Lecture 9
Stacks and Queues

15-122: Principles of Imperative Computation (Spring 2023)
Frank Pfenning, André Platzer, Rob Simmons

In this lecture we introduce *queues* and *stacks* as data structures, e.g., for managing tasks. They follow similar principles of organizing the data. Each provides simple functions for adding and removing elements. But they differ in terms of the order in which the elements are removed. They can be implemented easily as an *abstract data type* in C0, like the abstract `ssa_t` type of self-sorting arrays that we discussed in the previous lectures). Today we will not talk about the implementation of stacks and queues; we will implement them in the next lecture.
Relating this to our learning goals, we have

Computational Thinking: We illustrate the power of *abstraction* by considering new data structures from the client side.

Algorithms and Data Structures: Queues and stacks are two important data structures to understand.

Programming: Use and design of interfaces.

1 The Stack Interface

Stacks are data structures that allow us to insert and remove items. They operate like a stack of papers or books on our desk — we add new things to the *top* of the stack to make the stack bigger, and remove items from the top as well to make the stack smaller. This makes stacks a LIFO (Last In First Out) data structure — the data we have put in last is what we will get out first.

Before we consider the implementation of a data structure it is helpful to consider the interface. We then program against the specified interface. Based on the description above, we require the following functions:

```
// typedef ______* stack_t;
```



```
bool stack_empty(stack_t S)     /* O(1), check if stack empty */
  /*@requires S != NULL; @*/;

stack_t stack_new()             /* O(1), create new empty stack */
  /*@ensures \result != NULL; @*/
  /*@ensures stack_empty(\result); @*/;

void push(stack_t S, string x)  /* O(1), add item on top of stack */
  /*@requires S != NULL; @*/
  /*@ensures !stack_empty(S); @*/;

string pop(stack_t S)           /* O(1), remove item from top */
  /*@requires S != NULL; @*/
  /*@requires !stack_empty(S); @*/;
```

Like `ssa_t`, the abstract type `stack_t` is representing a *mutable* data structure, where pushing and popping modifies the contents of the stack. Therefore, we will again be explicit in the interface that stacks are pointers to allocated memory, though we won't be explicit about what they point to.

We want the creation of a new (empty) stack as well as pushing and popping an item all to be constant-time operations, as indicated by $O(1)$. Furthermore, pop is only possible on non-empty stacks. This is a fundamental aspect of the interface to a stack, that a client can only read data from a non-empty stack. So we include this as a `@requires` contract in the interface.

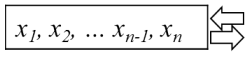
One thing to observe is that there's nothing special about the **`string`** type here. It would be nice to have a data structure that was *generic*, and able to work with strings, integers, arrays, and so on, but we will discuss that possibility later.

2 Using the Stack Interface

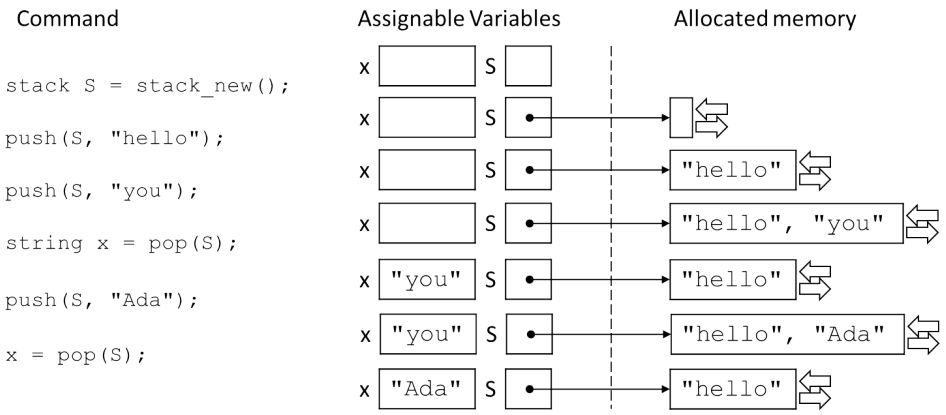
We play through some simple examples to illustrate the idea of a stack and how to use the interface above. We write a stack as

$$x_1, x_2, \ldots, x_n$$

where x_1 is the *bottom* of the stack and x_n is the *top* of the stack. We *push* elements on the top and also *pop* them from the top. If we're feeling artistic, we can draw stacks with arrows to emphasize that we're pushing and popping from the top:



Here is a more complex example, showing the effect of several steps on the state of assignable variables and allocated memory, where the stack data structure resides:



Remember that we think of the assignable `S` like a pointer or an array: it is not literally an arrow but a number representing the address of the in-memory representation of the stack.

3 Abstraction

An important point about formulating a precise interface to a data structure like a stack is to achieve *abstraction*. This means that as a client of the data structure we can only use the functions in the interface. In particular, we are not permitted to use or even know about details of the implementation of stacks.

Let's consider an example of a client-side program. We would like to examine the element at the top of the stack without removing it from the stack. Such a function would have the declaration

```
string peek(stack_t S)
/*@requires S != NULL && !stack_empty(S); @*/ ;
```

If we knew how stacks were implemented, we might be able to implement, as clients of the stack data structure, something like this:

```
1 string peek(stack_t S)
2 //@requires is_stack(S) && !stack_empty(S);
3 {
4   return S->data[S->top];
5 }
```

However, *we don't know how stacks are implemented, so we cannot do this*. As clients of the stack data structure, we only know about the functions provided by the interface. However, it is possible to implement the peek operation correctly *without violating the abstraction*!

The idea is that we pop the top element off the stack, remember it in a temporary variable, and then push it back onto the stack before we return.

```
1 string peek(stack_t S)
2 //@requires S != NULL && !stack_empty(S);
3 {
4   string x = pop(S);
5   push(S, x);
6   return x;
7 }
```

Depending on the implementation of stacks, this might be less efficient than a library-side implementation of peek. However, as long as `push` and `pop` are still a constant-time operations, `peek` will still be constant time ($O(1)$).

4 Computing the Size of a Stack

Let's exercise our data structure once more by considering how to implement a function that returns the size of a stack, using only the interface. In the next lecture we'll consider how to do this on the library's side, exploiting the data representation.

Here's the signature of a client-side implementation.

```
int stack_size(stack_t S)
/*@requires S != NULL;   @*/
/*@ensures \result >= 0; @*/ ;
```

We encourage you to consider this problem and program it before you read on.

First we reassure ourselves that it will not be a simple operation. We do not have access to the array (in fact, as the client, we cannot know that there is an array), so the only thing we can do is pop all the elements off the stack. This can be accomplished with a while-loop that finishes as soon as the stack is empty.

```
1  int stack_size(stack_t S)
2  //@requires S != NULL;
3  //@ensures \result >= 0;
4  {
5    int count = 0;
6    while (!stack_empty(S)) {
7      pop(S);
8      count++;
9    }
10   return count;
11 }
```

However, this function has a *big* problem: in order to compute the size we have to destroy the stack! Clearly, there may be situations where we would like to know the number of elements in a stack without deleting all of its elements. Fortunately, we can use the idea from the `peek` function in amplified form: we maintain a new *temporary stack* T to hold the elements we pop from S. Once we are done counting, we push them back onto S to repair the damage.

```
1  int stack_size(stack_t S)
2  //@requires S != NULL;
3  //@ensures \result >= 0;
4  {
5    stack_t T = stack_new();
6    int count = 0;
7    while (!stack_empty(S)) {
8      push(T, pop(S));
9      count++;
10   }
11   while (!stack_empty(T)) {
12     push(S, pop(T));
13   }
14   return count;
15 }
```

The complexity of this function is clearly $O(n)$, where n is the number of elements in the stack S, since we traverse each while loop n times, and perform a constant number of operations in the body of both loops. For that, we need to know that `push` and `pop` are constant time ($O(1)$).

A library-side implementation of `stack_size` can be done in $O(1)$, but we won't consider that today.

5 The Queue Interface

A *queue* is a data structure where we add elements at the back and remove elements from the front. In that way a queue is like "waiting in line": the first one to be added to the queue will be the first one to be removed from the queue. This is also called a FIFO (First In First Out) data structure. Queues are common in many applications. For example, when we read a book from a file, it would be natural to store the words in a queue so that when we are finished reading the file the words are in the order they appear in the book. Another common example are buffers for network communication that temporarily store packets of data arriving on a network port. Generally speaking, we want to process them in the order they arrive.

Here is our interface:

```
// typedef ______* queue_t;

bool queue_empty(queue_t Q)       /* O(1) */
  /*@requires Q != NULL; @*/;

queue_t queue_new()               /* O(1) */
  /*@ensures \result != NULL; @*/
  /*@ensures queue_empty(\result); @*/;

void enq(queue_t Q, string e)     /* O(1) */
  /*@requires Q != NULL; @*/;

string deq(queue_t Q)             /* O(1) */
  /*@requires Q != NULL; @*/
  /*@requires !queue_empty(Q); @*/ ;
```

Dequeuing is only possible on non-empty queues, which we indicate by a `@requires` contract in the interface.

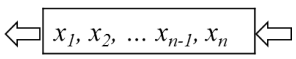
Again, we can write out this interface without committing to an implementation of queues. In particular, the type `queue_t` remains *abstract* in the sense that we have not given its definition. This is important so that different implementations of the functions in this interface can choose different representations. Clients of this data structure should not care about the internals of the implementation. In fact, they should not be allowed to access them at all and operate on queues only through the functions in this interface. Some languages with strong module systems enforce such abstraction rigorously. In C, it is mostly a matter of adhering to conventions.

6 Using the Queue Interface

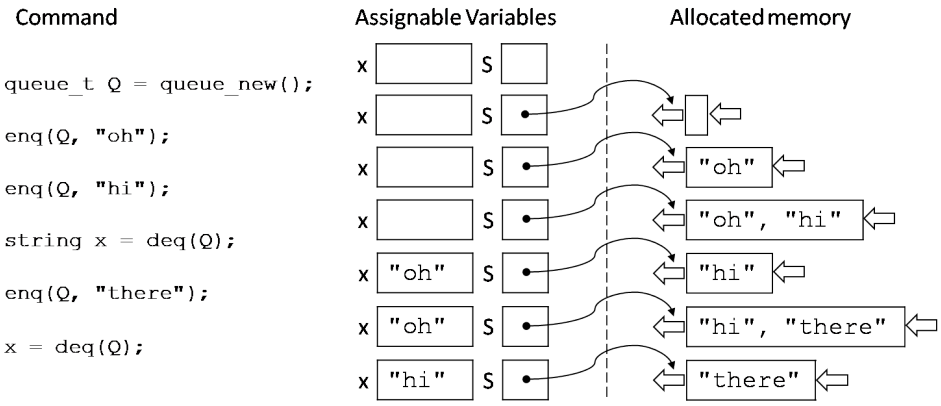
We play through some simple examples to illustrate the idea of a queue and how to use the interface above. We write a queue as

$$x_1, x_2, \ldots, x_n$$

where x_1 is the *front* of the queue and x_n is the *back* of the queue. We *enqueue* elements in the back and *dequeue* them from the front. If we want to emphasize this, we can draw queues like this:



Here's a trace of the queues in action:



7 Copying a Queue Using Its Interface

Suppose we have a queue Q and want to obtain a copy of it. That is, we want to create a new queue C and implement an algorithm that will make sure that Q and C have the same elements and in the same order. How can we do that? Before you read on, see if you can figure it out for yourself.

The first thing to note is that

```
queue_t C = Q;
```

will not have the effect of copying the queue Q into a new queue C. This assignment makes C and Q aliases, so if we change one of the two, for example enqueue an element into C, then the other queue will have changed as well. Just as for the case of stack size, we need to implement a function for copying the data.

The queue interface provides functions that allow us to dequeue data from the queue, which we can do as long as the queue is not empty. So we create a new queue C. Then we read all data from queue Q and put it into the new queue C.

```
1  queue_t C = queue_new();
2  while (!queue_empty(Q)) {
3    enq(C, deq(Q));
4  }
5  //@assert queue_empty(Q);
```

Now the new queue C will contain all data that was previously in Q, so C is a copy of what used to be in Q. But there is a problem with this approach. Before you read on, can you find out which problem?

Queue C now is a copy of what used to be in Q before we started copying. But our copying process was destructive! By dequeuing all elements from Q to put them into C, Q has now become empty. In fact, our assertion at the end of the above loop even indicated `queue_empty(Q)`. So what we need to do is put all data back into Q when we are done copying it all into C. But where do we get it from? We could read it from the copy C to put it back into Q, but, after that, the copy C would be empty, so we are back to where we started from. Can you figure out how to copy all data into C and make sure that it also ends up in Q? Before you read on, try to find a solution for yourself.

We could try to enqueue all data that we have read from Q back into Q before putting it into C.

```
1  queue_t C = queue_new();
2  while (!queue_empty(Q)) {
3    string s = deq(Q);
4    enq(Q, s);
5    enq(C, s);
6  }
7  //@assert queue_empty(Q);
```

But there is something very fundamentally wrong with this idea. Can you figure it out?

The problem with the above attempt is that the loop will never terminate unless Q is empty to begin with. For every element that the loop body dequeues from Q, it enqueues one element back into Q. That way, Q will always have the same number of elements and will never become empty. Therefore, we must go back to our original strategy and first read all elements from Q. But instead of putting them into C, we will put them into a third queue T for temporary storage. Then we will read all elements from the temporary storage T and enqueue them into both the copy C *and* back into the original queue Q. At the end of this process, the temporary queue T will be empty, which is fine, because we will not need it any longer. But both the copy C and the original queue Q will be replenished with all the elements that Q had originally. And C will be a copy of Q.

```
1  queue_t queue_copy(queue_t Q)
2  //@requires Q != NULL;
3  //@ensures \result != NULL;
4  {
5    queue_t T = queue_new();
6    while (!queue_empty(Q)) {
7      enq(T, deq(Q));
8    }
9    //@assert queue_empty(Q);
10   queue_t C = queue_new();
11   while (!queue_empty(T)) {
12     string s = deq(T);
13     enq(Q, s);
14     enq(C, s);
15   }
16   //@assert queue_empty(T);
17   return C;
18 }
```

For example, when `queue_copy` returns, neither C nor Q will be empty. Except if Q was empty to begin with, in which case both C and Q will still be empty in the end.

8 Exercises

Exercise 1 (sample solution on page 15)**.** *The following code is intended to return an exact copy of a stack. It falls short of this goal however. What's wrong with the code?*

```
stack_t stack_copy(stack_t S)
//@requires S != NULL;
{
  stack_t tmp = stack_new();    // temporary stack
  stack_t copy = stack_new();   // stack to be returned

  // move all elements of S into tmp
  while (!stack_empty(S)) {
    string x = pop(S);
    push(copy, x);
    push(tmp, x);
  }

  while (!stack_empty(tmp))
    push(S, pop(tmp));

  return copy;
}
```

Implement a correct version of `stack_copy` *that does return a copy of its input stack.*

Exercise 2 (sample solution on page 15)**.** *Implement the client-side function*

```
stack_t stack_reverse(stack_t S);
```

that destructively *returns a stack with the same elements as its input stack but in reverse order. The input stack shall be empty when the call returns.*

Exercise 3 (sample solution on page 15)**.** *Implement the client-side functions*

```
bool stack_sorted_ascending(stack_t S);
bool stack_sorted_descending(stack_t S);
```

The first returns `true` *if the items in stack* `S` *are sorted in ascending order, with the largest item at the top and the smallest at the bottom, and* `false` *otherwise. Similarly, the second function checks that its argument is sorted in descending order. For simplicity, you may assume that the stack items have type* `int` *rather than* `string`*.*

Your code may use temporary stacks but no other data structures. Besides the functions provided by the stack interface (adapted to items of type `int`*), you may use any function on stacks defined in this lecture or in earlier exercises..*

Exercise 4 (sample solution on page 16)**.** *Implement the function*

```
void stack_sort(stack_t S);
```

that sorts its input stack. The resulting stack should be sorted in ascending order, with the largest item at the top and the smallest at the bottom. For simplicity, you may assume that the stack items have type `int` *rather than* `string`*.*

Your code may use temporary stacks but no other data structures. Besides the functions provided by the stack interface (adapted to items of type `int`*), you may use any function on stacks defined in this lecture or in earlier exercises.*

Hint: an effective way to solve this exercise is carefully consider what loop invariants should hold at various points.

Exercise 5 (sample solution on page 17)**.** *Implement the client-side function*

```
int queue_sum(queue_t Q);
```

that takes as input a queue `Q` *of integers and returns their sum (or 0 if the queue is empty). Upon returning, the input queue should contain the same elements in the same order as when it was called.*

Exercise 6 (sample solution on page 18)**.** *Give a* recursive *implementation of the client-side function*

```
int stack_size(stack_t S);
```

that takes as input a stack `S` *and returns the number of elements in it. Upon returning, the input stack should contain the same elements in the same order as when it was called.*

Sample Solutions

Solution of exercise 1 The returned copy of the input stack is in reverse order. Here's a fixed version:

```
stack_t stack_copy(stack_t S)
//@requires S != NULL;
{
  stack_t tmp = stack_new();    // temporary stack
  stack_t copy = stack_new();   // stack to be returned

  // move all elements of S into tmp
  // -- they will be in the reverse order
  while (!stack_empty(S))
    push(tmp, pop(S));
  //@assert stack_empty(S);

  // move them back to both S and to copy
  while (!stack_empty(tmp)) {
    string x = pop(tmp);
    push(S, x);     // restore x onto S
    push(copy, x);  // put a copy of x onto copy
  }
  //@assert stack_empty(tmp);

  return copy;
}
```

Solution of exercise 2 The code for `stack_reverse` is as follows:

```
stack_t stack_reverse(stack_t S)
//@requires S != NULL;
//@ensures stack_empty(S);
{
  stack_t rev = stack_new();
  while (!stack_empty(S))
    push(rev, pop(S));
  //@assert stack_empty(S);
  return rev;
}
```

Solution of exercise 3 The main challenge in writing these functions is that we want the input stack to remain unchanged no matter what is returned.

An easy way to do so is to borrow the function `stack_copy` from a previous exercise to make a copy of the input stack. Then, we can repeatedly pop and compare elements until we either we empty it out or we find elements out of order.

```
1  bool stack_sorted_ascending(stack_t S)
2  //@requires S != NULL;
3  {
4    if (stack_empty(S))            // the empty stack is sorted
5      return true;
6
7    //@assert !stack_empty(S);
8    stack_t tmp = stack_copy(S);   // make a copy of S
9    //@assert !stack_empty(tmp);
10
11   int n = pop(tmp);
12   while (!stack_empty(tmp)) {
13     int m = pop(tmp);
14     if (!(n > m)) return false;  // tmp is not in ascending order
15     n = m;
16   }
17   //@assert stack_empty(tmp);
18   return true;
19 }
```

`stack_sorted_descending` is identical except for the test `!(n < m)` on line 14.

Solution of exercise 4 The following algorithm relies on a temporary stack `tmp` which will contain the top elements of the input stack `S` sorted in ascending order. For every item `n` in `S`, we compare it to the top of `tmp`: if it is smaller we push it on `tmp`. Otherwise, we repeatedly pop items from `tmp` onto `S` until either we find an item that is larger than or equal to `n` or `tmp` is empty. Note that the same item may be shuffled between `S` and `tmp` multiple times. Once all elements of `S` have been moved to `tmp` (where they will occur in descending order), we simply move them back onto `S`. The stack discipline ensures that they will appear in ascending order.

```
void stack_sort(stack_t S)
//@requires S != NULL;
{
  stack_t tmp = stack_new();              // auxiliary stack
  //@assert stack_sorted_ascending(tmp);  // invariant

  while (!stack_empty(S))
  //@loop_invariant stack_sorted_ascending(tmp);
  {
    int n = pop(S);

    // Move all items in tmp smaller than n back onto S
    while (!stack_empty(tmp) && peek(tmp) < n) {
      push(S, pop(tmp));
    }
    // Push n onto tmp: tmp remains sorted
    push(tmp, n);
  }
  //@assert stack_empty(S);

  // Move them back to S
  while (!stack_empty(tmp))
  //@loop_invariant stack_sorted_ascending(tmp);
  //@loop_invariant stack_sorted_descending(S);
  {
    push(S, pop(tmp));
  }
}
```

We used the functions `stack_sorted_ascending` and `stack_sorted_descending` from the previous exercise to write very precise contracts loop invariants for this function. As written, this code will now compile however. The reason is that these functions write to allocated memory — they are *impure*. In general this is problematic because a program may produce different outcome depending on whether it is compiled with or without the `-d` flag. There is no way to write pure versions of these functions based on the given interface of our stack library. An extended stack library may however export pure versions.

Solution of exercise 5 The following is an implementation of `queue_sum`:

```
int queue_sum(queue_t Q)
//@requires Q != NULL;
{
  int sum = 0;
  queue_t T = queue_new();

  while (!queue_empty(Q)) {
    int x = deq(Q);
    sum += x;
    enq(T, x);
  }

  while (!queue_empty(T))
    enq(Q, deq(T));

  return sum;
}
```

Solution of exercise 6 The following is a recursive implementation of `stack_size`:

```
int stack_size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  if (stack_empty(S))     // Base case
    return 0;

  int data = pop(S);      // Recursive case
  int n = stack_size(S);
  push(S, data);

  return n+1;
}
```

Observe that it is much shorter than the iterative version seen in Section 4. In particular, it does not make use of a temporary stack to hold the data elements of `S` while traversing it. Each call to `stack_size` "remembers" the data item it pops, so when coming back from the recursive call it can be pushed back onto `S`.