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:

```c
/* type elem must be defined by the client */

bool stack_empty(stack S); /* O(1), check if stack empty */
stack stack_new(); /* O(1), create new empty stack */
void push(stack S, elem e); /* O(1), add item on top of stack */
elem pop(stack S) /* O(1), remove item from top */
```

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.

We are being quite abstract here — we do not write, in this file, what type the elements of the stack have to be. Instead we assume that at the top
of the file, or before this file is read, we have already defined a type `elem` for the type of stack elements. We say that the implementation is *generic* or *polymorphic* in the type of the elements. Unfortunately, neither C nor C0 provide a good way to enforce this in the language and we have to rely on programmer discipline.

In the future, we will sometimes indicate that we have a `typedef` waiting to be filled in by the client by writing the following:

```c
typedef _________ elem;
```

This is not actually valid C0, but the client using this library will be able to fill in the underscores with a valid type to make the stack a stack of this type. In this example, we will assume that the client wrote

```c
typedef string elem;
```

The critical point here is that this is a choice that is up to the user of the library (the client), and it is not a choice that the stack library needs to know or care about.

## 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.

For example:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Command</th>
<th>Other variables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>stack ( S = \text{stack_new()} ); push( (S, &quot;a&quot;) ); push( (S, &quot;b&quot;) ); string ( e = \text{pop}(S) ); push( (S, &quot;c&quot;) ); e = ( &quot;c&quot; ); push( (S, &quot;a&quot;) );</td>
<td>( e = &quot;b&quot; )</td>
</tr>
</tbody>
</table>
3 One Stack Implementation (With Arrays)

Any programming language is going to come with certain data structures “built-in.” Arrays, the only really complex data structure we have used so far in this class, are one example in C0. Other data structures, like stacks and queues, need to be constructed using more primitive language features.

We will get to a more proper implementation of stacks in the next lecture, using linked lists. For this lecture we will implement stacks by using the familiar arrays that we have already been using so far in this class.

The idea is to put all data elements in an array and maintain an integer \( \text{top} \), which is the index where we read off elements.

\[
\begin{array}{ccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
\text{\textcolor{blue}{a}} & \text{\textcolor{blue}{b}} & \text{\textcolor{blue}{c}} & \text{\textcolor{red}{x}} & \text{\textcolor{red}{x}} & \text{\textcolor{red}{x}} & \text{\textcolor{red}{x}} & \text{\textcolor{red}{x}} & \text{\textcolor{red}{x}} & \text{\textcolor{red}{x}} & \text{\textcolor{red}{x}} \\
\end{array}
\]

To help identify the similarities with the queue implementation, we decide to also remember an integer \( \text{bottom} \), which is the index of the bottom of the stack. (The bottom will, in fact, remain 0.)

With this design decision, we do not have to handle the bottom of the stack much different than any other element on the stack. The difference is that the data at the bottom of the stack is meaningless and will not be used in our implementation.

There appears to be a very big limitation to this design of stacks: our stack can’t contain more than 9 elements, like a pile of books on our desk that cannot grow too high lest it reach the ceiling or fall over. There are multiple solutions to this problem, but for this lecture we will be content to work with stacks that have a limited maximum capacity.

3.1 Structs and data structure invariants

Currently, our picture of a stack includes three different things: an array containing the data, an integer indicating where the top is, and an integer indicating where the bottom is. This is similar to the situation in Homework 1 where we had data (an array of pixels) and two integers, a width and a height.
C0 has a feature that allows us to bundle these things up into a struct rather than passing around all the pieces separately. We define:

```c
struct stack_header {
    elem[] data;
    int top;
    int bottom;
};
typedef struct stack_header* stack;
```

What this notation means exactly, and especially what the part with `struct stack_header*` is all about, will be explained in the next lecture. (These are pointers and it is crucial to understand them, but we defer this topic for now.) For now, it is sufficient to think of this as providing a notation for bundling aggregate data. When we have a struct `S` of type `stack`, we can refer to the data as `S->data`, the integer representing the top of the stack as `S->top`, and the integer representing the bottom of the stack as `S->bottom`.

When does a struct of this type represent a valid stack? Whenever we define a new data type representation we should first think about the data structure invariants. Making these explicit is important as we think about and write the pre- and postconditions for functions that implement the interface. Here, it is a simple check of making sure that the `bottom` and `top` indices are in the range of the array and that `bottom` stays at 0, where we expect it to be.

```c
bool is_stack(stack S) {
    if (!(S->bottom == 0)) return false;
    if (!(S->bottom <= S->top)) return false;
    //@assert S->top < \length(S->data);
    return true;
}
```

**WARNING:** This specification function is missing something very important (a check for NULL) – we will return to this next time!

When we write specification functions, we use a style of repeatedly saying

```c
if (!(some invariant of the data structure)) return false;
```
so that we can read off the invariants of the data structure. A specification function like is_stack should be safe – it should only ever return true or false or raise an assertion violation – and if possible it should avoid raising an assertion violation. Assertion violations are sometimes unavoidable because we can only check the length of an array inside of the assertion language.

3.2 Checking for emptiness

To check if the stack is empty, we only need to check whether top and bottom are the same number.

```cpp
bool stack_empty(stack S)
//@requires is_stack(S);
{
    return S->top == S->bottom;
}
```

3.3 Popping from a stack

To pop an element from the stack we just look up the data that is stored at the position indicated by the top field of the stack in the array S->data of the data field of the stack. To indicate that this element has now been removed from the stack, we decrement the top field of the stack. We go from

![Diagram of an array-based stack before popping](chart1.png)

... to ...

![Diagram of an array-based stack after popping](chart2.png)
The "c" can still be present in the array at position 3, but it is now a part of the array that we don’t care about, which we indicate by putting an X over it. In code, popping looks like this:

```c
elem pop(stack S)
//@requires is_stack(S);
//@requires !stack_empty(S);
//@ensures is_stack(S);
{
    elem r = S->data[S->top];
    S->top--;
    return r;
}
```

Notice that contracts are cumulative. Since we already indicated
//@requires !stack_empty(S);

in the interface of pop, we would not have to repeat this requires clause in the implementation. We repeat it regardless to emphasize its importance.

### 3.4 Pushing onto a stack

To push an element onto the stack, we increment the top field of the stack to reflect that there are more elements on the stack. And then we put the element e into the array S->data at position top. While this is simple, it is still a good idea to draw a diagram. We go from

![Diagram with "a", "b", "c" in positions 0, 1, 2, and "a", "b", "c", "e" in positions 0, 1, 2, 3.]
In code:

```c
void push(stack S, elem e)
//@requires is_stack(S);
//@ensures is_stack(S);
{
    S->top++;
    S->data[S->top] = e;
}
```

Why is the array access $S->data[S->top]$ safe? Is it even safe? At this point, it is important to note that it is not safe if we ever try to push more elements on the stack than we have reserved space for. We fully address this shortcoming of our stack implementation in the next lecture. What we can do right now to address the issue is to redesign the struct `stack_header` by adding a `capacity` field that remembers the length of the array of the `data` field:

```c
class stack_header {
    elem[] data;
    int top;
    int bottom;
    int capacity; // capacity == \length(data);
};
typedef struct stack_header* stack;
```

Giving us the following updated view of array-based stacks:

![Diagram of stack implementation](image)

The comment that `capacity == \length(data)` is helpful for indicating what the intent of `capacity` is, but it is preferable for us to modify our `is_stack` function to account for the change. (The `WARNING` from before still applies here.)

```c
bool is_stack(stack S)
```
Lectures Notes 2013

Array-Based Stacks

{  
  if (!(S->bottom == 0)) return false;  
  if (!(S->bottom <= S->top)) return false;  
  if (!(S->top < S->capacity)) return false;  
  //@assert S->capacity == \length(S->data);  
  return true;  
}

With a capacity in hand, we check for sufficient space with an explicit assert statement before we try to access the array or change top.

void push(stack S, elem e)  
//@requires is_stack(S);  
//@ensures is_stack(S);  
{  
  assert(S->top < S->capacity - 1); // otherwise no space left  
  S->top++;  
  S->data[S->top] = e;  
}

This assertion can indeed fail if the client tries to push too many elements on the stack, which is why we use a hard assert – an assertion that will run whether or not we compile with -d. The alternative would be to expose the capacity of the stack to the user with a stack_full function and then add a precondition //@requires !stack_full(S) to our push() function.

3.5 Creating a new stack

For creating a new stack, we allocate a struct stack_header and initialize the top and bottom numbers to 0.

stack stack_new()  
//@ensures stack_empty(result);  
//@ensures is_stack(result);  
{  
  stack S = alloc(struct stack_header);  
  S->bottom = 0;  
  S->top = 0;  
  S->capacity = 100; // arbitrary resource bound  
  S->data = alloc_array(elem, S->capacity);  
}
As shown above, we also need to allocate an array data to store the elements in. At this point, at the latest, we realize a downside of our stack implementation. If we want to implement stacks in arrays in the simple way that we just did, the trouble is that we need to decide its capacity ahead of time. That is, we need to decide how many elements at maximum will ever be allowed in the stack at the same time. Here, we arbitrarily choose the capacity 100, but this gives us a rather poor implementation of stacks in case the client needs to store more data. We will see how to solve this issue with a better implementation of stacks in the next lecture.

This completes the implementation of stacks, which are a very simple and pervasive data structure.

4 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

```c
string peek(stack S)
//@requires !stack_empty(S);
{
    return S->data[S->top];
}
```

The first instinct might be to write it as follows:

```c
string peek(stack S)
//@requires !stack_empty(S);
{
    return S->data[S->top];
}
```

However, this would be completely wrong. Let’s recall the interface:

```c
bool stack_empty(stack S);    /* O(1), check if stack empty */
stack stack_new();            /* O(1), create new empty stack */
```
void push(stack S, elem e); /* O(1), add item on top of stack */
elem pop(stack S); /* O(1), remove item from top */
//@requires !stack_empty(S);
;

We don’t see any top field, or any data field, so accessing these as a
client of the data structure would violate the abstraction. Why is this so
wrong? The problem is that if the library implementer decided to improve
the code, or perhaps even just rename some of the structures to make it easi-
ter to read, then the client code will suddenly break! In fact, we will provide
a different implementation of stacks in the next lecture, which would make
the above implementation of peek break. With the above client-side im-
plementation of peek, the stack interface does not serve the purpose it is
intended for, namely provide a reliable way to work with a data structure.
Interfaces are supposed to separate the implementation of a data structure
in a clean way from its use so that we can change one of the two without
affecting the other.

So what can we do? It is possible to implement the peek operation
without violating the abstraction! Consider how before you read on.
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.

string peek(stack S)
//@requires !stack_empty(S);
{
    string x = pop(S);
    push(S, x);
    return x;
}

This is clearly less efficient: instead of just looking up the fields of a struct
and accessing an element of an array we actually have to pop an element
and then push it back onto the stack. However, it is still a constant-time
operation (O(1)) since both pop and push are constant-time operations.
Nonetheless, we have a possible argument to include a function peek in
the interface and implement it library-side instead of client-side to save a
small constant of time.
If we are actually prepared to extend the interface, then we can go back
to our original implementation.

string peek(stack S)
Array-Based Stacks

//@requires !stack_empty(S);
{
  return S->data[S->top];
}

Is this a good implementation? Not quite. First we note that inside the library we should refer to elements as having type elem, not string. For our running example, this is purely a stylistic matter because these two are synonyms. But, just as it is important that clients respect the library interface, it is important that the library respect the client interface. In this case, that means that the users of a stack can, without changing the library, decide to change the definition of elem type in order to store different data in the stack.

Second we note that we are now missing a precondition. In order to even check if the stack is non-empty, we first need to be assured that it is a valid stack. On the client side, all elements of type stack come from the library, and any violation of data structure invariants could only be discovered when we hand it back through the library interface to a function implemented in the library. Therefore, the client can assume that values of type stack are valid and we don’t have explicit pre- or post-conditions for those. Inside the library, however, we are constantly manipulating the data structure in ways that break and then restore the invariants, so we should check if the stack is indeed valid.

From these two considerations we obtain the following code for inside the library:

elem peek(stack S)
//@requires is_stack(S);
//@requires !stack_empty(S);
{
  return S->data[S->top];
}

5 Computing the Size of a Stack

Let’s exercise our data structure once more by developing two implementations of a function that returns the size of a stack: one on the client’s side, using only the interface, and one on the library’s side, exploiting the data representation. Let’s first consider a client-side implementation, using only the interface so far.
int stack_size(stack S);

Again, 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.

```c
int stack_size(stack S) {
    int count = 0;
    while (!stack_empty(S)) {
        pop(S);
        count++;
    }
    return count;
}
```

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.

```c
int stack_size(stack S) {
    stack T = stack_new();
    int count = 0;
    while (!stack_empty(S)) {
        push(T, pop(S));
        count++;
    }
    while (!stack_empty(T)) {
        push(S, pop(T));
    }
    return count;
}
```

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
Array-Based Stacks

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 \( O(1) \).

What about a library-side implementation of stack_size? This can be done more efficiently.

```c
int stack_size(stack S)
//@requires is_stack(S);
{
    return S->top;
}
```