

# Stacks: Implementation in C



Computer Science Department  
University of Central Florida

*COP 3502 – Computer Science I*

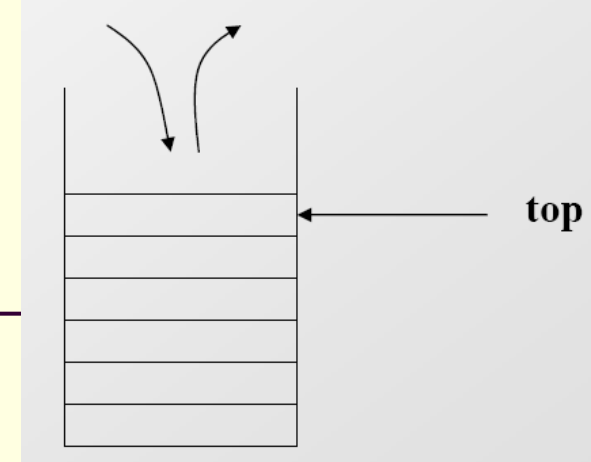


# Stacks – An Overview

---

- Stacks:
  - Stacks are an Abstract Data Type
    - They are NOT built into C
  - We must define them and their behaviors
  - So what is a stack?
    - A data structure that stores information in the form of a stack.
    - Consists of a variable number of homogeneous elements
      - i.e. elements of the same type

# Stacks – An Overview



## ■ Stacks:

### ■ Access Policy:

- The access policy for a stack is simple: the first element to be removed from the stack is the last element that was placed onto the stack
  - The main idea is that the last item placed on to the stack is the first item removed from the stack
- Known as the “Last in, First out” access policy
  - LIFO for short
- The classical example of a stack is cafeteria trays.
  - New, clean trays are added to the top of the stack.
  - and trays are also taken from the top
  - So the last tray in is the first tray taken out



# Stacks – An Overview

---

- Stacks:
  - Basic Operations:
    - PUSH:
      - This PUSHes an item on top of the stack
    - POP:
      - This POPs off the top item in the stack and returns it
  - Other important tidbit:
    - The end of the stack,
      - where PUSHes and POPs occur,
    - is usually referred to as the TOP of the stack



# Stacks – An Overview

---

- Stacks:
  - Basic Operations:
    - PUSH:
      - This PUSHes an item on top of the stack
    - POP:
      - This POPs off the top item in the stack and returns it
  - Other important tidbit:
    - The end of the stack,
      - where PUSHes and POPs occur,
    - is usually referred to as the TOP of the stack



# Stacks – An Overview

---

- Stacks:
  - Other useful operations:
    - empty:
      - Typically implemented as a boolean function
      - Returns TRUE if no items are in the stack
    - full:
      - Returns TRUE if no more items can be added to the stack
      - In theory, a stack should NEVER become full
      - Actual implementations do have limits on the number of elements a stack can store
    - top:
      - Simply returns the value at the top of the stack without actually popping the stack.



# Stacks: Implementation in C

---

- Implementation of Stacks in C:
  - As discussed on the previous lecture, there are two obvious ways to implement stacks:
    - 1) Using arrays
    - 2) Using linked lists
  - We will go over both...



# Stacks: Implementation in C

---

- Array Implementation of Stacks:
  - What components will we need to store?
    - 1) The array storing the elements
      - The actual stack
    - What else?
      - 2) An index to the top of the stack
        - We assume the bottom of the stack is index 0
          - Meaning, the 1<sup>st</sup> element will be stored in index 0
        - and we move up from there





# Stacks: Implementation in C

---

- Array Implementation of Stacks:
  - Here is the struct (skeleton) for our stack:

```
struct stack {  
    int items[SIZE];  
    int top;  
};
```

- SIZE clearly represents the max number of items in the stack
- If the stack becomes full, at that point, the top item will be stored at index 'SIZE-1'



# Stacks: Implementation in C

---

## ■ Array Implementation of Stacks:

- Here are the functions we will need to control our stack behavior:
- `void initialize(struct stack* stackPtr);`
- `int empty(struct stack* stackPtr);`
- `int full(struct stack* stackPtr);`
- `int push(struct stack* stackPtr, int value);`
- `int pop(struct stack* stackPtr);`
- `int top(struct stack* stackPtr);`



# Stacks: Implementation in C

---

- Array Implementation of Stacks:
  - `initialize`:
    - The `initialize` function has one line of code
    - It sets the “top” equal to -1
    - Remember, the first element will be at index 0
    - So if the top is set to -1
    - You know that the stack is empty
  - Here’s the code:

```
void initialize(struct stack* stackPtr) {  
    stackPtr->top = -1;  
}
```



# Stacks: Implementation in C

---

- Array Implementation of Stacks:
  - `empty`:
    - The `empty` function simply checks if the stack has no elements
    - Based on what you know thus far, how would you determine if the stack is empty?
      - If the top currently equals -1
  - Here's the code:

```
int empty(struct stack* stackPtr) {  
    return (stackPtr->top == -1);  
}
```



# Stacks: Implementation in C

---

## ■ Array Implementation of Stacks:

### ■ `full`:

- The `full` function checks to see if the stack is full
- How would we do this?
- Remember, `SIZE` is the max # of elements in the stack
  - Item 1 goes at index 0
  - If the stack is full, the top item will be at index '`SIZE-1`'

### ■ Here's the code:

```
int full(struct stack* stackPtr) {  
    return (stackPtr->top == SIZE - 1);  
}
```



# Stacks: Implementation in C

---

## ■ Array Implementation of Stacks:

### ■ push:

- Remember, we can only push if the stack is not full
  - Meaning, if there is room to push
- So if the stack is full
  - We return 0 showing the push could not be done
- If there is room
  - we simply copy the value into the next location for the top of the stack
  - Then we adjust the top accordingly
  - Finally, we return 1 showing the push was successful



# Stacks: Implementation in C

## ■ Array Implementation of Stacks:

### ■ push:

- To `push` an element, we simply copy the value into the next location for the top of the stack
- Then we adjust the top accordingly

### ■ Here's the code:

```
int push(struct stack* stackPtr, int value) {
    if (full(stackPtr))
        return 0;
    stackPtr->items[stackPtr->top+1] = value;
    (stackPtr->top)++;
    return 1;
}
```



# Stacks: Implementation in C

---

- Array Implementation of Stacks:
  - pop:
    - Remember, we can only pop if the stack is not empty
      - Meaning, there is at least one element to pop
    - So if the stack is empty
      - We return -1 showing that we cannot pop (stack empty)
    - If the stack has at least one element:
      - We save the value at the top of the stack into a temporary variable
      - We change the value for top
        - Meaning if top was 20 before the pop, it will now be 19
        - Meaning it will now reference index 19
      - Finally, we return the temporary variable (the popped off top)





# Stacks: Implementation in C

---

## ■ Array Implementation of Stacks:

### ■ pop:

- To `pop` an element, we simply copy the top into a temporary variable, adjust the top accordingly, and return the temporary variable.

### ■ Here's the code:

```
int pop(struct stack* stackPtr) {
    int retval;
    if (empty(stackPtr))
        return -1;
    retval = stackPtr->items[stackPtr->top];
    (stackPtr->top)--;
    return retval;
}
```



# Stacks: Implementation in C

---

- Array Implementation of Stacks:
  - `top`:
    - The `top` function is very similar to `pop`
    - Remember, we can only check for the top of the stack if the stack is not empty
      - Meaning, there is at least one element in the stack
    - So if the stack is empty
      - We return -1 showing that there is no top to check for
    - If the stack has at least one element:
      - We simply return the topmost element



# Stacks: Implementation in C

---

- Array Implementation of Stacks:
  - top:
    - Simply returns the top item in the stack
  - Here's the code:

```
int top(struct stack* stackPtr) {  
    if (empty(stackPtr))  
        return -1;  
    return stackPtr->items[stackPtr->top];  
}
```



# Stacks: Implementation in C

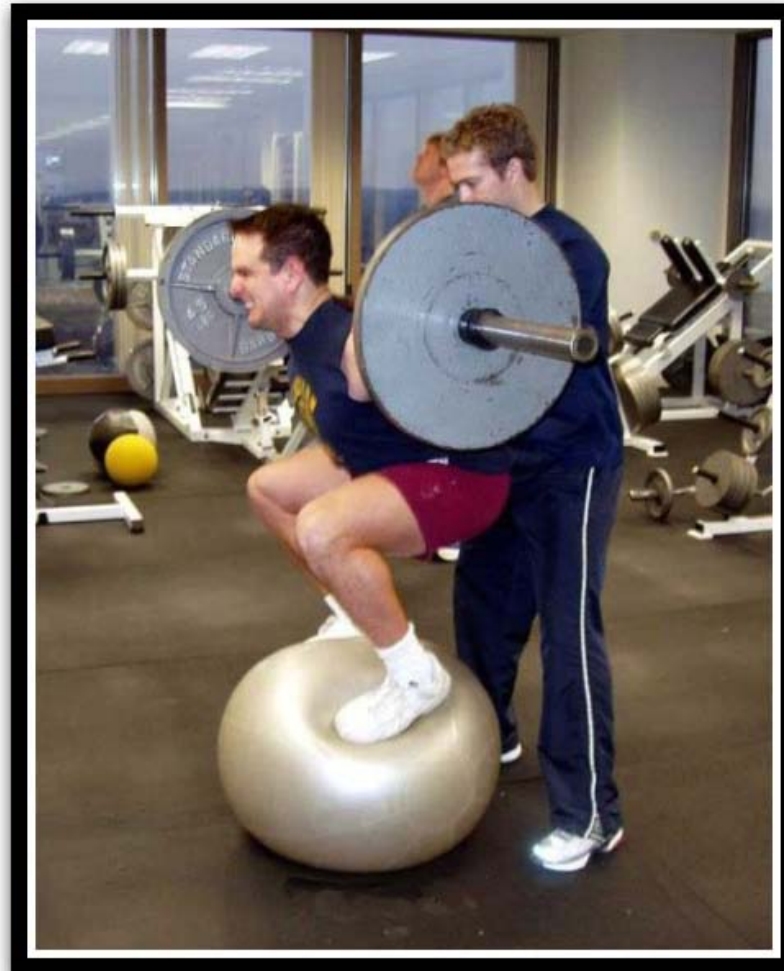
---

- Array Implementation of Stacks:
  - Here the link to this code on the site:
  - <http://www.cs.ucf.edu/courses/cop3502/spr2011/programs/stacksqueues/stack.c>



# Brief Interlude: Human Stupidity

---





# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks:
  - We essentially use a standard linked list
  - But we limit the functionality of a linked list
    - Thus creating the behavior required of a stack
  - A `push` is simply designated as **inserting into the front** of the linked list
  - A `pop` would be **deleting the front node**
- So we basically create just one struct for the stack
  - It acts similar to the struct defined for use with linked lists



# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks:
  - So each node will be an element of the stack
  - Each node has a data value
  - Each node also has a next
  - We simply `push` (insert at front)
  - And `pop` (delete the front node)



# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks:
  - Here's the struct for the stack (for each node)

```
struct stack {  
    int data;  
    struct stack *next;  
};
```

- Notice that we do not have a 'top'
- Why?
  - The top will ALWAYS be the first node
  - And we don't need to worry about the size getting too large since this is a linked list (in heap memory)





# Stacks: Implementation in C

---

## ■ Linked Lists Implementation of Stacks:

- Here are the functions we will need to control our stack behavior:

- `void init(struct stack **front);`

- `int empty(struct stack *front);`

- `int push(struct stack **front, int num);`

- `struct stack* pop(struct stack **front);`

- `int top(struct stack *front);`



# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks:
  - `initialize`:
    - The `initialize` function has one line of code
    - It simply sets the pointer of the list to `NULL`
      - Specifying that the list is empty at this point
  - Here's the code:

```
void init(struct stack **front) {  
    *front = NULL;  
}
```



# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks:
  - empty:
    - The lists is empty when the main list pointer is NULL
    - So if `front` equals NULL
      - Return 1 showing the list is empty
    - Else, return 0 showing that the list is not empty
  - Here's the code:

```
int empty(struct stack *front) {
    if (front == NULL)
        return 1;
    else
        return 0;
}
```



# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks :
  - `push`:
    - Remember, `push` means that we add a new node at the front of the list
    - So we need to allocate this node
    - We need to save the data value into this node
    - We then need to update pointers accordingly
      - The new node will now be the FIRST node
      - So the address of the current front node needs to be saved into the `next` of this new node
        - Allowing the new node to point to the previous first node
        - The pointer to the front of the list needs to get updated
    - Finally, we return 1 to show a successful `push`



# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks :
  - push:
  - Here's the code:

```
int push(struct stack **front, int num) {
    struct stack *temp;
    temp = (struct stack *)malloc(sizeof(struct stack));
    if (temp != NULL) {
        temp->data = num;
        temp->next = *front;
        *front = temp;
        return 1;
    }
    else
        return 0;
}
```



# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks :
  - pop:
    - Assuming that there is at least one node to pop
    - We make a temp pointer to point to the front node
      - The node we will pop
    - We then update our pointers accordingly
      - The 2<sup>nd</sup> node now becomes the first node
    - Finally, we return the address of the temp pointer



# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks :
  - pop:
  - Here's the code:

```
struct stack* pop(struct stack **front) {
    struct stack *temp;
    temp = NULL;

    if (*front != NULL) {
        temp = (*front);
        *front = (*front)->next;
        temp -> next = NULL;
    }
    return temp;
}
```



# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks :
  - top:
    - Assuming that there is at least one node
    - We simply return the data value of that node
  
    - Otherwise,
      - If there is no nodes
    - We return -1 showing that the list is empty





# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks :
  - top:
  - Here's the code:

```
int top(struct stack *front) {  
    if (front != NULL) {  
        return front->data;  
    }  
    else  
        return -1;  
}
```



# Stacks: Implementation in C

---

- Linked Lists Implementation of Stacks:
  - Here the link to this code on the site:
  - <http://www.cs.ucf.edu/courses/cop3502/spr2011/programs/stacksqueues/stackll.c>



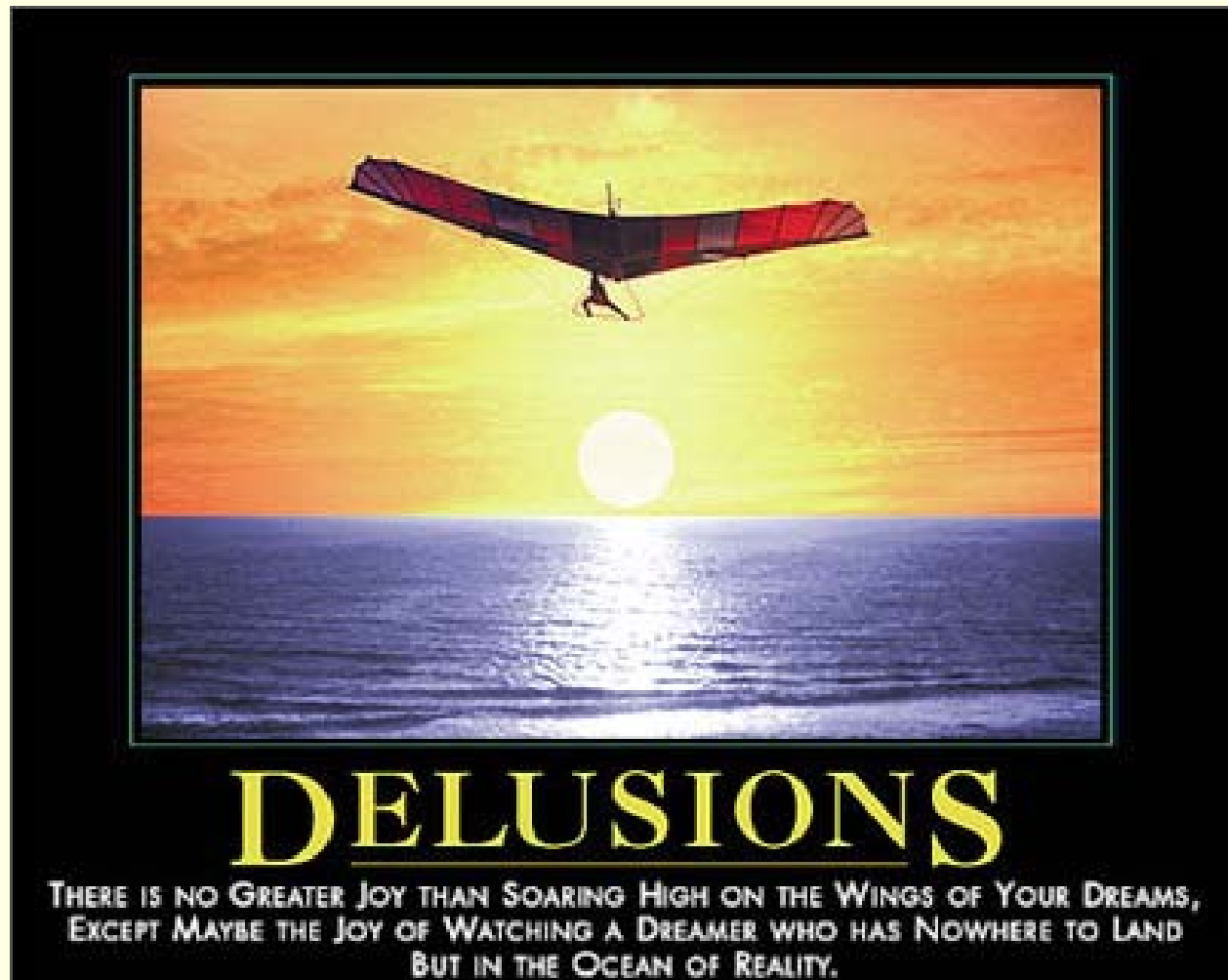
# Stack Application(s)

---

**WASN'T  
THAT  
SPLENDID!**



# Daily Demotivator



# Stacks: Implementation in C



Computer Science Department  
University of Central Florida

*COP 3502 – Computer Science I*