An Introduction to the Stack Data Structure

Jeff Bienstadt

Abstract

This article presents the stack data structure. It describes the layout of the stack and the operations that can be performed with its use. Finally, it presents annotated source code for implementations in the C and C++ languages. The full source code, in both languages, including code to demonstrate the usage of stacks in applications, is available for download from GitHub[1][2].

Part I
Description

The stack is a linear collection implementing a *last in, first out* (**LIFO**) data structure. Elements are added to or removed from a stack one at a time with two primary operations:

- *push*, which adds an item to the stack
- *pop*, which removes an item from the stack

A stack may be either bounded—having a fixed size—or unbounded—able to grow indefinitely.

1 Visualizing a Stack

A stack data structure can be visualized as a linear collection laid out vertically, with the "top" of the stack containing the last added value, if any. A common analogy is a stack of plates at a buffet. New plates are added to the top of the stack, and a plate is taken from the top of the stack for use by each diner. Only the top plate is accessible at any given time. Figure 1 shows the layout of a stack.

[1] https://github.com/jeffbi/data-structures/tree/master/Stack/C
[2] https://github.com/jeffbi/data-structures/tree/master/Stack/C++

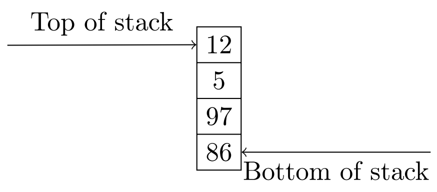

Figure 1: Layout of a stack with four elements.

2 Adding an Item to a Stack

To add an item to a stack, the value is *pushed* onto the top of the stack, making the value's position on the stack the new top. Figure 2 shows a value being pushed onto a stack.

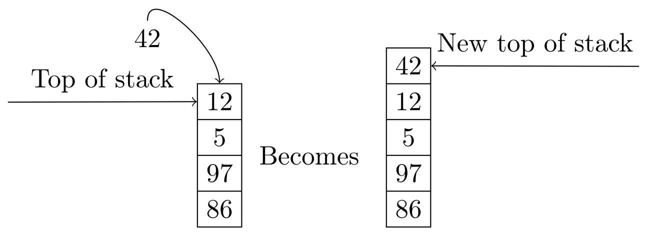

Figure 2: Pushing the value 42 onto a stack.

3 Removing an Item from a Stack

To remove an item from a stack, the item is *popped* from the stack. If there is an item "under" the removed item, that item becomes the new top of the stack. Figure 3 shows a value being popped off of a stack. Some stack implementations

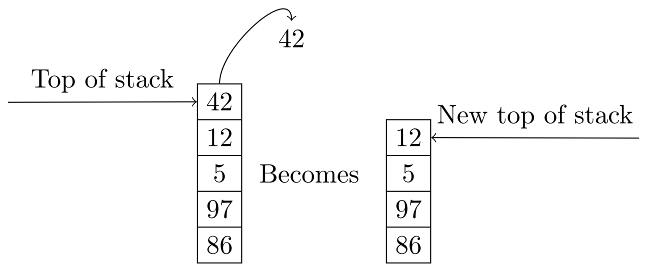

Figure 3: Popping the top value off a stack.

return the popped value, while others simply remove the topmost item and provide a separate operation to access the current top of the stack.

4 Additional Stack Operations.

In general, *push* and *pop* are the only two operations that are required of a stack data structure.

As mentioned above, some implementations of a stack provide a *peek*—or *top*—operation to look at the current top of the stack. These implementations usually—but not always—simply remove the current top-most item when the pop operation is invoked.

An implemtation might also provide a method to report the number of items currently on the stack, and possibly a method to determine whether the stack is empty.

5 Stack Usage

Use a stack when you need a last-in-first-out (LIFO) data structure.

One example of using a stack is in validating the positions of brackets, braces, and parentheses in a string. For example the string

```
[bracket {brace (parentheses)}]
```

is valid because each opening symbol has a matching closing symbol in the correct order. On the other hand,

```
[bracket {brace parentheses)}]
```

is invalid because there is no opening parenthesis to match the closing symbol.

We can write a small program that uses a stack to validate strings such as these. In C++, such a program might look like this:

```
#include <iostream>
#include <stack>  // The C++ library provides a stack type

// return true if the string is valid, false if it is not.
bool validate(const char *string)
{
  std::stack<char> stack;  // A stack of chars
  const char *p = string;

  while (*p)
  {
    // Each opening brace/bracket/parenthesis is stored in the stack
    if (*p == '[' || *p == '{' || *p == '(')
    {
      stack.push(*p);
    }
    else if (*p == ']' || *p == '}' || *p == ')')
18      {
19        if (stack.empty())
20          return false;
21
22        char  match = stack.top();
23
24        if (match == '}' && *p != '{')
25          return false;
26        else if (match == ']' && *p != '[')
27          return false;
28        else if (match == ')' && *p != '(')
29          return false;
30        stack.pop();   // found a match, pop it off the stack.
31      }
32      ++p;
33    }
34
35    return stack.empty(); // There should be nothing left on the stack
36  }
37
38  int main()
39  {
40    const char *strings[]
41      {
42        "{brace [bracket (paren)]}",
43        "{brace [bracket paren)]}",
44        "{brace [bracket (paren)]"
45      };
46
47    for (const char *s : strings)
48    {
49      std::cout << '"' << s << "\" ";
50      if (validate(s))
51        std::cout << "valid\n";
52      else
53        std::cout << "invalid\n";
54    }
55
56    return 0;
57  }
```

In this program we use the **stack** type provided by the C++ standard library with the `#include <stack>` statement on line 2. Later we will implement our own stack type.

The `validate` function examines each character in the string passed to it. If the character is one of the opening bracket/brace/parenthesis characters, that character is pushed onto the stack (lines 13–16). Otherwise, if the character is one of the closing bracket/brace/parenthesis symbols (line 17), we first determine if the stack is empty. If it is empty, there are no opening symbols to compare against and the function returns `false`.

If there are still characters on the stack, we examine the character on the top of the stack. If the opening character on the stack does not match the coresponding closing character found in the string, the function returns `false` (lines 22–29), otherwise it pops the matching character off the stack and examines the next character in the string.

Once each character in the string has been checked, we once again look to see if the stack is empty, on line 35. If it is empty, we have matched all of the symbols in the string and we return `true`. If the stack is not empty, there are still symbols on the stack that were not matched in the string, so we return `false` to indicate that the string is not valid.

Out test program's `main` function simple loops over an array of strings, passing each to the `validate` function and printing whether the string is valid or not. For the array of strings in `main`, the output is

```
"{brace [bracket (paren)]}" valid
"{brace [bracket paren)]}" invalid
"{brace [bracket (paren)]" invalid
```

Part II
Implementation

This part provides sample implementations of stacks, in both the C and C++ languages. The C program implements a bounded, fixed-size stack, using an array as the underlying data type, and is simple and straightforward, while the C++ implementation utilizes an Object Oriented Programming approach to create an unbounded stack utilizing a singly-linked list.

6 C Implementation

Here we present an implementation of a stack in the C language. This implementation uses simple integers for the type of data stored on the stack. The code can be easily modified to work with any data type.

The implementation consists of two C source files, `stack.h` and `stack.c`. It begins in `stack.h` with the definition of the structure and functions that provide the interface for our stack.

```
typedef struct stack
{
  size_t  capacity; // max capacity of our fixed-size stack
  size_t  top;      // current position of the top of the stack
  int    *data;     // contents of the stack
} stack;
```

```

stack *stack_create(size_t capacity);
void stack_delete(stack *stack);
void stack_push(stack *stack, int value);
void stack_pop(stack *stack);
int stack_top(const stack *stack);
size_t stack_size(const stack *stack);
size_t stack_capacity(const stack *stack);
int stack_is_empty(const stack *stack);
int stack_is_full(const stack *stack);
```

The `stack` structure contains information about our stack: in particular it stores the maximum capacity, the position of the top of the stack, and a pointer to the stack data. The `stack_create` function will allocate memory and assign it to the `data` pointer.

The `top` structure member is the index into the `data` array of the current top of the stack. Since `top` is an unsigned value, and zero is a valid array index, we will use zero in `top` to indicate an empty stack, and a value one more than the current index for each stack position. This will be an internal detail and invisible to the user.

This completes `stack.h`. The function implementations are in `stack.c`.

The first thing to do is to `#include` two standard C library header files. The file `assert.h` defines the `assert` macro which we will use for bounds checking. The file `stdlib.h` file declares the `malloc` and `free` functions which we will use to allocate and free memory for our stack.

Following the C library files, we `#include` our `stack.h` file that we looked at earlier.

Next is the implementation code for the functions that operate on the stack, starting with `stack_create`.

```
 1  stack *stack_create(size_t capacity)
 2  {
 3    stack *new_stack = (stack *)malloc(sizeof(stack));
 4    if (new_stack == NULL)
 5      return NULL;
 6
 7    new_stack->capacity = capacity;
 8    new_stack->top = (size_t)0;
 9    new_stack->data = (int *)malloc(capacity * sizeof(int));
10    if (new_stack->data == NULL)
11    {
12      free(new_stack);
13      return NULL;
14    }
15
16    return new_stack;
17  }
```

The `capacity` parameter on line 1 specifies the maximum number of elements we can store in our stack. On line 3 we use the `malloc` function to allocate memory for our stack structure. If the memory allocation fails, we return NULL to indicate an error.

Next, on line 7 we store the capacity in out stack structure, then on line 8 we initialize the top-of-stack index to 0, indicating that the stack is empty.

On line 9 we attempt to allocate memory for capacity integers, and store the resultant pointer in the data structure member. If the allocation fails, which we test for on line 10, then we make sure to free the memory we allocated to the structure (line 12) and again return NULL to indicate an error in creating the stack. If our allocation succeeds, on line 16 we return a pointer to our stack.

When we are done using our stack, we will want to free any memory allocated to it. The stack_delete function on line 18 does this by first freeing the data array, then freeing the memory for the stack structure itself.

```
18 void stack_delete(stack *stack)
19 {
20   free(stack->data);
21   free(stack);
22 }
```

The stack_push function pushes a new value onto the stack

```
23 void stack_push(stack *stack, int value)
24 {
25   assert(!stack_is_full(stack));
26 
27   stack->data[stack->top++] = value;
28 }
```

Line 25 assures the stack is not already at full capacity. In line 27 we store the pushed value at the current top of the stack and increment the top index value. Note that top is incremented *after* the value is added to the array: this allows us to use a top value of zero to indicate the stack is empty.

The stack_pop function removes the top-most item from the stack.

```
29 void stack_pop(stack *stack)
30 {
31   assert(!stack_is_empty(stack));
32 
33   --stack->top;
34 }
```

Line 31 uses assert to ensure the stack is not empty. Then line 33 simply decrements the top-of-stack index.

Next, the stack_top function returns the value at the top of the stack.

```
35 int stack_top(const stack *stack)
36 {
37   assert(!stack_is_empty(stack));
38 
39   return stack->data[stack->top-1];
40 }
```

Again, assert is used to ensure the stack is not empty. Line 39 returns the value at the top of the stack. Remember, the top structure member always indexes the location one position beyond the top-most element of the array so we use top-1 to index the correct array element.

The stack_size function returns the number of items currently on the stack.

```
41  size_t stack_size(const stack *stack)
42  {
43    return stack->top;
44  }
```

This function is very simple. It simply returns the top member of the stack structure. Since top always contains the array index one beyond the top-most item on the stack, it doubles as a counter.

The stack_capacity function is equally simple.

```
45  size_t stack_capacity(const stack *stack)
46  {
47    return stack->capacity;
48  }
```

It returns the capacity member of the stack structure. This is the same value passed as the stack capacity to the stack_create function.

Finally, we define the stack_is_empty and stack_is_full functions.

```
49  int stack_is_empty(const stack *stack)
50  {
51    return stack->top == 0;
52  }
53  int stack_is_full(const stack *stack)
54  {
55    return stack->top == stack->capacity;
56  }
```

We used these functions earlier. Line 51 in function stack_is_empty returns a non-zero value if the top structure member is equal to zero, indicating the stack is empty. Line 55 in the function stack_is_full function returns a non-zero value if the top structure member is equal to the stack capacity, indicating the stack is full.

This completes the implementation of a fixed-size stack in C, produced in just 72 lines of code.

7 C++ Implementation

This section presents an implementation of an unbounded stack in the C++ language. The implementation utilizes Object Oriented Programming principles, particularly encapsulation. Templates are also used so a stack can use any copyable data type.

Note that this implementation in purely expositive. The C++ standard library provides an excellent stack implementation that is much more thoroughly optimized and tested, and should be preferred in production code.

Unlike the C version, which employed two source files, a `.h` header and a `.c` source file, the C++ version will be implemented as header-only, using only a single C++ `.h` header file.

Like the C version, this implementation uses the `assert` macro to handle errors. The definition of this macro is in the `<cassert>` standard header file.

```
1  #include <cassert>
```

Next we begin the definition of the `Stack` class. The entire implementation will be in this class.

```
2  template <typename T>
3  class Stack
4  {
```

Class `Stack` is a class template, so we will be able to use any copyable data type for the data stored in our stack.

We will be implementing this stack using a singly-linked list as the underlying data type. A linked list is essentially a sequence of nodes, connected by pointers. For more information about linked lists, see this article: https://github.com/jeffbi/data-structures/blob/master/LinkedList/doc/linkedlist.pdf

We first define our linked list node type:

```
5  private:
6    struct node
7    {
8      explicit node(const T& value)
9        : _data{value},
10         _next{nullptr}
11     {}
12
13     T     _data;
14     node *_next;
15   };
```

There are two data members of the `node` structure: `_data`, defined on line 13, is the actual data stored on the stack; `_next`, defined on line 14, is a pointer to the next node in the linked list. The `node` constructor, defined on lines 8–11, initializes the node with a data value and sets the `_next` pointer to `nullptr`.

The `Stack` has two data members:

```
16   size_t  _size;
17   node   *_head;
```

The `_size` member stores the number of items currently on the stack, while the `_head` member is a pointer to a `node` and is the head node of our linked list. If the `_head` pointer is the `nullptr`, then the stack is empty. Otherwise this pointer points to the head of the linked list, which is also the top element of the stack.

We now begin the public interface of the Stack class. First, we provide a default constructor that initializes an empty stack, setting the _size member to zero and the _head member to nullptr:

```
18    Stack()
19     : _size{0},
20       _head{nullptr}
21    {
22    }
```

The destructor is the most complicated function in the Stack class. It iterates over the linked list, head to tail (top of stack to bottom), and frees the memory allocated to each node:

```
23    ~Stack()
24    {
25      while (_head)
26      {
27        node *new_head = _head->_next;
28        delete _head;
29        _head = new_head;
30      }
31    }
```

On line 25, so long as the _head data member is not the nullptr, the destructor stores a temporary copy of the _head's _next pointer (line 27), deletes the memory allocated to the _head linked list node (line 28) then re-assigns the _head node to the copy of the previous _head's _next pointer. This frees all the memory allocated to storing the stack data.

The push member function is how we get a new item onto the top of the stack.

```
32    void push(const T &value)
33    {
34      node *new_head = new node(value);
35
36      new_head->_next = _head;
37      _head = new_head;
38
39      ++_size;
40    }
```

The value parameter is the item we want to add to the top of the stack. On line 34 we create a new node with value. Line 36 inserts the newly-created node as the new head of the list by setting the new head's _next pointer to point to the current head (which will be the nullptr if the stack is currently empty). Line 37 re-assigns the current _head pointer to the new head. Finally, on line 39 we simply add one to the _size data member. Note that we did not check to see if the stack is full. This stack implementation is unbounded so we can push as many item as we can create nodes for.

The pop member function removes the top-most item from the stack.

```
41    void pop()
42    {
43      assert(!is_empty());
44      node *old_head = _head;
45
46      _head = old_head->_next;
47      delete old_head;
48
49      --_size;
50    }
```

Here, we do care if the stack is empty. We cannot pop a value if there are no values on the stack, so the first thing we do is use `assert` on line 43 to ensure that the stack is not empty. We then store a temporary copy of the stack's `_head` pointer on line 44, reset the head to the old `_head`'s `_next` pointer on line 46, then delete the old head node (line 47). Finally we decrement the `_size` data member to indicate that there is one less item on the stack.

The `top` function is used to see the value currently on the top of the stack.

```
51    const T &top() const
52    {
53      assert(!is_empty());
54      return _head->_data;
55    }
```

Like the `pop` function, we need to check if the stack is empty—we cannot look at a value that does not exist—so we again use `assert` to ensure that the stack is not empty. We then, on line 54 simply return the value in the `_data` member of the `_head` node.

There are actually two versions of the `top` function. The one we saw above returns a |verb|const| reference, which cannot be modified. We also have a version that returns a non-`const` reference, one that can be modified. Except for the function signatures, the two work exactly alike:

```
56    T &top()
57    {
58      assert(!is_empty());
59      return _head->_data;
60    }
```

Next is the `is_empty` function which simply determines if the stack is empty.

```
61    bool is_empty() const noexcept
62    {
63      return _head == nullptr;
64    }
```

For our purposes, the stack is empty if the `_head` data member is equal to the `nullptr`. Line 63 returns `true` is that is the case, and `false` otherwise.

The final function in our stack implementation, `size()`, returns the number of items on the stack, by returning the currently value of the `_size` data member.

```cpp
  size_t size() const noexcept
  {
    return _size;
  }
```

This completes the implementation of an unbounded stack in C++, in under 70 lines of code.