Data Structures

12. Stacks

Stack

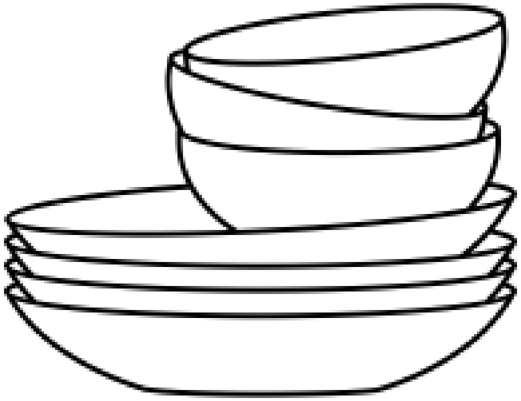
- A stack is a special kind of list
  - Insertion and deletions takes place at one end called top

- Other names
  - Push down list
  - Last In First Out (LIFO)

Stack Examples

- Books on floor

- Dishes on a shelf

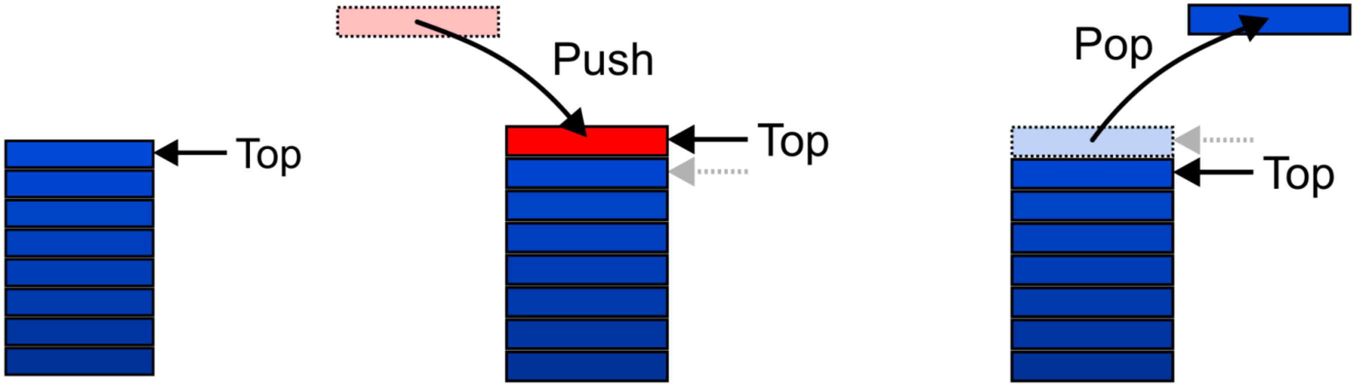


Stack ADT

- Stack ADT emphasizes specific operations
  - Uses a explicit linear ordering
  - Insertions and removals are performed individually
  - Inserted objects are pushed onto the stack
  - Top of the stack is the most recently object pushed onto the stack
  - When an object is popped from the stack, the current top is erased

Stack ADT – Operations (1)

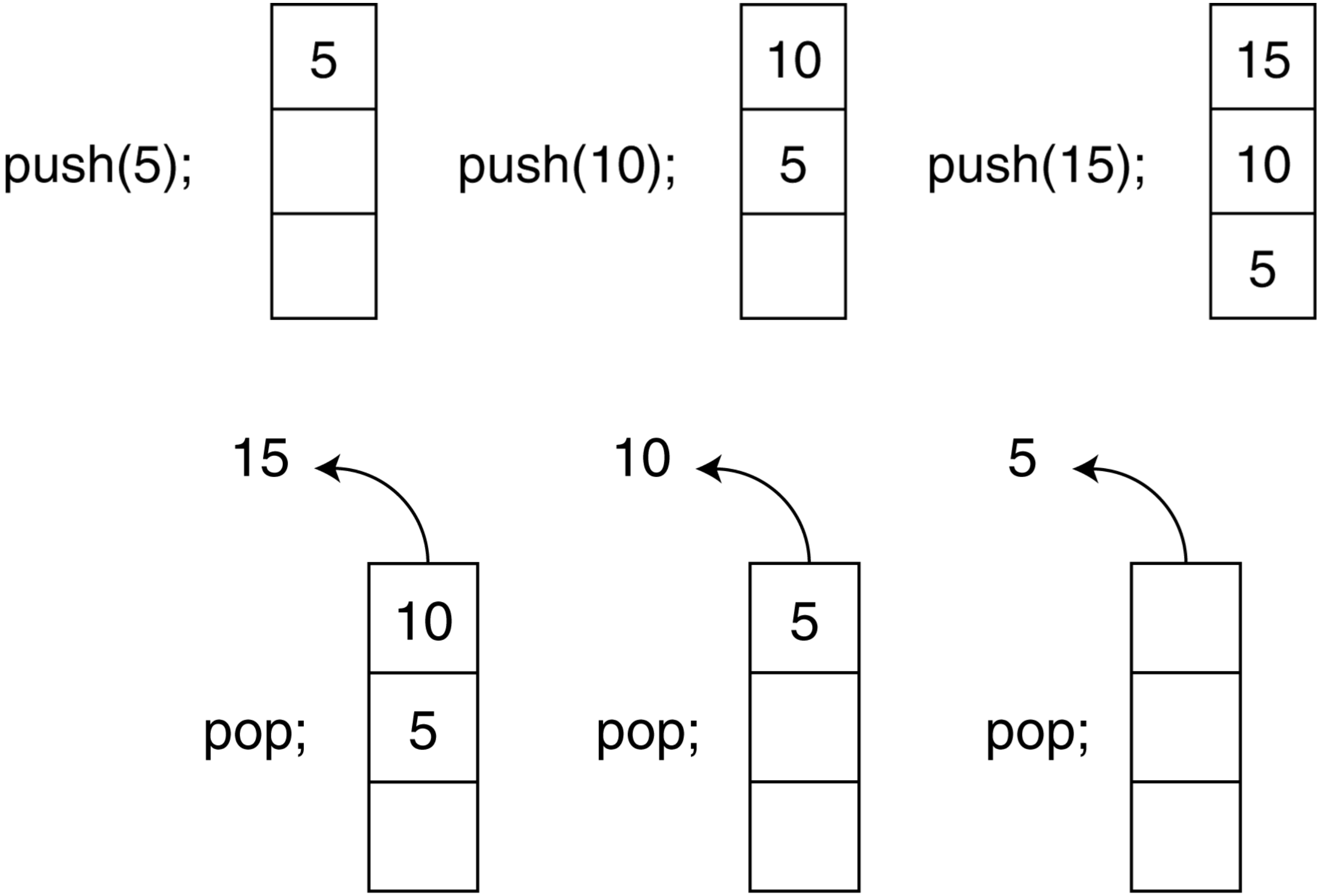
- Graphically, the stack operations are viewed as follows:



Stack ADT – Operations (2)

- MAKENULL(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

pop;
15
10
5

pop;
10
5

pop;
5

Applications (1)

- Many applications
  - Parsing code
    - Matching parenthesis
    - XML (e.g., XHTML)
  - Tracking function calls
  - Dealing with undo/redo operations

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

Applications (2)

- Problem solving
  - Solving one problem may lead to subsequent problems
  - These problems may result in further problems
  - As problems are solved, focus shifts back to the problem which lead to the solved problem

- Notice that function calls behave similarly
  - A function is a collection of code which solves a problem

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 record contains all the necessary information about a function call, including
  - Parameters passed by the caller function
  - Local variables
  - Content of the registers
  - (Callee) Function's return value(s)
  - Return address of the caller function
    - ➢ Address of instruction following the function call

Use of Stack in Function Calls (2)

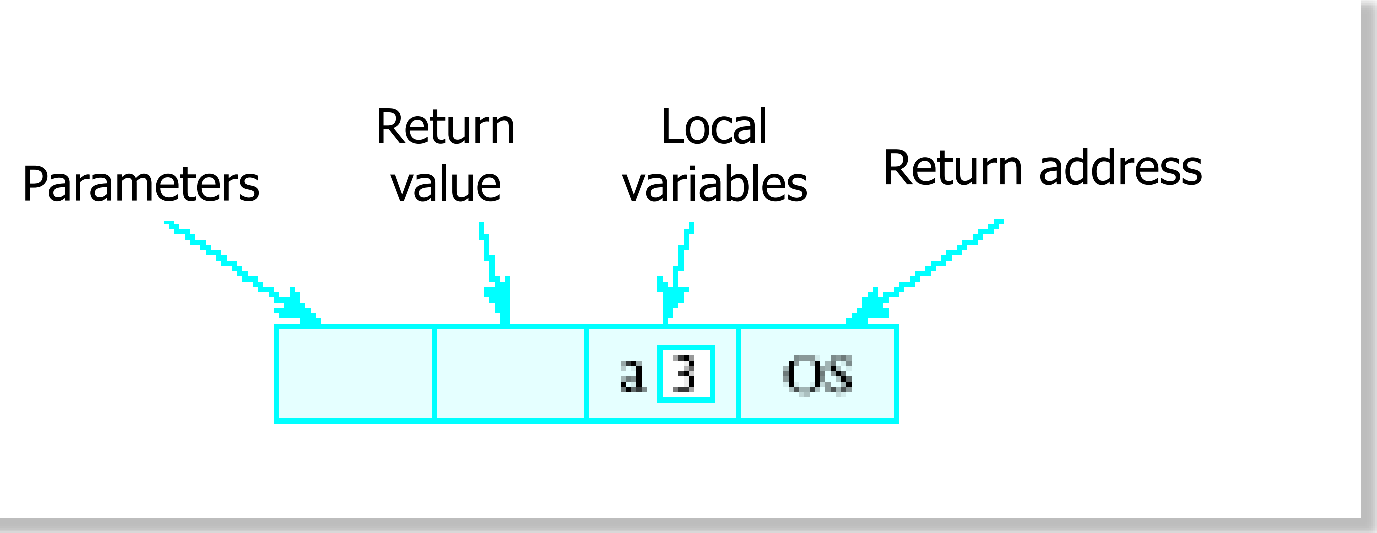
- Each invocation of a function has its own activation record
- Recursive/Multiple calls to the functions require several activation records to exist simultaneously
- A function returns only after all functions it calls have returned Last In First Out (LIFO) behavior
- A program/OS keeps track of all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

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

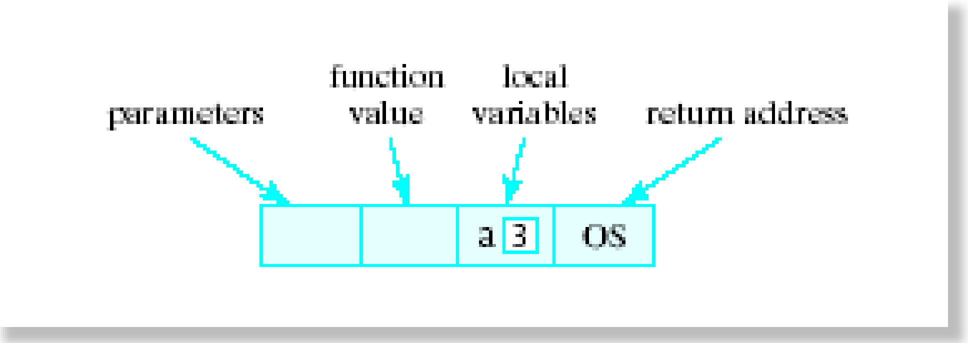
- When a function is called …
  - Copy of activation record pushed onto run-time stack
  - Arguments copied into parameter spaces
  - Control transferred to starting address of body of function

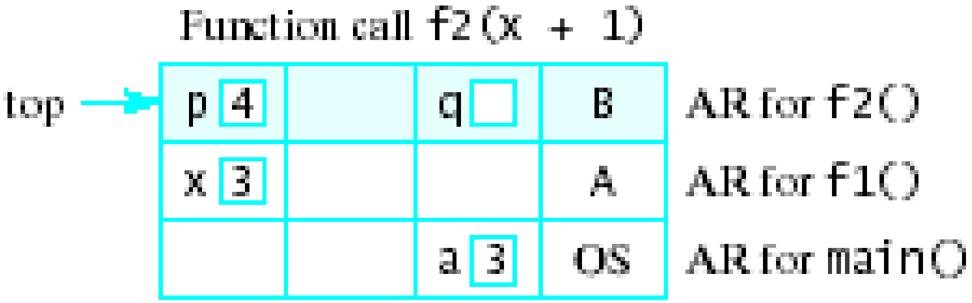


OS denotes that when execution of main() is completed, it returns to the operating system

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;
}
```





Static and Dynamic Stacks

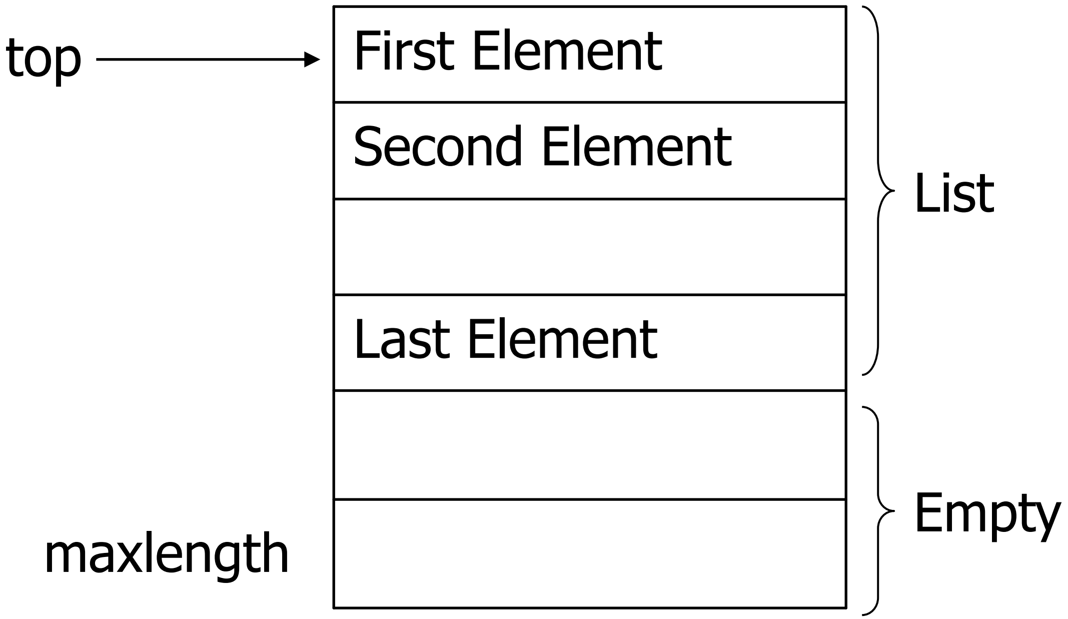
- Two possible implementations of stack data structure
  - Static, i.e., fixed size implementation using arrays
  - Dynamic implementation using linked lists

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



top

First Element

Second Element

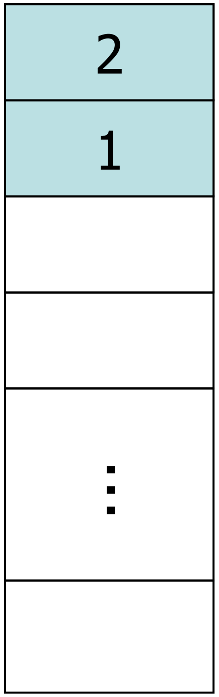
Last Element

List

Empty

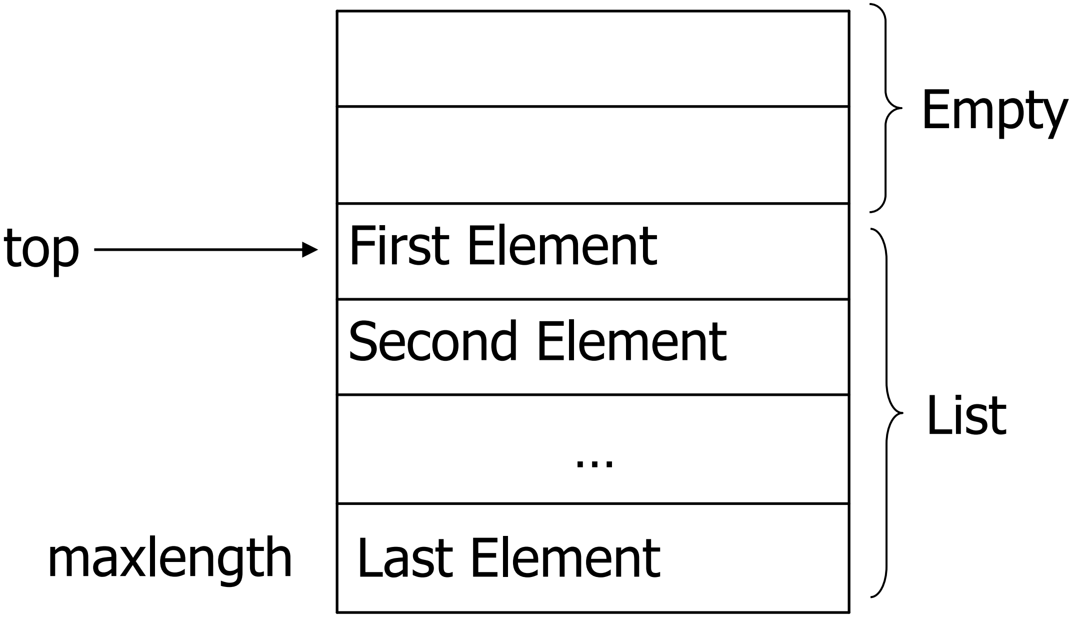
maxlength

Array Implementation – First Solution (2)



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

Array Implementation – Better Solution (2)



Idea

- Anchor the top of the stack at the bottom of the array
- Let the stack grow towards the top of the array
- Top indicates the current position of the first stack element

Array Implementation – Code (1)

```
#ifndef INTSTACK_H
#define INTSTACK_H

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

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

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)
{
   bool status;
   if (top == stackSize - 1)
      status = true;
   else
      status = false;
   return status; // 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

```
void IntStack::push(int num)
{
   if (isFull())
   {
      cout << "The stack is full.\n";
   }
   else
   {
      top++;
      stackArray[top] = num;
   }
}
```

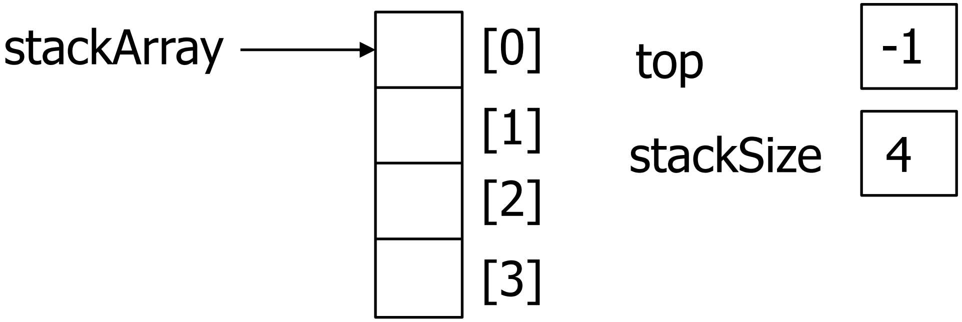
Array Implementation – Code (5)

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

```
void IntStack::pop(int &num)
{
   if (isEmpty())
   {
      cout << "The stack is empty.\n";
   }
   else
   {
      num = stackArray[top];
      top--;
   }
}
```

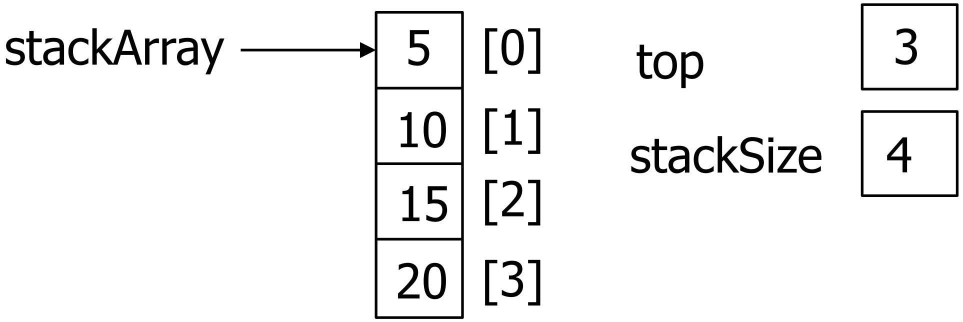
Using Stack (1)

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

}
```



Using Stack (2)

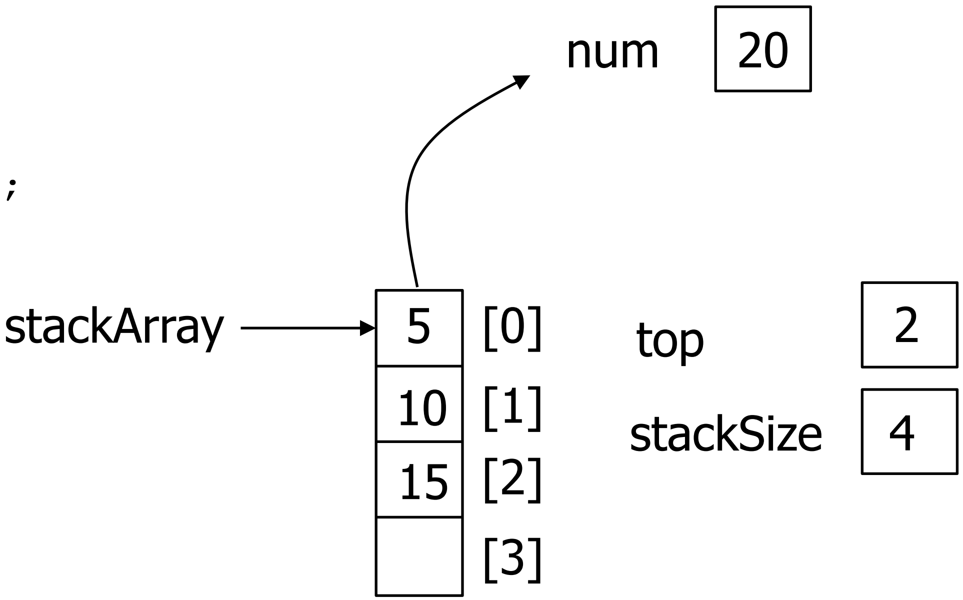
```
void main(void)
{
   IntStack stack(4);
   int catchVar;

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



```
}
```

Using Stack (3)

```
void main(void)
{
   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)

```
void main(void)
{
   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;

}
```

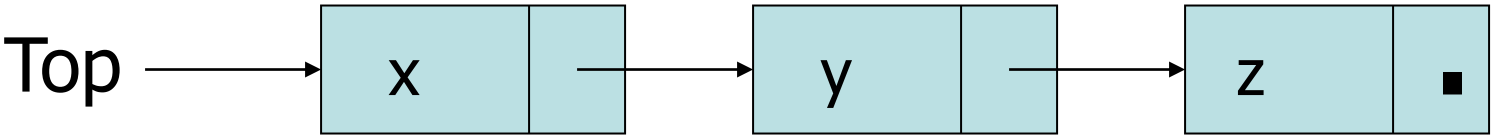
Output:

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

Pointer-based Implementation

Pointer-based Implementation of Stacks

- Stack can expand or shrink with each push or pop operation
- Push and pop operate only on the header cell, i.e., the first cell of the list



Pointer Implementation – Code (1)

```
class Stack
{
    struct node
    {
        int data;
        node *next;
    }*top;

    public:
        Stack( );
        ~Stack( );
        void Push(int newelement);
        void Pop(int &);
        bool IsEmpty();
};
```

Pointer Implementation – Code (2)

- `IsEmpty` function returns true if the stack is empty

```
bool Stack::IsEmpty()
{
    if (top==NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Pointer Implementation – Code (3)

- `Push` function inserts a node at the top/head of the stack

```
void Stack::Push(int newelement)
{
   node *newptr;
   newptr=new node;

   newptr->data=newelement;
   newptr->next=top;

   top=newptr;
}
```

Pointer Implementation – Code (4)

- `Pop` function deletes the node from the top of the stack and returns its data by reference

```
void Stack:Pop(int& value)
{
   if (IsEmpty())
   {
      cout<<“underflow error”;
   }
   else
   {
      tempptr = top;
      value = top->data;
      top = top->next;
      delete tempptr;
   }
}
```

Any Question So Far?