```
void make_empty(Stack s)
{
  s->top = 0;
}

bool is_empty(Stack s)
{
  return s->top == 0;
}

bool is_full(Stack s)
{
  return s->top == s->size;
}

void push(Stack s, Item i)
{
  if (is_full(s))
    terminate("Error in push: stack is full.");
  s->contents[s->top++] = i;
}

Item pop(Stack s)
{
  if (is_empty(s))
    terminate("Error in pop: stack is empty.");
  return s->contents[--s->top];
}
```

The `create` function now calls `malloc` twice: once to allocate a `stack_type` structure and once to allocate the array that will contain the stack items. Either call of `malloc` could fail, causing `terminate` to be called. The `destroy` function must call `free` twice to release all the memory allocated by `create`.

The `stackclient.c` file can again be used to test the stack ADT. The calls of `create` will need to be changed, however, since `create` now requires an argument. For example, we could replace the statements

```
s1 = create();
s2 = create();
```

with the following statements:

```
s1 = create(100);
s2 = create(200);
```

Implementing the Stack ADT Using a Linked List

Implementing the stack ADT using a dynamically allocated array gives us more flexibility than using a fixed-size array. However, the client is still required to specify a maximum size for a stack at the time it's created. If we use a linked-list implementation instead, there won't be any preset limit on the size of a stack.