Stacks

CS 250 – C++ Programming 2

ABSTRACT DATA TYPES

- Often the solution to a problem requires operations on **data**:
  - **Add** data to a data collection
  - **Remove** data from a data collection
  - **Ask questions** about the data in a data collection

- Details of the operations may vary, but the idea is to **manage data**.

Abstract Data Types

- An **abstract data type** (**ADT**) is
  - A **collection of data** AND
  - A **set of operations** on the data

- You can use an ADT's operation without knowing how it is implemented or how the data is stored.

Data Structures

- If you implement an ADT, you will need to choose a specific **data structure**.

- **Data structure**
  - A specific way to store and organize data in a computer so that it can be used efficiently
  - An **array** is a data structure
  - A **linked list** is a data structure

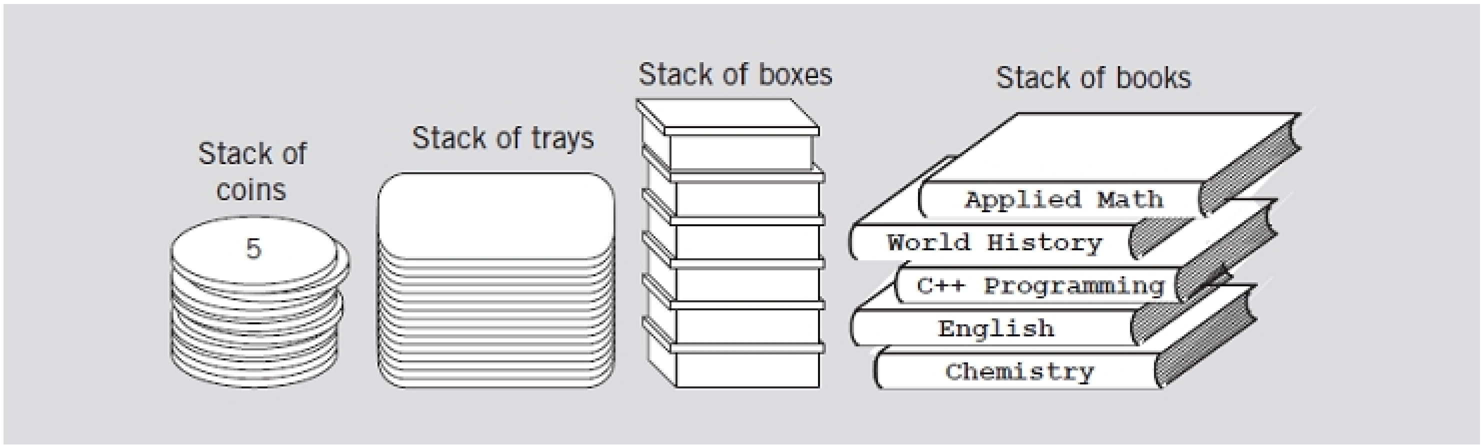
Developing an ADT

- Assume you are creating a **Web browser** and you are at the point where you need to implement the **BACK button**.
  - The user goes to the **OCC Web site**
  - Clicks on **Academics**
  - Clicks on **Academic Divisions**
    → **Business & Computing**
  - Clicks on **Computer Science**

  - How can your button **go back** to the OCC Web site?

Stacks

- The **ADT Stack** (a **data structure**)
  - Elements are **added** and **removed** from **one end only**: the **top** of the **stack**
  - Last In First Out (**LIFO**)



Various examples of stacks

Which Operations Are Needed?

- There are only a few operations needed for the **ADT stack**:
  - Test whether a stack is empty
  - Add a new item to the stack
  - Remove from the stack the item that was added most recently
  - Get the item that was added to the stack most recently

STL Stack

- The **Standard Template Library** (STL) provides a **class** to implement a **stack**.
  - It is a **template** class

```
#include <stack>

...

stack<int> intStack;            // creates a stack of integers

stack<string> stringStack;      // creates a stack of strings
                                // need to include <string>
```

STACK OPERATIONS

Operation	What it does
push(obj)	**Inserts** a new **element** at the **top** of the stack.

Stack Operations (cont.)

Operation	What it does
push(obj)	**Inserts** a new **element** at the **top** of the stack.
pop( )	**Removes** the **element** at the **top** of the stack.

STACK OPERATIONS (CONT.)

Operation	What it does
push(obj)	**Inserts** a new **element** at the **top** of the stack.
pop( )	**Removes** the **element** at the **top** of the stack.
empty( )	Returns **true** if the stack is **empty**, and returns **false** otherwise.

Stack Operations (cont.)

Operation	What it does
push(obj)	**Inserts** a new **element** at the **top** of the stack.
pop( )	**Removes** the **element** at the **top** of the stack.
empty( )	Returns **true** if the stack is **empty**, and returns **false** otherwise.
top( )	**Retrieves** (**without removing**) the element at the **top** of the stack.

Stack Operations

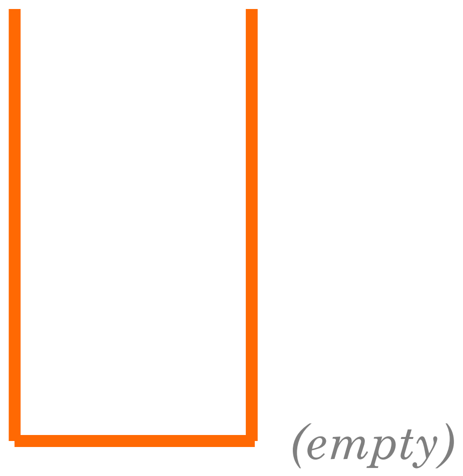
Operation	What it does
push(obj)	**Inserts** a new **element** at the **top** of the stack.
pop( )	**Removes** the **element** at the **top** of the stack.
empty( )	Returns **true** if the stack is **empty**, and returns **false** otherwise.
top( )	**Retrieves** (**without removing**) the element at the **top** of the stack.
size( )	Returns the **number of elements** in the stack.

Tracing Code

We will create a **stack** of **integers**, **myStack**

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

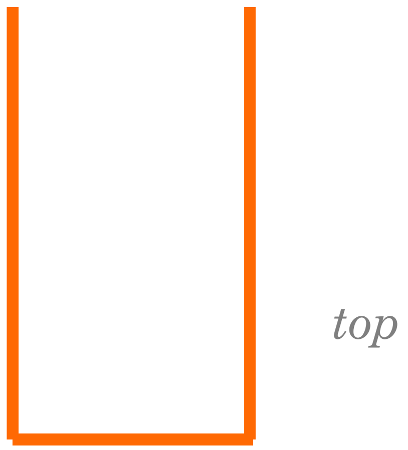
Tracing Code



This is our **stack** of **integers** (now empty).

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

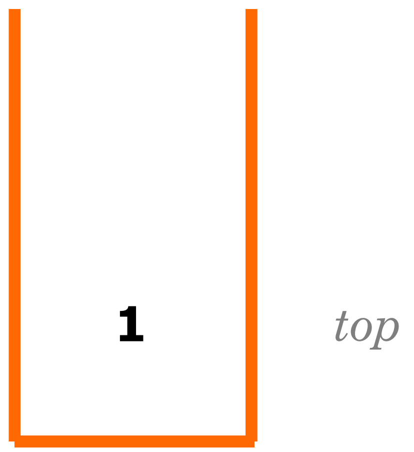
Tracing Code



We **push** integer **1** into the **stack**.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

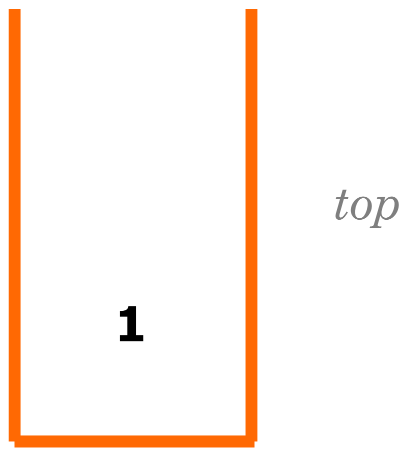
Tracing Code



We **push** integer 1 into the **stack**.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

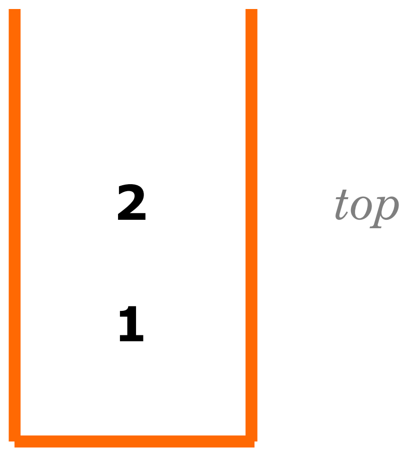
Tracing Code



We **push** integer **2** into the **stack**.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

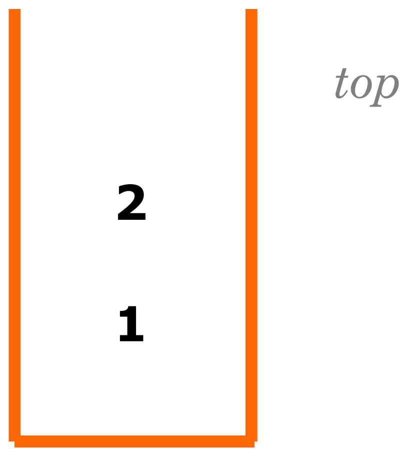
Tracing Code



We **push** integer 2 into the **stack**.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

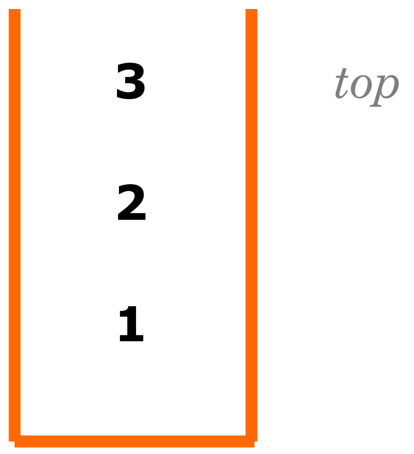
Tracing Code



We **push** integer **3** into the **stack**.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

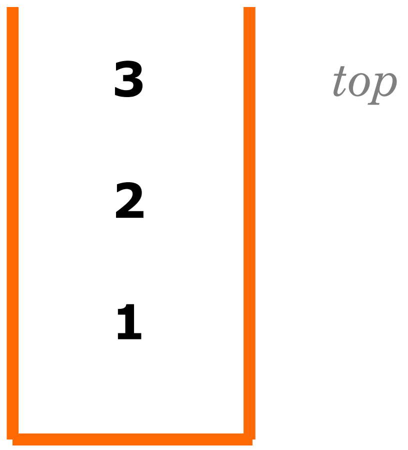
Tracing Code



We **push** integer **3** into the **stack**.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

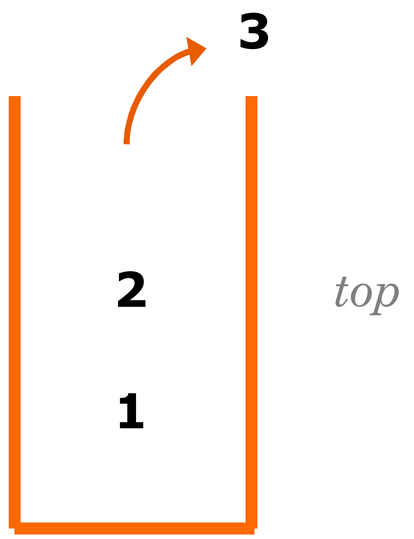
Tracing Code



The **IF** statement is **TRUE** when the **stack** is **not** empty.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

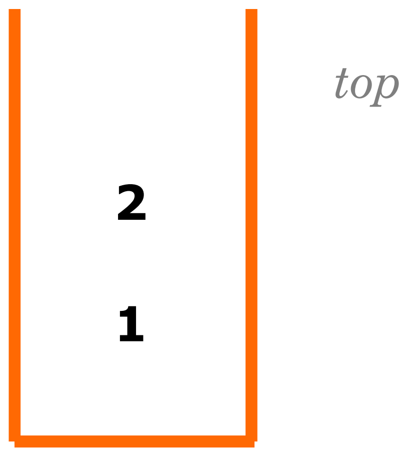
Tracing Code



We **pop** the **top** element from the **stack** (**no** return value when popping).

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

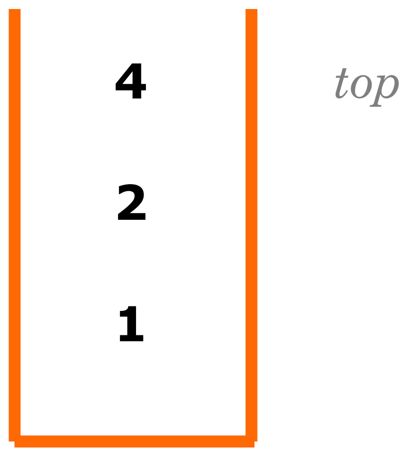
TRACING CODE



We **push** integer 4 into the **stack**.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

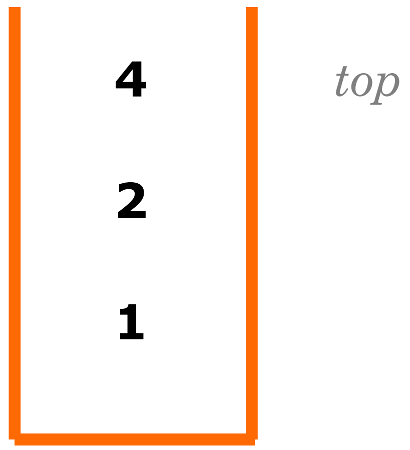
Tracing Code



We **push** integer 4 into the **stack**.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

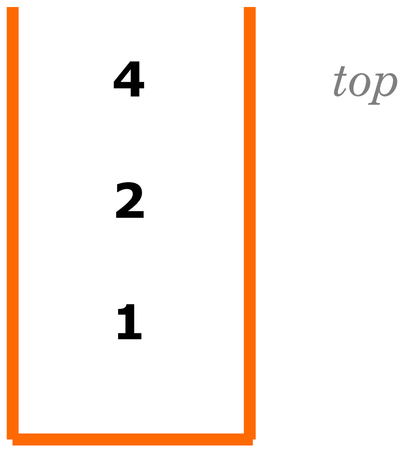
Tracing Code



WHILE statement will execute as long as the **stack** is **not** empty.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

TRACING CODE



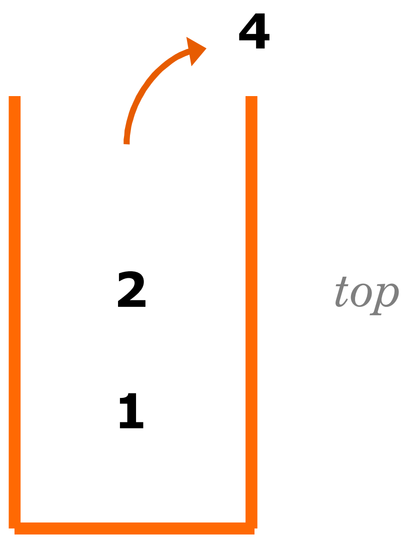
Retrieve (*without* removing) the **element** at the **top** of the **stack** and print it.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

Output:

```
4
```

Tracing Code



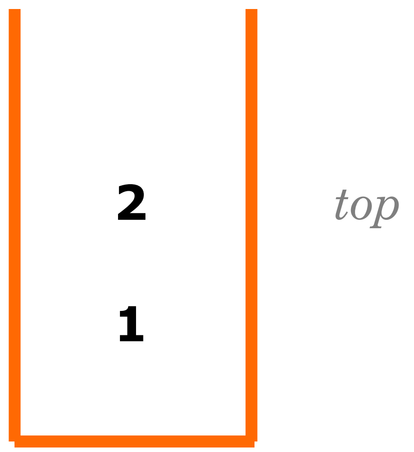
Pop the **element** at the **top** of the **stack.**

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

Output:

```
4
```

Tracing Code



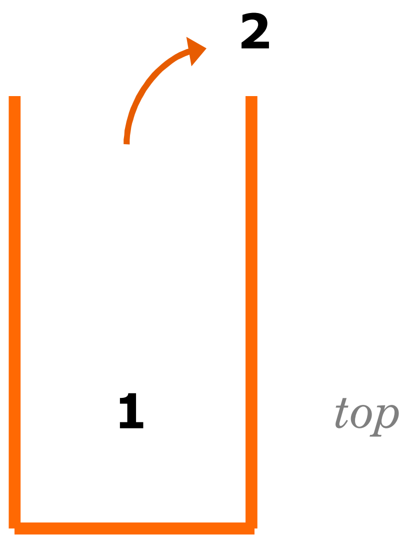
```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

Output:

```
4 2
```

Retrieve (*without* removing) the **element** at the **top** of the **stack** and print it.

Tracing Code



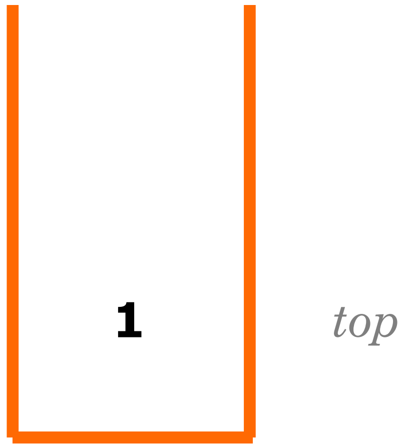
Pop the **element** at the **top** of the **stack.**

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

Output:

```
4 2
```

TRACING CODE



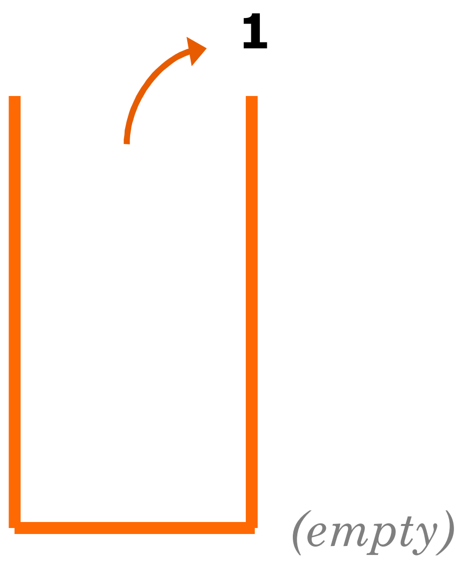
Retrieve (*without* removing) the **element** at the **top** of the **stack** and print it.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

Output:

```
4 2 1
```

Tracing Code



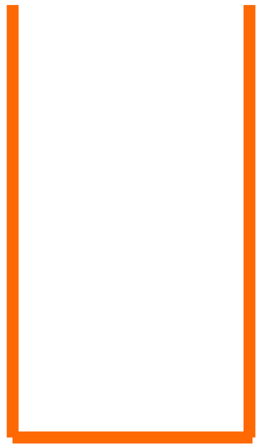
Pop the **element** at the **top** of the **stack.**

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

Output:

```
4
```

TRACING CODE



Stack is now **empty**; **WHILE** statement ends.

```
stack<int> myStack;
myStack.push(1);
myStack.push(2);
myStack.push(3);
if (!myStack.empty())
    myStack.pop();
myStack.push(4);
while (!myStack.empty())
{
    cout << myStack.top() << " ";
    myStack.pop();
}
```

Implementing a Stack

- Although the STL provides us a stack class, we can implement a stack as:
  - An **array**
    - How would you insert the elements at the top of the stack?

Implementing a Stack (cont.)

- Although the STL provides us a stack class, we can implement a stack as:
  - An **array**
    - How would you insert the elements at the top of the stack?
    - Easier if inserting from left to right
    - **Top** is at **index[numOfElements – 1]**

Implementing a Stack (cont.)

- Although the STL provides us a stack class, we can implement a stack as:
  - An **array**
    - How would you insert the elements at the top of the stack?
    - Easier if inserting from left to right
    - **Top** is at **index[numOfElements – 1]**
  - A **linked list**
    - How would you insert the elements at the top of the stack?

Implementing a Stack (cont.)

- Although the STL provides us a stack class, we can implement a stack as:
  - An **array**
    - How would you insert the elements at the top of the stack?
    - Easier if inserting from left to right
    - **Top** is at **index[numOfElements – 1]**
  - A **linked list**
    - How would you insert the elements at the top of the stack?

Implementing a Stack (cont.)

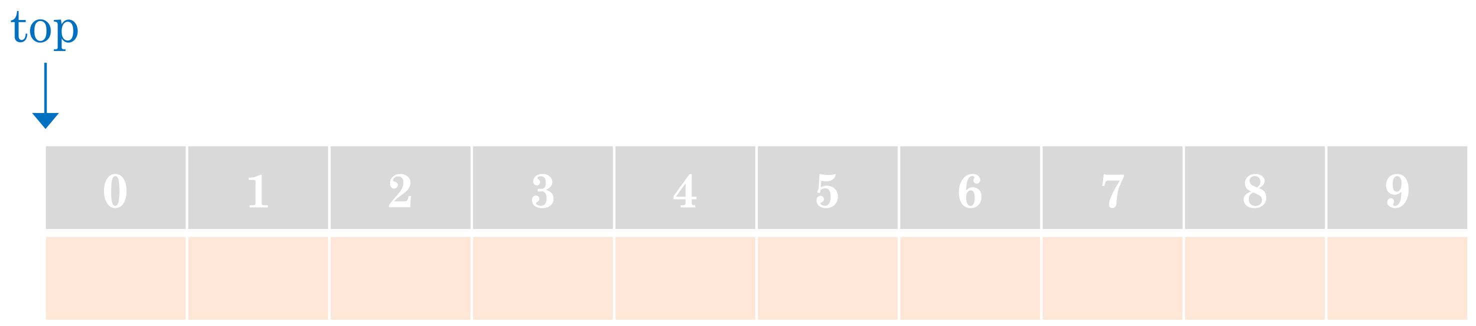
- Although the STL provides us a stack class, we can implement a stack as:
  - An **array**
    - How would you insert the elements at the top of the stack?
    - Easier if inserting from left to right
    - **Top** is at **index[numOfElements – 1]**

  - A **linked list**
    - How would you insert the elements at the top of the stack?
    - In a singly-linked list, the **top** is usually the **first** node

STACK ADT AS AN ARRAY

- Assume you are entering the following numbers, in this order, into the **stack**:

  3 7 2 6 8

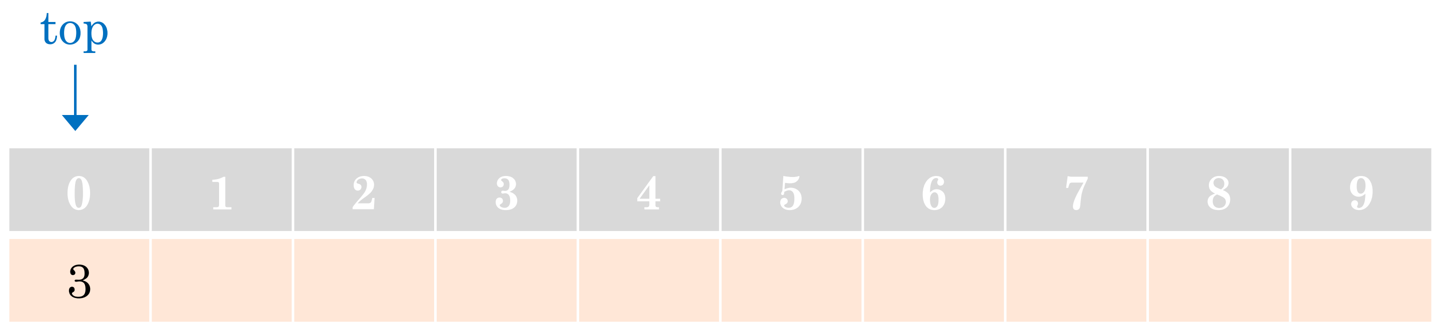


Stack ADT as an Array (cont.)

- Assume you are entering the following numbers, in this order, into the **stack**:

  3 7 2 6 8

  - Push 3 into the stack

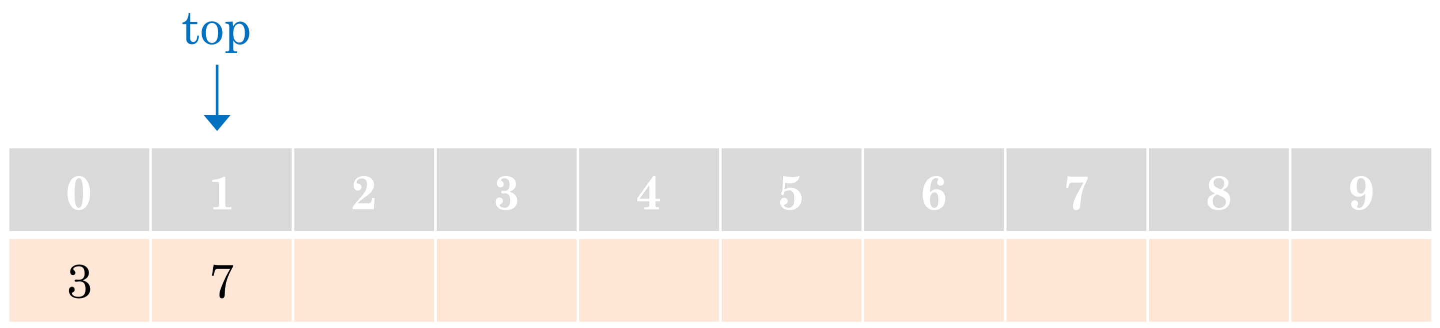


Stack ADT as an Array (cont.)

- Assume you are entering the following numbers, in this order, into the **stack**:

  3 7 2 6 8

  - Push 7 into the stack



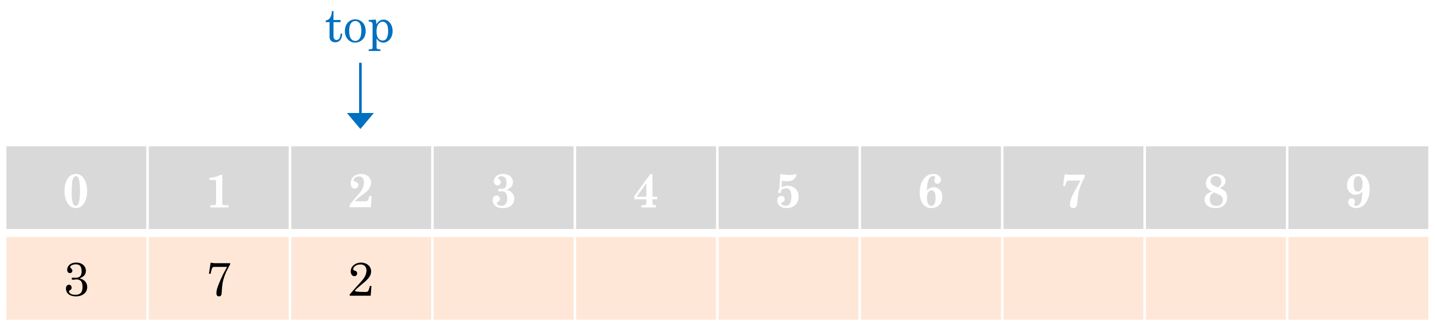
0	1	2	3	4	5	6	7	8	9
3	7								

Stack ADT as an Array (cont.)

- Assume you are entering the following numbers, in this order, into the **stack**:

  3 7 2 6 8

  - Push 2 into the stack



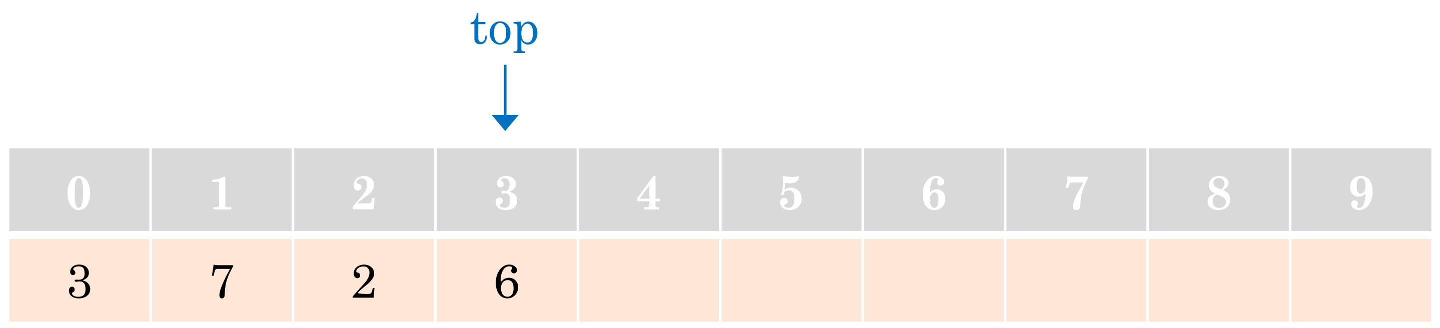
0	1	2	3	4	5	6	7	8	9
3	7	2							

Stack ADT as an Array (cont.)

- Assume you are entering the following numbers, in this order, into the **stack**:

  3 7 2 6 8

  - Push 6 into the stack



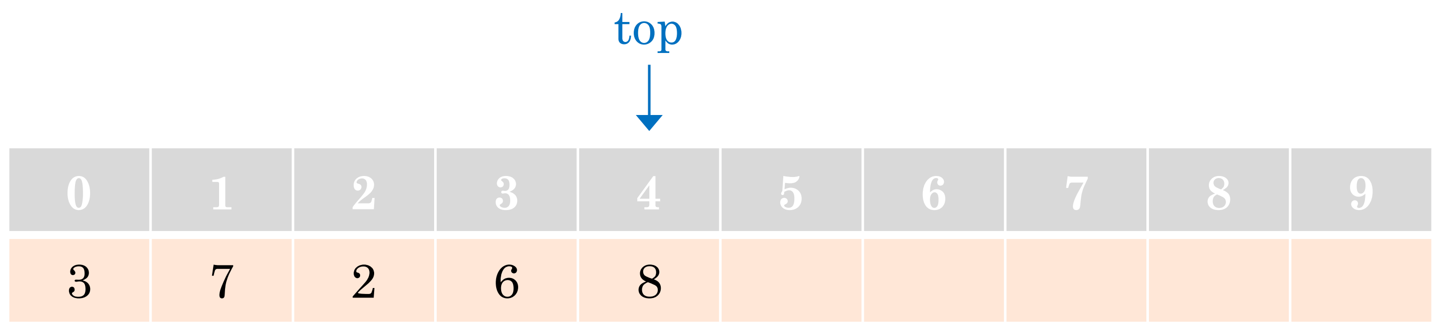
0	1	2	3	4	5	6	7	8	9
3	7	2	6						

STACK ADT AS AN ARRAY (CONT.)

- Assume you are entering the following numbers, in this order, into the **stack**:

  3 7 2 6 8

  - Push 8 into the stack



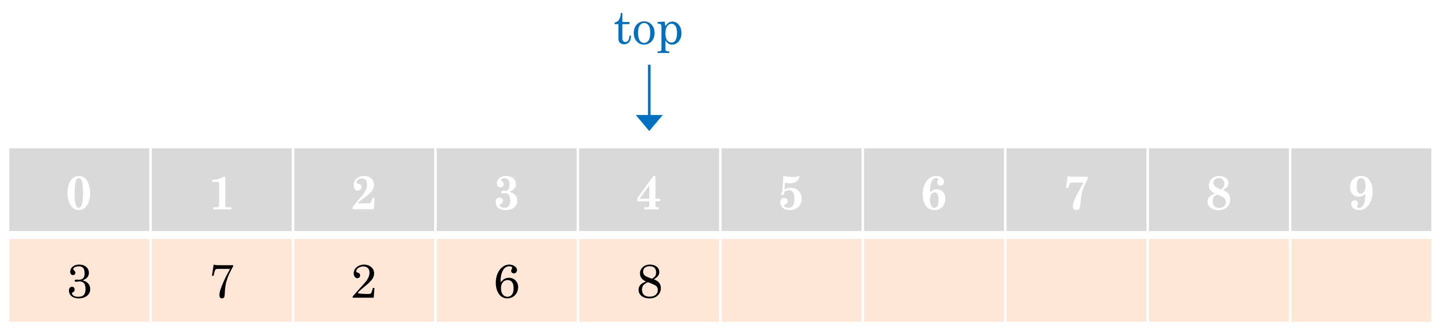
0	1	2	3	4	5	6	7	8	9
3	7	2	6	8					

STACK ADT AS AN ARRAY (CONT.)

- Assume you are entering the following numbers, in this order, into the **stack**:

  3 7 2 6 8

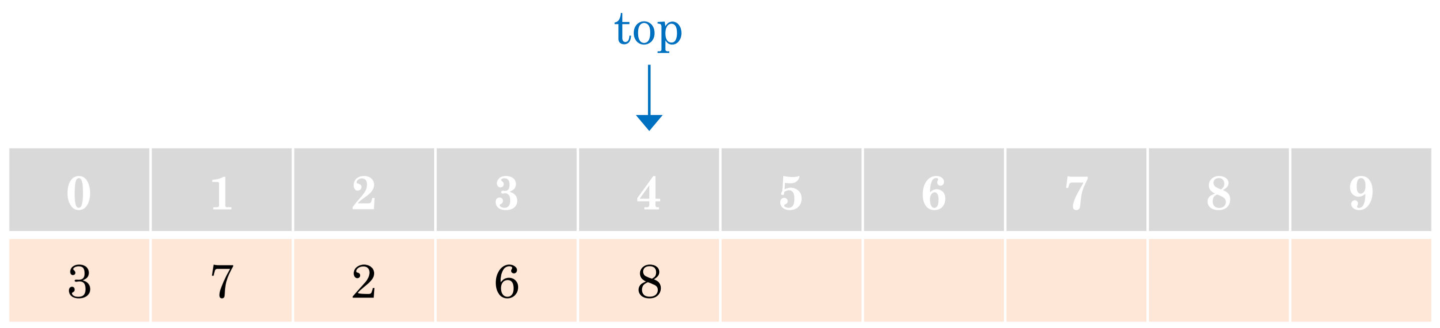
  - 3 will be at the **bottom** of the stack
  - 8 will be at the **top** of the stack
    - Variable **top** will be at **index 4**



0	1	2	3	4	5	6	7	8	9
3	7	2	6	8					

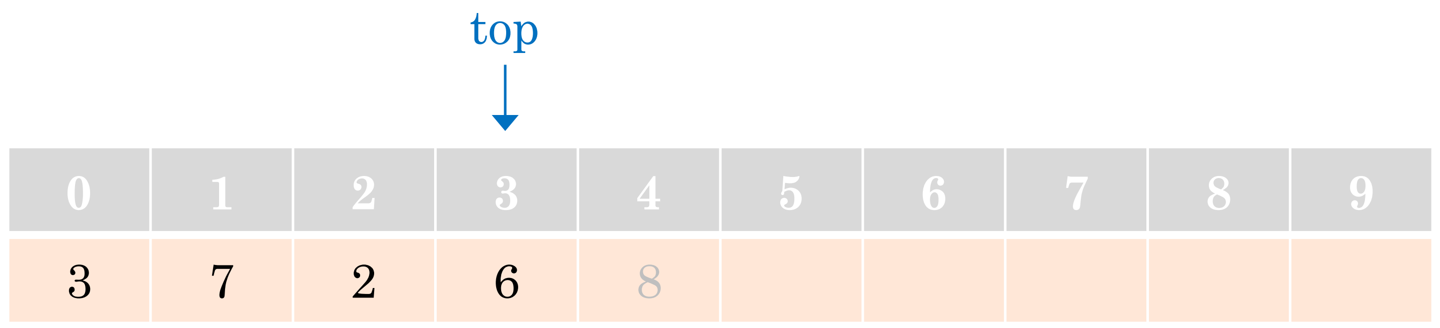
Stack ADT as an Array (cont.)

- If you need to **pop** the **top item** from the **stack**
  - What do you need to do?



Stack ADT as an Array (cont.)

- If you need to **pop** the **top item** from the **stack**
  - What do you need to do?
  - Simply move **top** to the previous index
    - No need to overwrite the element at index 4



0	1	2	3	4	5	6	7	8	9
3	7	2	6	8					

Stack ADT as a Singly-linked List

- Assume you are entering the following numbers, in this order, into the **stack**:

  7 2 6 4

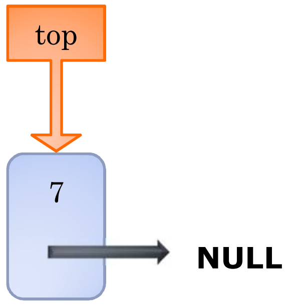
  - You only need the pointer **top**
    - It is actually the pointer you have been naming **first** (or **head**)

Stack ADT as a Singly-linked List

- Assume you are entering the following numbers, in this order, into the **stack**:

  7 2 6 4

  - Insert 7 to the **front** of the list

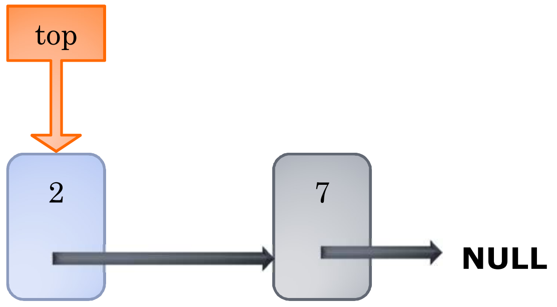


Stack ADT as a Singly-linked List

- Assume you are entering the following numbers, in this order, into the **stack**:

  7 2 6 4

  - Insert 2 to the **front** of the list

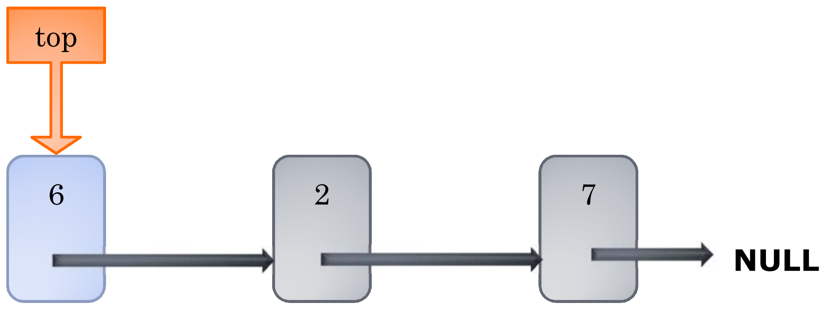


Stack ADT as a Singly-linked List

- Assume you are entering the following numbers, in this order, into the **stack**:

  7 2 6 4

  - Insert 6 to the **front** of the list

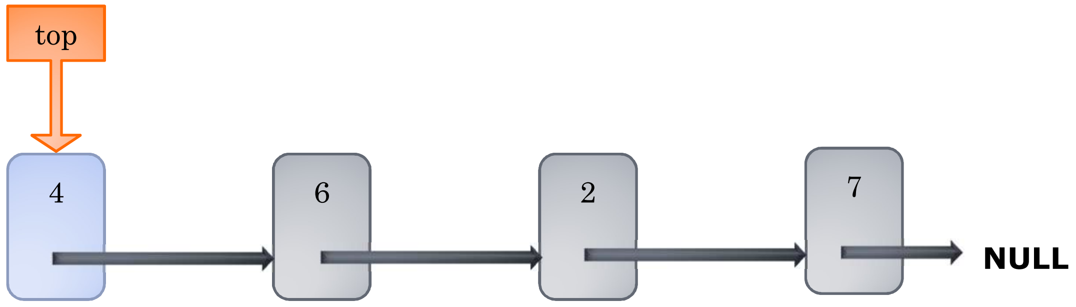


Stack ADT as a Singly-linked List

- Assume you are entering the following numbers, in this order, into the **stack**:

  7 2 6 4

  - Insert 4 to the **front** of the list



Common Operation Identifiers

- Other **identifiers** used for common operations on the **stack**:
  - **empty**( ) = **isEmpty( )**
  - **top**() = **peek( )** = **retrieve( )**
- **Note** that in some implementations the function **pop( )** returns a value **and** removes the element as well.

Stack Applications

- **Stacks** are used in many **applications**:
  - Track C++ function calls
  - Compilers perform syntax analysis (loops)
  - Back button in a browser
  - Undo button in a word processor (or other applications)
  - And more…

Stacks (end)