variable declaration is included in the source file that contains the variable's definition, enabling the compiler to check that the two match.

Although sharing variables among files is a long-standing practice in the C world, it has significant disadvantages. In Section 19.2, we'll see what the problems are and learn how to design programs that don't need shared variables.

Nested Includes

A header file may itself contain #include directives. Although this practice may seem a bit odd, it can be quite useful in practice. Consider the stack.h file, which contains the following prototypes:

```
int is_empty(void);
int is_full(void);
```

Since these functions return only 0 or 1, it's a good idea to declare their return type to be Bool instead of int, where Bool is the type that we defined earlier in this section:

```
Bool is_empty(void);
Bool is_full(void);
```

Of course, we'll need to include the boolean.h file in stack.h so that the definition of Bool is available when stack.h is compiled. (In C99, we'd include <stdbool.h> instead of boolean.h and declare the return types of the two functions to be bool rather than Bool.)

Traditionally, C programmers shun nested includes. (Early versions of C didn't allow them at all.) However, the bias against nested includes has largely faded away, in part because nested includes are common practice in C++.

Protecting Header Files

If a source file includes the same header file twice, compilation errors may result. This problem is common when header files include other header files. For example, suppose that file1.h includes file3.h, file2.h includes file3.h, and prog.c includes both file1.h and file2.h (see the figure at the top of the next page). When prog.c is compiled, file3.h will be compiled twice.

Including the same header file twice doesn't always cause a compilation error. If the file contains only macro definitions, function prototypes, and/or variable declarations, there won't be any difficulty. If the file contains a type definition, however, we'll get a compilation error.

Just to be safe, it's probably a good idea to protect all header files against multiple inclusion; that way, we can add type definitions to a file later without the risk that we might forget to protect the file. In addition, we might save some time during program development by avoiding unnecessary recompilation of the same header file.