file?

You probably can think of some good answers to that question, maybe having to do with reducing your headaches as a programmer, or maybe you suspect your favorite editor might not handle a 20-million-line file very well. But are there any reasons beyond headache-reduction? Sure. Let's think about what we want out of software.

First, software needs to be correct and secure. That's true for the applications we use every day, and it's even more true of the software that operates communications systems, airplanes, and hospitals.

Second, software exists in an incredibly dynamic environment: companies grow and shrink, rise and fall, standards change, trends and expectations shift all the time. This is probably more true now than it's ever been in the past. Our software needs to be able to keep up—it needs to be easy to modify it.

Finally, there has always been a huge demand for new software, but writing new software is slow, very difficult, and expensive. So software should be as re-usable as possible.

You can probably think of even more characteristics of good software, but that's a pretty good start.

Let's ask our question again in terms of these goals: how should the source code for the Linux kernel be organized in order to support analyzing its correctness, making it easy to modify, and being able to re-use pieces of it? Hopefully, your intuition says that putting all the code in one big file is about the *worst* way to organize it. Hopefully, your intuition says, hmmm, all that stuff—looking at correctness and security, modifying and reusing code—would be way easier if it the code were broken up into a bunch of small chunks. It's much easier to understand — and analyze the correctness of — a small piece of software than a large complex application. It's much easier to change the behavior of an application by altering one of its components than by rewriting one large program. It's much easier to design small reusable software components than to try to adapt an entire application for another purpose.

OK, so how do we break a project up into small chunks?

Here's a really bad approach: start writing code. When you've written one thousand lines, stop, and divide the code into 10 files of 100 lines each.

What's bad about that? There's no *reason* for the contents of one of those 100-line files to be together. If our goals are correctness, modifiability, and reusability, then it would make sense for each of our small chunks to represent an element of the larger application—to put it another way, the code inside each chunk should be closely related, and the code in different chunks should *not* be closely related. (For example, the part of a web browser that receives data from the Internet is not closely related to the part that displays a web page on the screen—even though all the browser code is *somewhat* related, simply because it's all part of the same application.

In general, we're talking about organizing our code into *modules*, where each module contains all the code for one aspect of an application's behavior. Some programming languages explicitly include the idea of a module (sometimes called a package, or a unit), but C and C++ don't. Nonetheless, we can use the idea to guide how we organize our code, and we've actually already seen a lot of the programming techniques we'll use to create C++ code that looks a lot like modules.

We need three ideas to talk about modular design: specification, encapsulation, and orthogonality.

First, specification. You've probably heard about specification already in another programming class. In general, specification is a description of the desired behavior of a piece of software. This usually gets pretty detailed, because this is the only way to be able to discern whether software is correct—if correct software is "software that does what it's supposed to do," then specification is the statement of "what it's supposed to do." We can specify at all levels, too: we can specify the behavior of a word processing application, and we can specify the behavior of a single C++ function. And, we can specify a module.

Specifying a module has two parts: we need to specify what tasks the module will be responsible for (which parts of the application's behavior it provides), *and* we need to specify how the module will interface with the rest of the code (how other modules can communicate with this one). Notice that these specifications will not only help us with correctness, but they also help with modifiability and re-usability—if I need to change a behavior, I can easily tell which module I'll need to work on, and if I want to use some behavior in another application, I can easily tell which module to take.

Encapsulation is related to specification; we've already talked about what encapsulation is: each module has two aspects: Obviously, we need all the code that *implements* the behavior—all the function definitions and class definitions that actually do the work. And then we also need the *interface* that tells other modules what this module offers them—declarations of all the functions and classes in the module that are available to other modules. (And yes, we need to include some documentation about what all these things do and how to use them.) Keeping the interface and implementation separate from each other is the essence of encapsulation. In C and C++, the fundamental way to achieve this encapsulation is by writing a .h and .cpp file for each module: the .h file is the interface, declaring everything that's available to the "public," and the .cpp file is where all the implementation details go.

Finally, orthogonality is the principle that tells us how to group things into modules. Think of orthogonality as a kind of independence: we want our modules to be as independent from each other as possible, and we want the code *inside* a particular module to be strong related to itself. One way of thinking about orthogonal design is to imagine working on a project with a team of people. Often, the most effective way of organizing that team is make each person responsible for a module. The idea is that everyone should be able to work on their implementations independently; everyone needs to know what the *interface* of *every* module is (so that one person can call a function that someone else is working on), but each person can work on their code without knowing exactly what everyone else is doing.

As a thought experiment, here is a very small example. Alice, Bernardo, and Cong are assigned to develop a sophisticated application able to read two integers from input, compute their sum and product, and print the two results to output. (That's the specification they're given… as a side note, is there any information missing from this specification?)

After analyzing the intended behavior of this application, the three decide that they need to implement 5 functions: getint, putint, newline, sum, and product—as well as main. (Yes, C++ makes it very easy to implement these functions! But I promised this would be a very small example.)

Three questions for you to think about:

1. How should Alice, Bernardo, and Cong divide the work so that they can each be as independent as possible?

2. What do they need to agree on before they can start working individually?

3. How can they best use g++ to support their work? Can they run the application when they're working individually? If not, how can they make sure their work is as free from problems as possible?