## Encapsulation

Unfortunately, `Stack` isn't an *abstract* data type, since `stack.h` reveals what the `Stack` type really is. Nothing prevents clients from using a `Stack` variable as a structure:

```
Stack s1;

s1.top = 0;
s1.contents[top++] = 1;
```

Providing access to the `top` and `contents` members allows clients to corrupt the stack. Worse still, we won't be able to change the way stacks are stored without having to assess the effect of the change on clients.

What we need is a way to prevent clients from knowing how the `Stack` type is represented. C has only limited support for ***encapsulating*** types in this way. Newer C-based languages, including C++, Java, and C#, are better equipped for this purpose.

## Incomplete Types

The only tool that C gives us for encapsulation is the ***incomplete type***. (Incomplete types were mentioned briefly in Section 17.9 and in the Q&A section at the end of Chapter 17.) The C standard describes incomplete types as "types that describe objects but lack information needed to determine their sizes." For example, the declaration

Q&A

```
struct t;   /* incomplete declaration of t */
```

tells the compiler that `t` is a structure tag but doesn't describe the members of the structure. As a result, the compiler doesn't have enough information to determine the size of such a structure. The intent is that an incomplete type will be completed elsewhere in the program.

As long as a type remains incomplete, its uses are limited. Since the compiler doesn't know the size of an incomplete type, it can't be used to declare a variable:

Q&A

```
struct t s;   /*** WRONG ***/
```

However, it's perfectly legal to define a pointer type that references an incomplete type:

```
typedef struct t *T;
```

This type definition states that a variable of type `T` is a pointer to a structure with tag `t`. We can now declare variables of type `T`, pass them as arguments to functions, and perform other operations that are legal for pointers. (The size of a pointer doesn't depend on what it points to, which explains why C allows this behavior.) What we can't do, though, is apply the `->` operator to one of these variables, since the compiler knows nothing about the members of a `t` structure.