doesn't matter whether we use `stack1.c` or `stack2.c` to implement the stack module. Both versions match the module's interface, so we can switch from one to the other without having to make changes elsewhere in the program.

19.3 Abstract Data Types

A module that serves as an abstract object, like the stack module in the previous section, has a serious disadvantage: there's no way to have multiple instances of the object (more than one stack, in this case). To accomplish this, we'll need to go a step further and create a new *type*.

Once we've defined a `Stack` type, we'll be able to have as many stacks as we want. The following fragment illustrates how we could have two stacks in the same program:

```
Stack s1, s2;

make_empty(&s1);
make_empty(&s2);
push(&s1, 1);
push(&s2, 2);
if (!is_empty(&s1))
  printf("%d\n", pop(&s1));    /* prints "1" */
```

We're not really sure what `s1` and `s2` are (structures? pointers?), but it doesn't matter. To clients, `s1` and `s2` are *abstractions* that respond to certain operations (`make_empty`, `is_empty`, `is_full`, `push`, and `pop`).

Let's convert our `stack.h` header so that it provides a `Stack` type, where `Stack` is a structure. Doing so will require adding a `Stack` (or `Stack *`) parameter to each function. The header will now look like this (changes to `stack.h` are in **bold**; unchanged portions of the header aren't shown):

```
#define STACK_SIZE 100

typedef struct {
  int contents[STACK_SIZE];
  int top;
} Stack;

void make_empty(Stack *s);
bool is_empty(const Stack *s);
bool is_full(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);
```

The stack parameters to `make_empty`, `push`, and `pop` need to be pointers, since these functions modify the stack. The parameter to `is_empty` and `is_full` doesn't need to be a pointer, but I've made it one anyway. Passing these functions a `Stack` *pointer* instead of a `Stack` *value* is more efficient, since the latter would result in a structure being copied.