For an example, we need look no further than the `inventory` program of Chapters 16 and 17. The original program (Section 16.3) stored part records in an array. Suppose that, after using this program for a while, the customer objects to having a fixed limit on the number of parts that can be stored. To satisfy the customer, we might switch to a linked list (as we did in Section 17.5). Making this change required going through the entire program, looking for all places that depend on the way parts are stored. If we'd designed the program differently in the first place—with a separate module dealing with part storage—we would have only needed to rewrite the implementation of that module, not the entire program.

Once we're convinced that modular design is the way to go, the process of designing a program boils down to deciding what modules it should have, what services each module should provide, and how the modules should be interrelated. We'll now look at these issues briefly. For more information about design, consult a software engineering text, such as *Fundamentals of Software Engineering*, Second Edition, by Ghezzi, Jazayeri, and Mandrioli (Upper Saddle River, N.J.: Prentice-Hall, 2003).

Cohesion and Coupling

Good module interfaces aren't random collections of declarations. In a well-designed program, modules should have two properties:

- ***High cohesion.*** The elements of each module should be closely related to one another; we might think of them as cooperating toward a common goal. High cohesion makes modules easier to use and makes the entire program easier to understand.
- ***Low coupling.*** Modules should be as independent of each other as possible. Low coupling makes it easier to modify the program and reuse modules.

Does the calculator program have these properties? The stack module is clearly cohesive: its functions represent operations on a stack. There's little coupling in the program. The `calc.c` file depends on `stack.h` (and `stack.c` depends on `stack.h`, of course), but there are no other apparent dependencies.

Types of Modules

Because of the need for high cohesion and low coupling, modules tend to fall into certain typical categories:

`<float.h>` header ➤*23.1*
`<limits.h>` header ➤*23.2*

- A ***data pool*** is a collection of related variables and/or constants. In C, a module of this type is often just a header file. From a design standpoint, putting variables in header files isn't usually a good idea, but collecting related constants in a header file can be useful. In the C library, `<float.h>` and `<limits.h>` are both data pools.
- A ***library*** is a collection of related functions. The `<string.h>` header, for example, is the interface to a library of string-handling functions.