CS-2001

Data Structures

Fall 2022

Introduction to Stack ADT

Mr. Muhammad Yousaf

National University of Computer and Emerging Sciences,

Faisalabad, Pakistan.

Roadmap

Previous Lecture

- Variations of linked lists
  - Doubly linked lists
  - Circular Linked lists

Today

- Introduction to Stack
  - Common applications
  - Array Based-Implementation
  - Linked-list Based Implementation

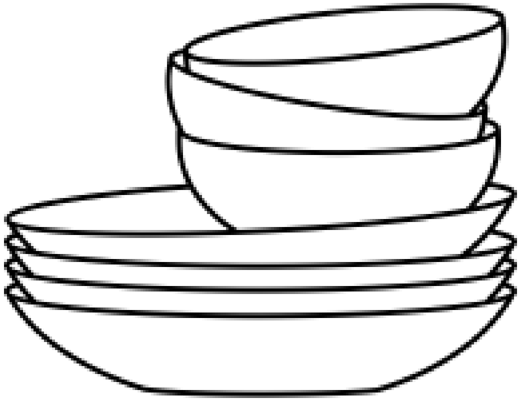
Stack

- A structure consisting of **homogeneous elements** and:
  - Insertion and deletions takes place at one end called top
  - It is a commonly used abstract data type with two major operations, namely push and pop.

- Other names
  - Last In First Out (LIFO) Structure
  - First In Last Out (FILO) Structure

Real life scenarios…

- Books on floor
- Dishes on a shelf
- In programming, consider doing **`X = (A+B) * (C+D)`**

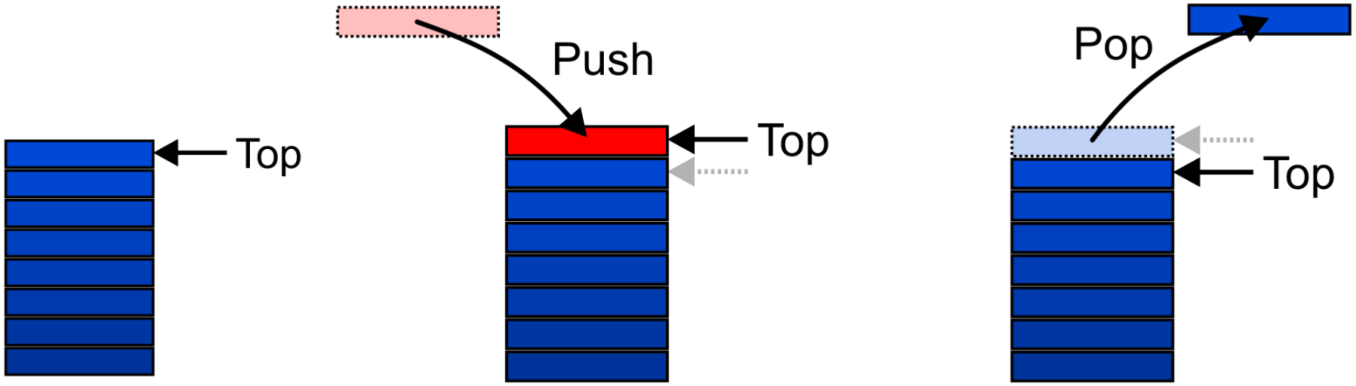


Stack ADT Operations

- Stack ADT emphasizes specific operations
  - Uses an explicit linear ordering (Random access is intentionally revoked)
  - Insertions and removals are performed individually.
  - Inserted objects are pushed onto the stack
  - Top of the stack is the most recent object pushed onto the stack
  - Push and pop operations changes the current top value of the stack

Stack ADT – Operations (1)

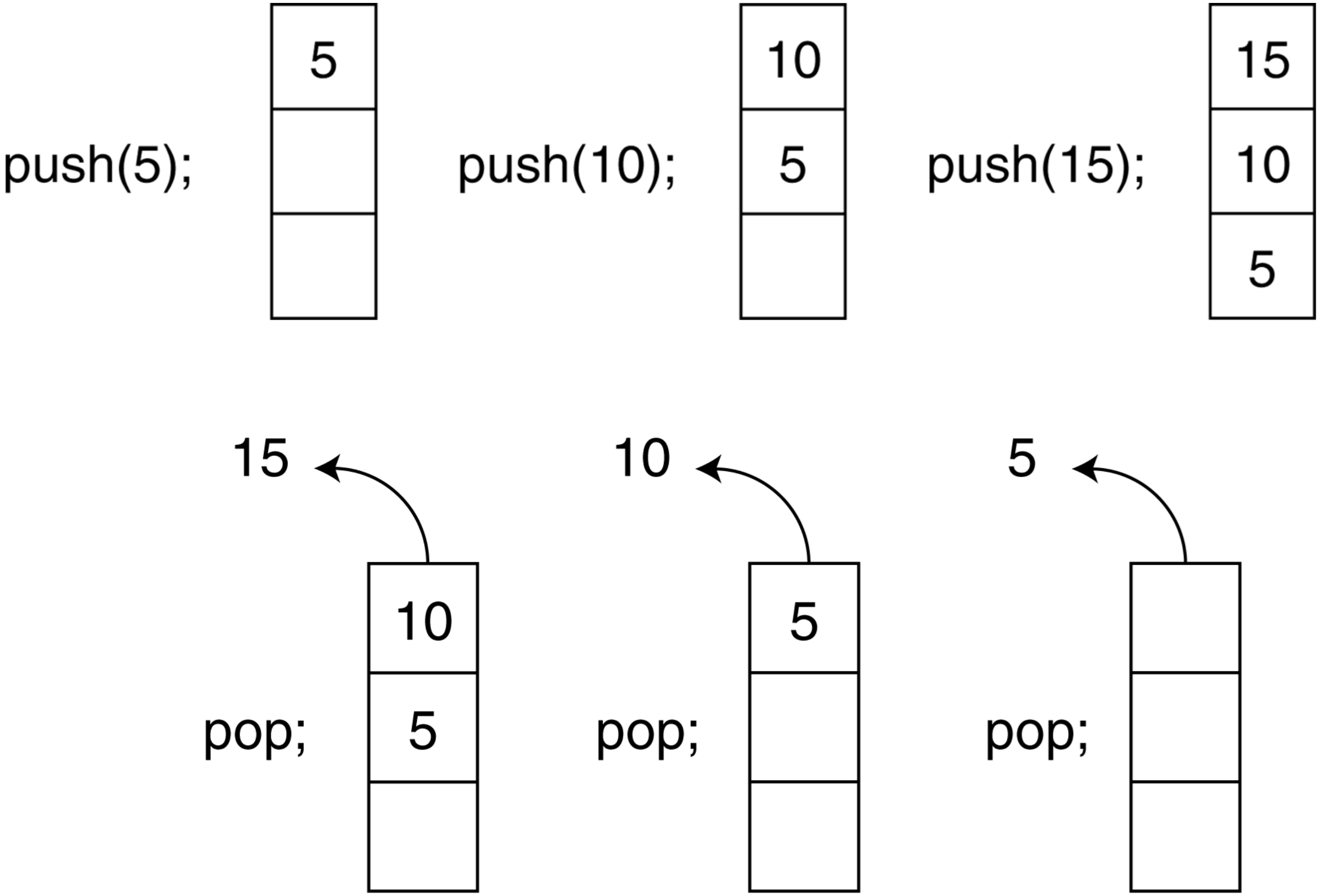
- Graphically, the stack operations are viewed as follows:



Stack ADT – Operations (2)

- CreateStack(S)
  - Make Stack S be an empty stack
- Top(S)
  - Return the element at the top of stack S
- Pop(S)
  - Remove the top element of the stack
- Push(S,x)
  - Insert the element x at the top of the stack
- Empty(S)
  - Return true if S is an empty stack and return false otherwise

Push and Pop Operations of Stack



push(5);

5

push(10);

10

5

push(15);

15

10

5

15

pop;

10

5

10

pop;

5

5

pop;

Applications (1)

- Many applications
  - Parsing code
    - ➢ Matching parenthesis problem
  - Tracking function calls (Call stack)
  - Reversing a string
  - Infix to postfix Conversion
  - Backtracking in Depth-First-Search

- The stack is a very simple data structure
  - Given any problem, if it is possible to use a stack, this significantly simplifies the solution

Use of Stack in Function Calls (1)

- When a function begins execution an activation record is created to store the current execution environment for that function

- Activation records all the necessary information about a function call, including
  - arguments passed by the caller function
  - Local variables
  - Content of the registers
  - Return address to the caller function
    - Address of instruction following the function call

Use of Stack in Function Calls (2)

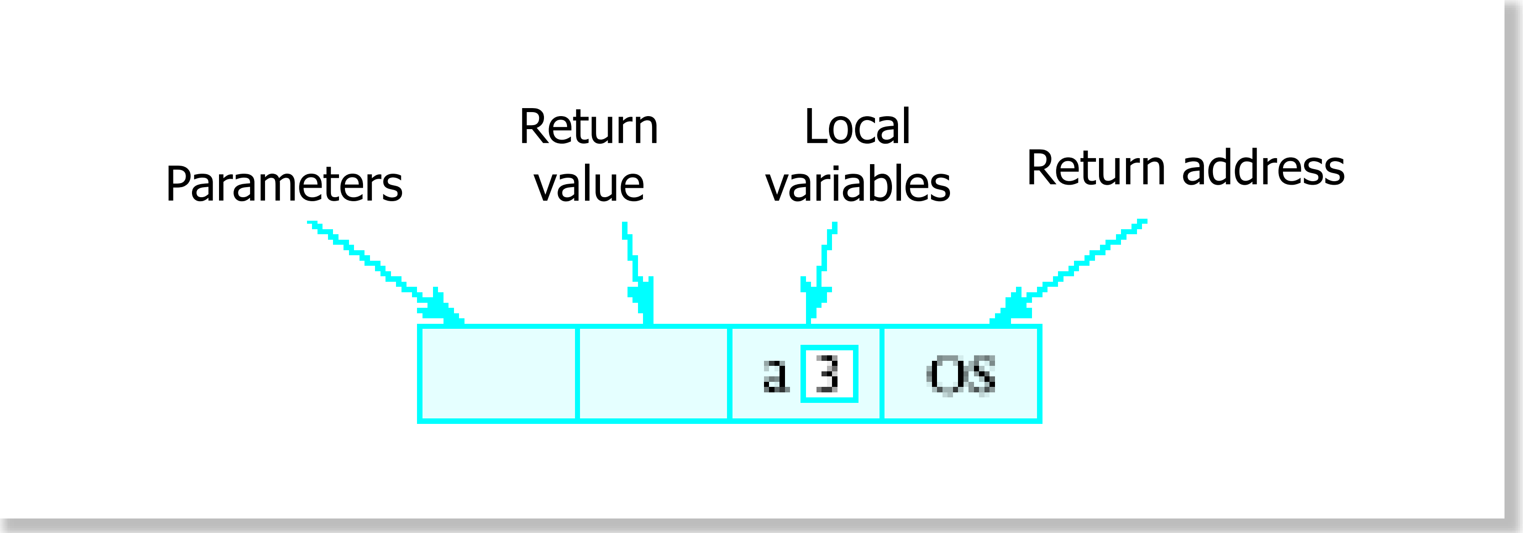
- Each invocation of a function has its own activation record
- Recursive/Multiple calls to the functions require several activation records to exist simultaneously
- A function returns only after all functions it calls have returned Last In First Out (LIFO) behavior
- A program/OS keeps track of all the functions that have been called using run-time stack or call stack

Runtime Stack Example (1)

```
void main(){
   int a=3;
   f1(a); // statement A
   cout << endl;
}

void f1(int x){
   cout << f2(x+1); // statement B
}

int f2(int p){
   int q=f3(p/2); // statement C
   return 2*q;
}

int f3(int n){
   return n*n+1;
}
```

Runtime Stack

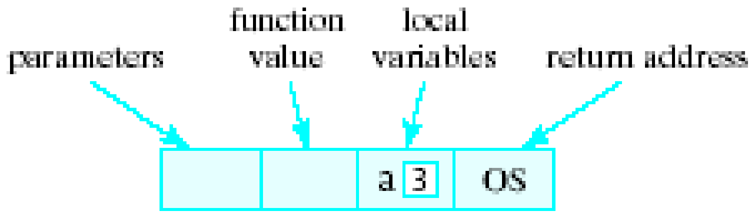
- When a function is called …
  - Copy of activation record pushed onto run-time stack
  - Arguments copied into parameter spaces
  - Control is transferred to starting address of body of function

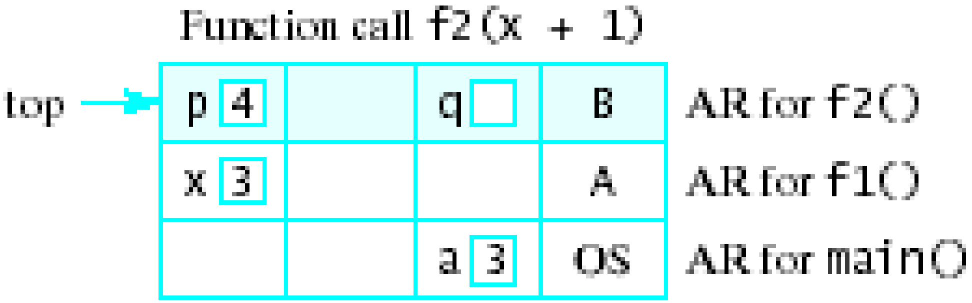


OS denotes that when execution of main() is completed, it returns to the operating system

Runtime Stack Example (2)

```
void main(){
   int a=3;
   f1(a); // statement A
   cout << endl;
}

void f1(int x){
   cout << f2(x+1); // statement B
}

int f2(int p){
   int q=f3(p/2); // statement C
   return 2*q;
}

int f3(int n){
   return n*n+1;
}
```





p 4		q □	B	AR for f2()
x 3			A	AR for f1()
		a 3	OS	AR for main()

Static and Dynamic Stacks

- Two possible implementations of stack data structure
  - Static, i.e., fixed size implementation using arrays
  - Dynamic implementation using linked lists

Stack – Library

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
        stack<int> stack;
        stack.push(21);
        stack.push(22);
        stack.push(24);
        stack.push(25);

        stack.pop();
        stack.pop();

        while (!stack.empty()) {
                cout << stack.top() <<" ";
                stack.pop();
        }
}
```

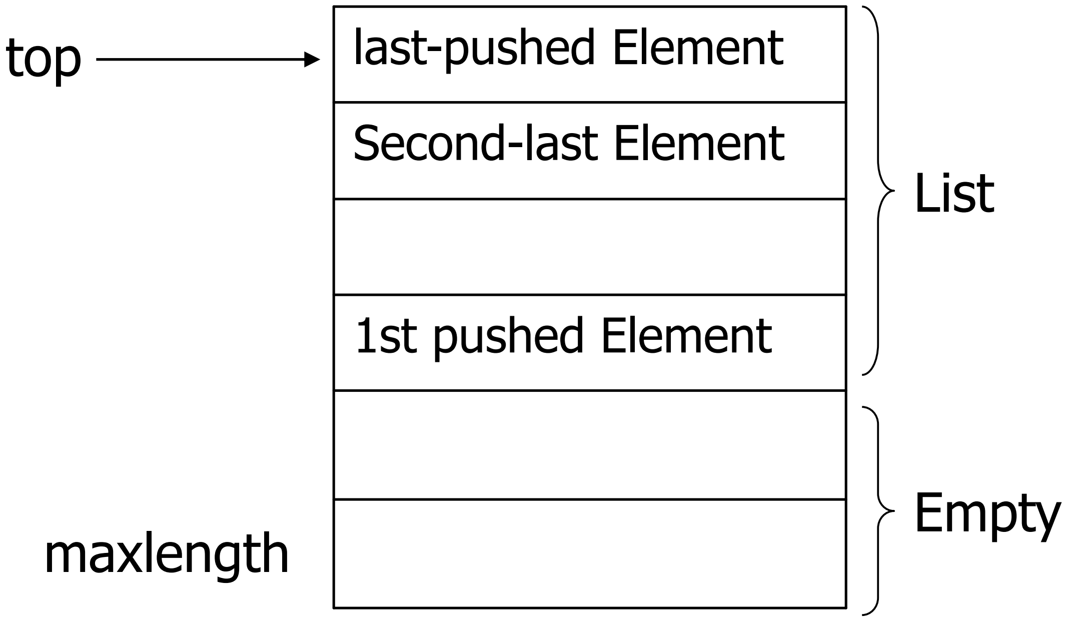
Output: 22 21

Array-based Implementation

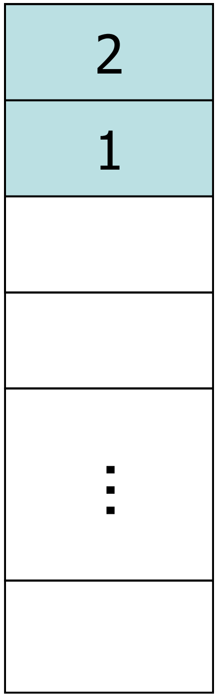
Array Implementation – First Solution (1)

- Elements are stored in contiguous cells of an array

- New elements can be inserted to the top of the list

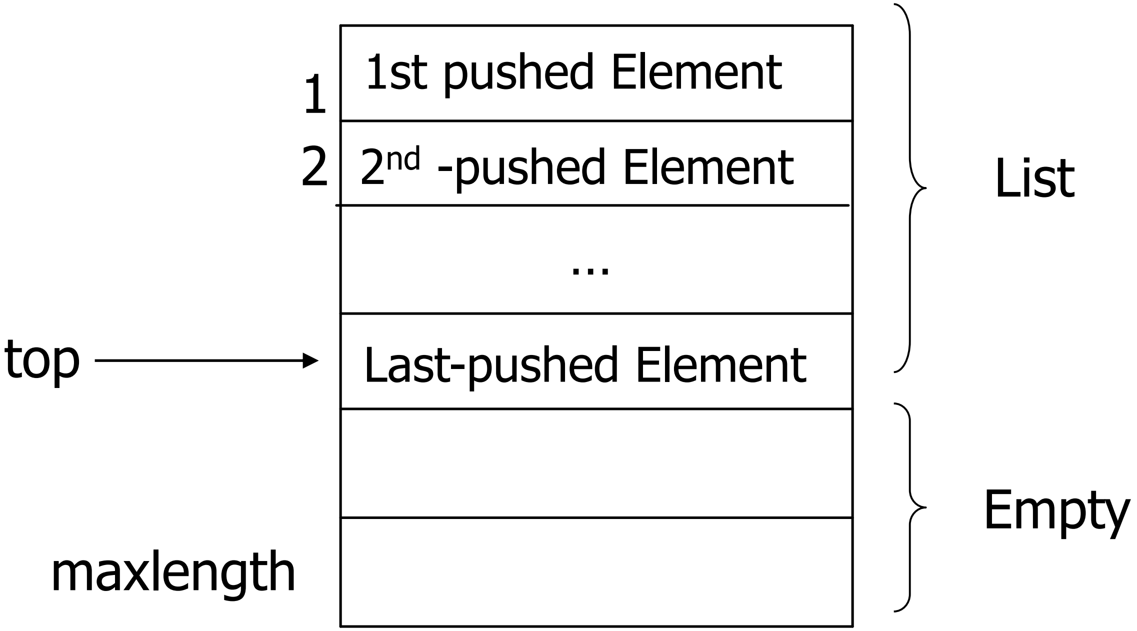


Array Implementation – First Solution (2)



- Problem
  - Every PUSH and POP requires moving the entire array up and down
  - Fixed size

Array Implementation – Tweaked Solution (2)



Idea

- Anchor the bottom of the stack at the start of the array
- Let the stack grow towards the last of the array
- Top indicates the current position of the recently inserted stack element

Array Implementation – Code (1)

```
class IntStack
{
   private:
      int *stackArray;
      int stackSize;
      int top;

   public:
      IntStack(int);
      ~IntStack( );
      bool push(int);
      bool pop(int &);
      bool isFull();
      bool isEmpty();
};
```

Array Implementation – Code (2)

- Constructor

```
IntStack::IntStack(int size) //constructor
{
   stackArray = new int[size];
   stackSize = size;
   top = -1;
}
```

- Destructor

```
IntStack::~IntStack(void) //destructor
{
   delete [] stackArray;
}
```

Array Implementation – Code (3)

- `isFull` function

```
bool IntStack::isFull(void)
{
    if (top == stackSize - 1)
        return true;
    else
        return false;
    // return (top == stackSize-1);
}
```

- `isEmpty` function

```
bool IntStack::isEmpty(void)
{
    return (top == -1);
}
```

Array Implementation – Code (4)

- `push` function inserts the argument `num` onto the stack

```
bool IntStack::push(int num)
{
   if (isFull())
   {
      cout << "The stack is full.\n";
      return false;
   }

   top++;
   stackArray[top] = num;
   return true;
}
```

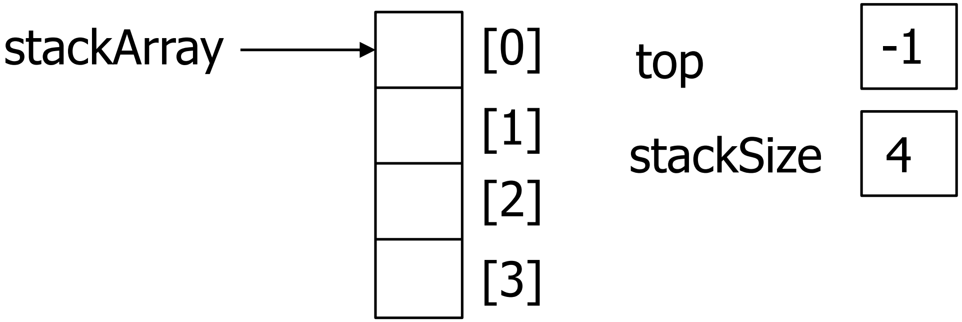
Array Implementation – Code (5)

- `Pop` function removes the value from top of the stack and returns it as a reference

```
bool IntStack::pop(int &num)
{
   if (isEmpty())
   {
      cout << "The stack is empty.\n";
      return false;
   }

   num = stackArray[top];
   top--;
   return true;
}
```

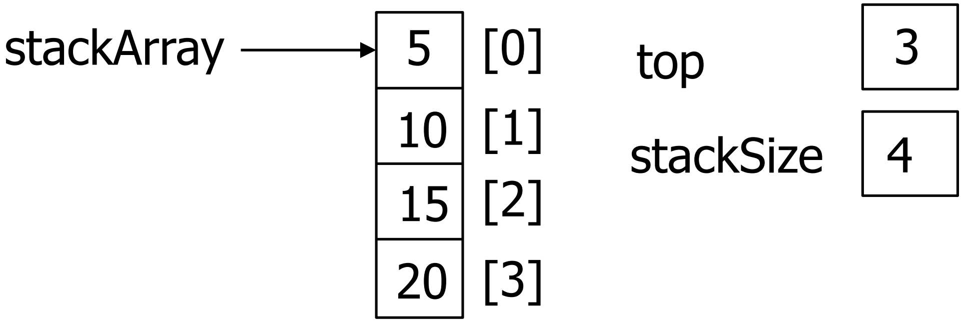
Using Stack (1)

```
int main()
{
   IntStack stack(4);

}
```



Using Stack (2)

```
int main()
{
   IntStack stack(4);
   int catchVar;

   cout << "Pushing Integers\n";
   stack.push(5);
   stack.push(10);
   stack.push(15);
   stack.push(20);















}
```



stackArray

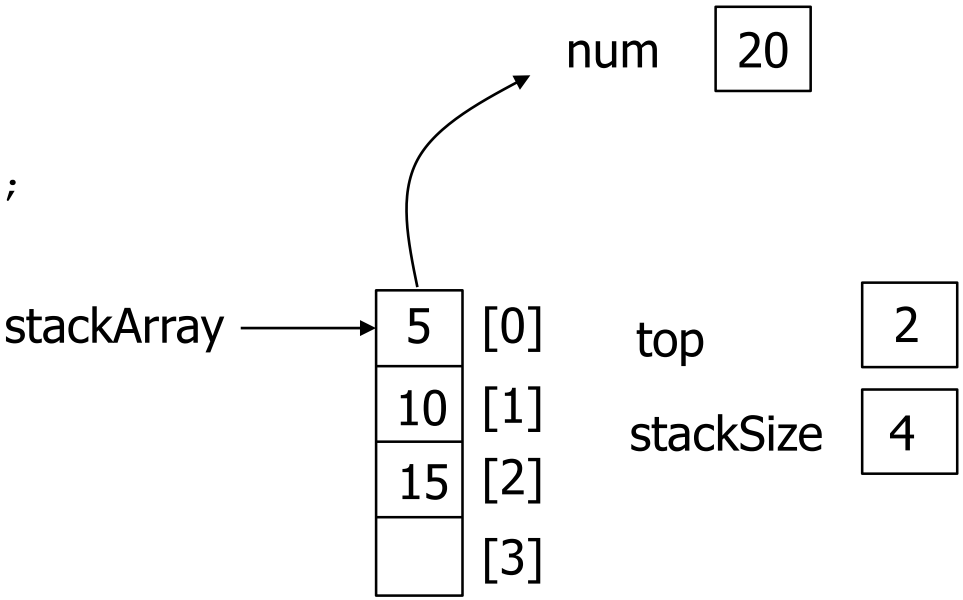
5	[0]
10	[1]
15	[2]
20	[3]

top 3

stackSize 4

Using Stack (3)

```
int main()
{
   IntStack stack(4);
   int catchVar;

   cout << "Pushing Integers\n";
   stack.push(5);
   stack.push(10);
   stack.push(15);
   stack.push(20);

   cout << "Popping...\n";
   stack.pop(catchVar);
   cout << catchVar << endl;






}
```



Using Stack (4)

```
int main()
{
   IntStack stack(4);
   int catchVar;

   cout << "Pushing Integers\n";
   stack.push(5);
   stack.push(10);
   stack.push(15);
   stack.push(20);

   cout << "Popping...\n";
   stack.pop(catchVar);
   cout << catchVar << endl;
   stack.pop(catchVar);
   cout << catchVar << endl;
   stack.pop(catchVar);
   cout << catchVar << endl;
   stack.pop(catchVar);
   cout << catchVar << endl;
   return 0;
}
```

Output:

```
Pushing Integers
Popping…
20
15
10
5
```

Any Question So Far?