National University of Computer & Emerging Sciences

Stacks

Stacks

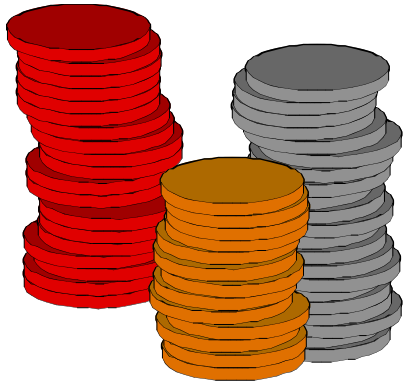
"A ***Stack*** is a special kind of list in which all insertions and deletions take place at one end, called the ***Top***"

Other Names

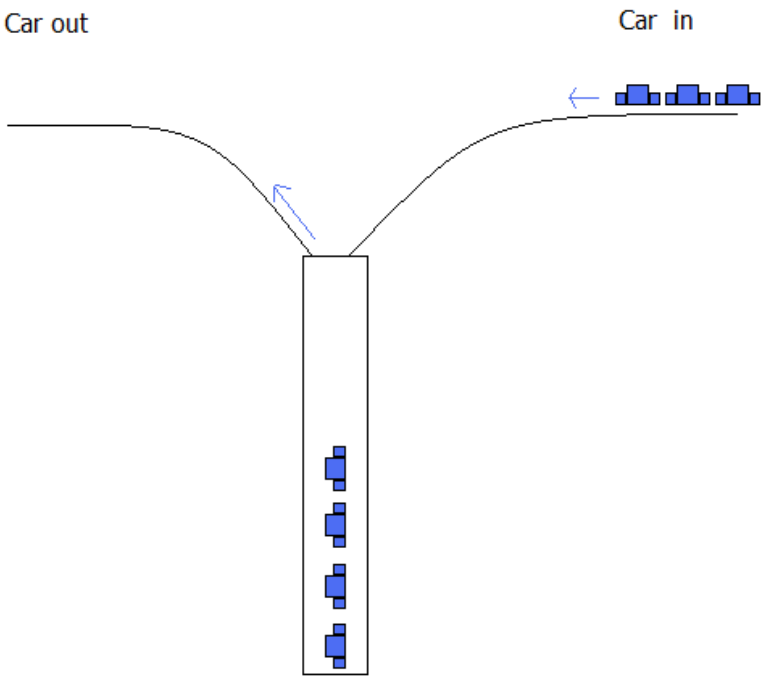
- Pushdown List
- Last In First Out (LIFO)

Stacks (examples)

- Books on a floor
- Dishes on a shelf



Stacks (examples)



Is there an appropriate data type to model this parking lot???

Use of Stack in Function calls

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

- Current environment includes the following (and more):
  - values of its parameters,
  - contents of registers,
  - the function's return value,
  - local variables
  - address of the instruction to which execution is to **return** when the function finishes execution (If execution is interrupted by a call to another function)

Use of Stack in Function calls

- Functions may call other functions and thus interrupt their own execution, some data structure must be used to store these activation records so they can be recovered and the system can be reset when a function resumes execution
- It is the fact that the last function interrupted is the first one reactivated
- A stack is the appropriate structure, and since it is manipulated during execution, it is called the **run-time stack**

Consider the following program

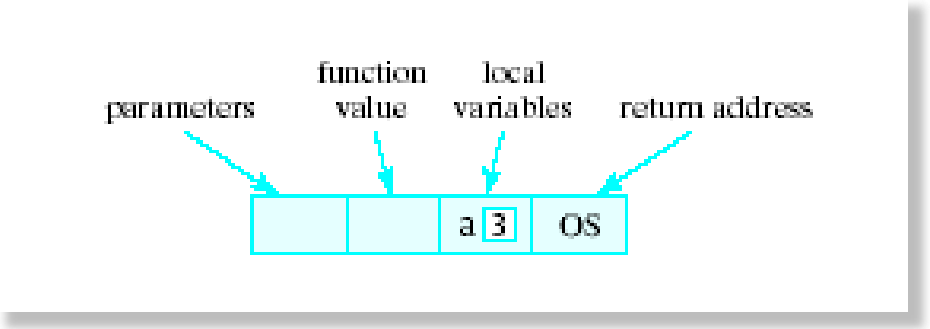
```
void main(){
    int a=3;
    f1(a); // statement A
    cout << endl;
}

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;
}
```

Run-time Stack

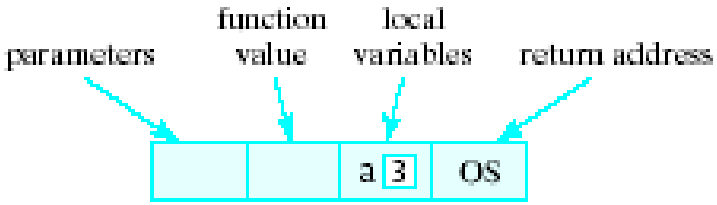


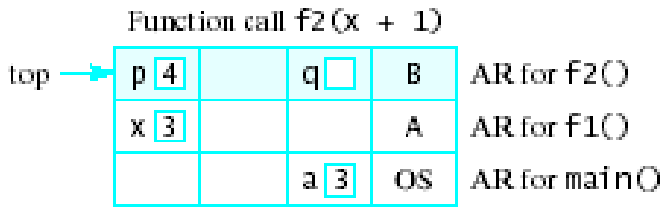
- OS denotes that when execution of main() is completed, it returns to the operating system

Use of Run-time 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





Stacks (examples)

- In general
  - A ***Stack*** is a special kind of list in which all insertions and deletions take place at one end, called the ***Top***
  - Last In First Out (LIFO)

Common Operations on Stacks

1. **MAKENULL(*S*):** Make Stack S be an empty stack.
2. **TOP(*S*):** Return the element at the top of stack *S*.

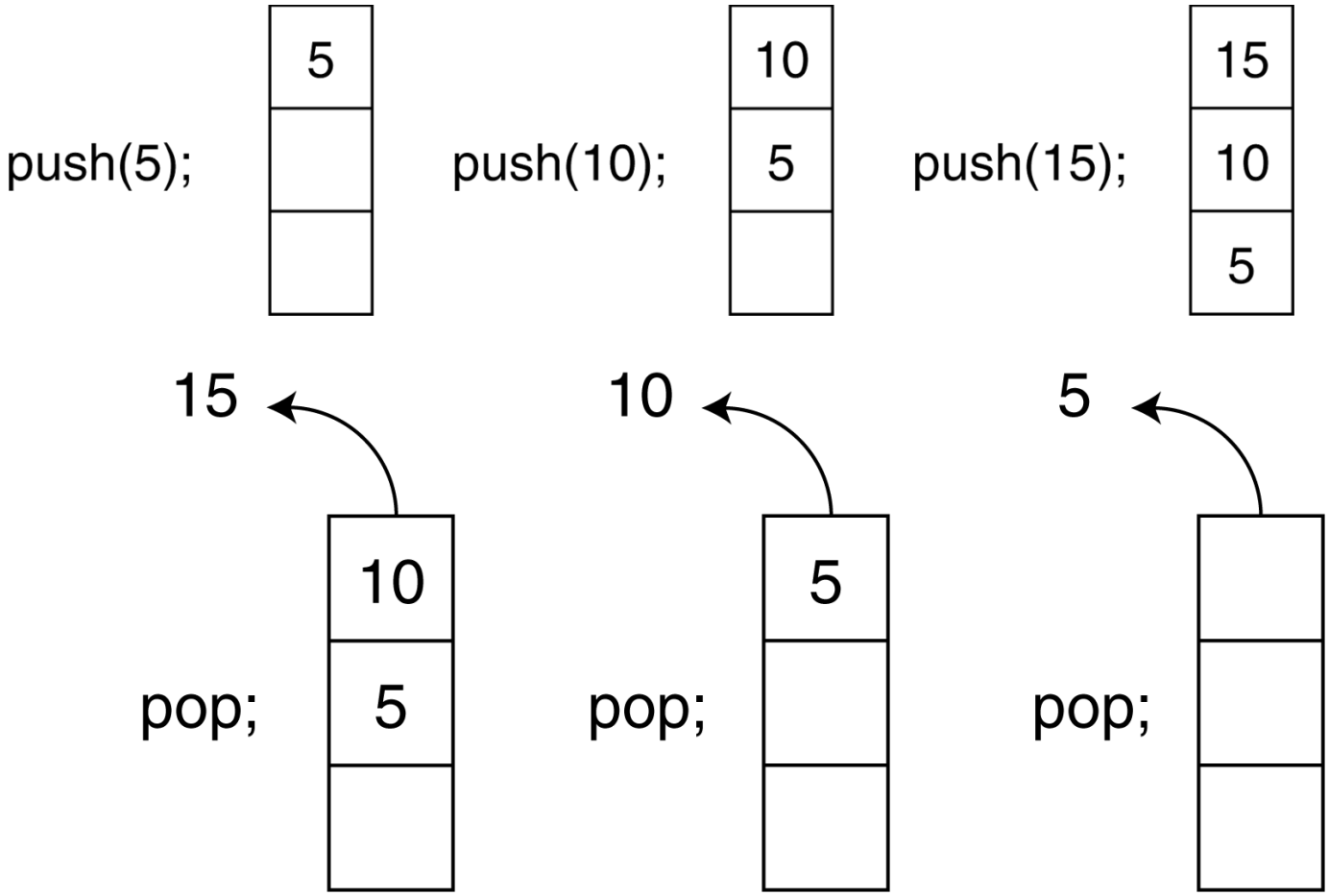
3. **POP(*S*):** Remove the top element of the stack.
4. **PUSH(*S*,*x*):** Insert the element *x* at the top of the stack.

5. **EMPTY(*S*):** Return true if S is an empty stack; return false otherwise.

Static and Dynamic Stacks

- There are two kinds of stack data structure -
  - a) **static**, i.e. they have a **fixed size**, and are *implemented as* **arrays**.
  - b) **dynamic**, i.e. they **grow in size** as needed, and *implemented as* **linked lists**

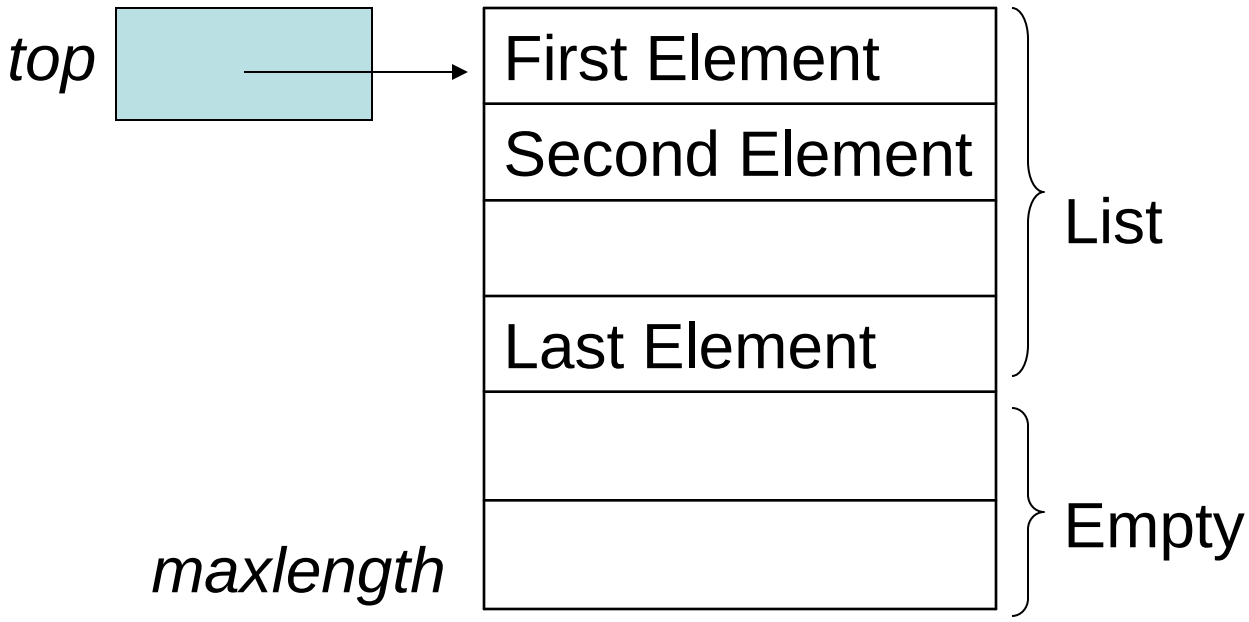
Push and Pop operations of Stack



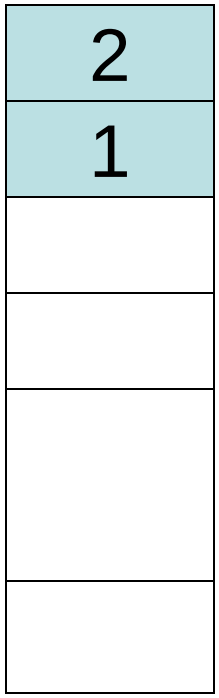
An Array Implementation of Stacks

First Implementation

- Elements are stored in contiguous cells of an array.
- New elements can be inserted to the top of the list.



An Array Implementation of Stacks



Problem with this implementation

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

An Array Implementation of Stacks

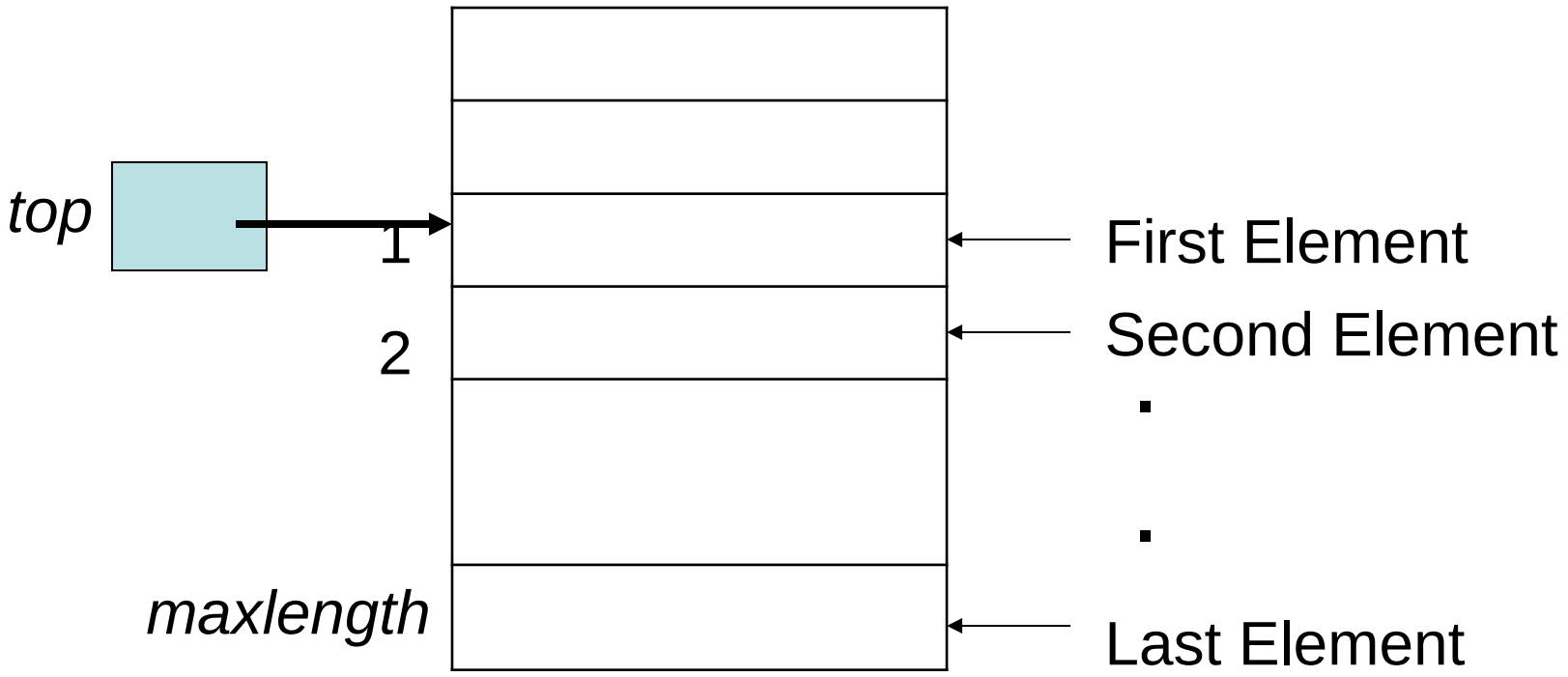
Since, in a stack the insertion and deletion take place only at the top, so…

A better Implementation:

- Anchor the bottom 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.

An Array Implementation of Stacks

A better Implementation:



A Stack Class

```
#ifndef INTSTACK_H
#define INTSTACK_H

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

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

#endif
```

Implementation

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

Push

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

```
// Member function pop pops the value at the top
// of the stack off, and copies it into the variable
// passed as an argument.
void IntStack::pop(int &num)
   {
        if (isEmpty())
                cout << "The stack is empty.\n";
        else
        {
                num = stackArray[top];
                top--;
        }
   }
```

```
//*****************************************************
// Member function isFull returns true if the stack *
// is full, or false otherwise.                      *
//*****************************************************

bool IntStack::isFull(void)
{
      bool status;

      if (top == stackSize - 1)
            status = true;
      else
            status = false;


      return status;

      // return (top == stackSize-1);
}
```

```
//*****************************************
// Member funciton isEmpty returns true if the
//stack *
// is empty, or false otherwise.*
//*****************************************

bool IntStack::isEmpty(void)

{

      bool status;

      if (top == -1)
            status = true;
      else
            status = false;

      return status;

      // return (top == -1);
}
```

```
// This program demonstrates the IntStack class.
#include <iostream.h>
#include "intstack.h“

void main(void)
{
      IntStack stack(5);
      int catchVar;

      cout << "Pushing 5\n";
      stack.push(5);
      cout << "Pushing 10\n";
      stack.push(10);
      cout << "Pushing 15\n";
      stack.push(15);
      cout << "Pushing 20\n";
      stack.push(20);
      cout << "Pushing 25\n";
      stack.push(25);
```

```
        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;

        stack.pop(catchVar);
        cout << catchVar << endl;
}
```

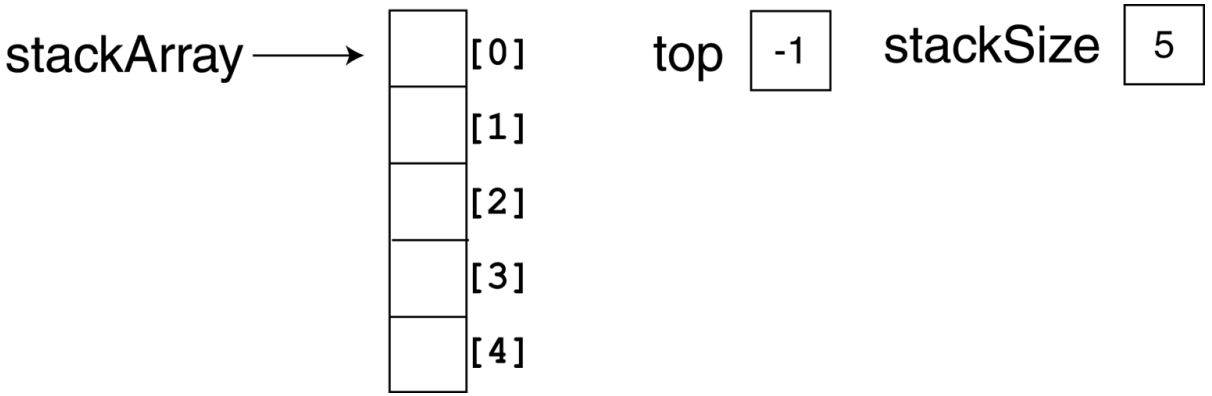
Program Output

```
Pushing 5
Pushing 10
Pushing 15
Pushing 20
Pushing 25
Popping...
25
20
15
10
5
```

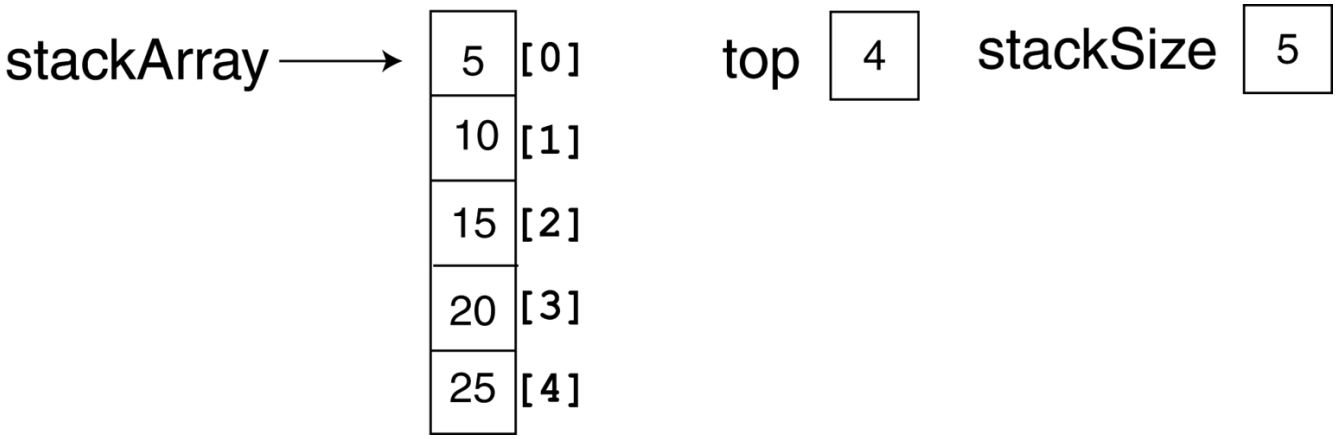
Note the sequence !

About Program 1

- In the program, the constructor is called with the argument 5. This sets up the member variables as shown in Figure 1. Since `top` is set to –1, the stack is empty



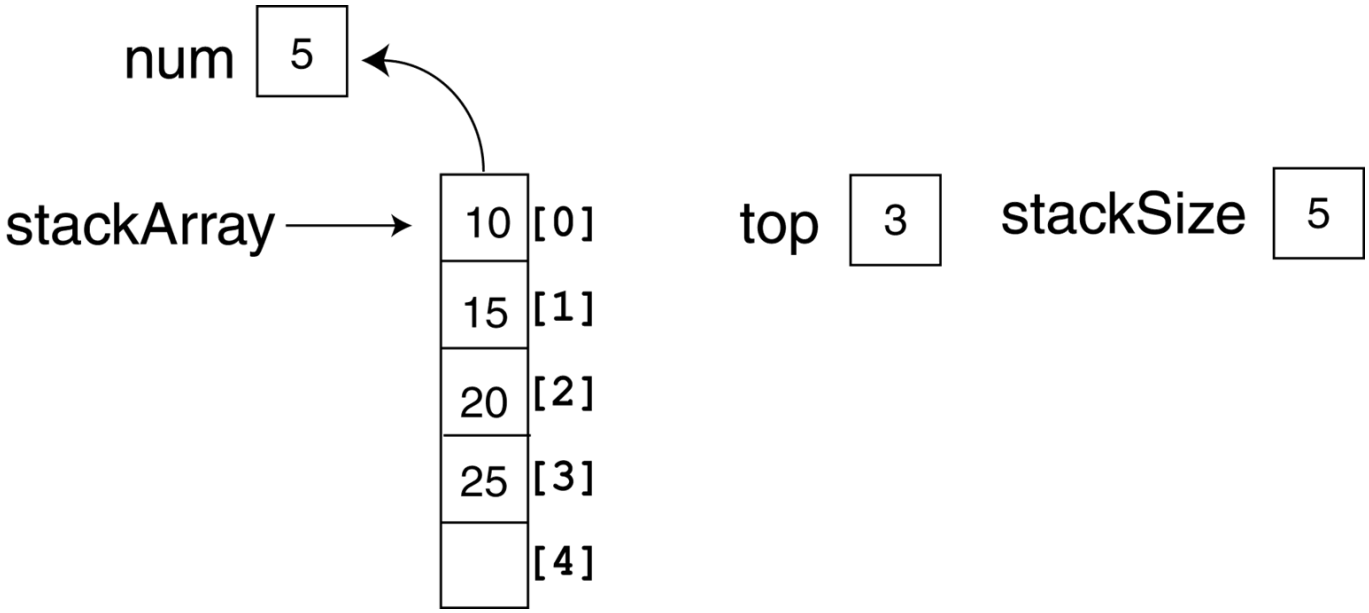
- Figure below shows the state of the member variables after all five calls to the **`push`** function. Now the top of the stack is at element 4, and the stack is full.



Notice that the `pop` function uses a reference parameter, `num`.

The value that is popped off the stack is copied into `num` so it can be used later in the program.

- Figure depicts the state of the class members, and the `num` parameter, just after the first value is popped off the stack.

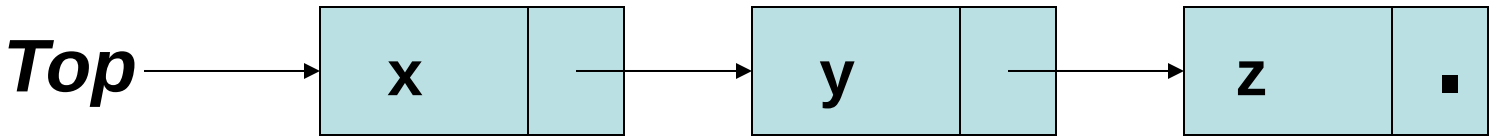


Stack Templates

The stack class so far work with integers only. A stack template can be used to work with any data type.

A Linked-List Implementation of Stacks

- Stack can *expand* or *shrink* with each PUSH or POP operation.
- PUSH and POP operate only on the header cell and the first cell on the list.



Linked List Implementation of Stack

```
class Stack
{
    struct node
    {
        int data;
        node *next;
    }*top;
public:
    void Push(int newelement);
    void Pop(int &);
    bool IsEmpty();
};
```

```
void Stack::Push(int newelement)
{
    node *newptr;
    newptr=new node;
    newptr->data=newelement;
    newptr->next=top;
    top=newptr;
}

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

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

Program 3

```
// This program demonstrates the dynamic stack
// class DynIntClass.

#include <iostream.h>
#include "dynintstack.h“

void main(void)
{
      DynIntStack stack;
      int catchVar;


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

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

        cout << "\nAttempting to pop again... ";
        stack.pop(catchVar);
}
```

Program Output

```
Pushing 5
Pushing 10
Pushing 15
Popping...
15
10
5

Attempting to pop again... The stack is empty.
```

APPLICATIONS OF STACKS

Algebraic Expression

- An algebraic expression is a legal combination of operands and the operators.
  - Operand is the quantity on which a mathematical operation is performed.
  - Operator is a symbol which signifies a mathematical or logical operation.

Infix, Postfix and Prefix Expressions

- **INFIX:** expressions in which operands surround the operator.
- **POSTFIX**: operator comes after the operands, also Known as Reverse Polish Notation (RPN).
- **PREFIX**: operator comes before the operands, also Known as Polish notation.
- Example
  - Infix: A+B-C Postfix: AB+C- Prefix: -+ABC

Examples of infix to prefix and postfix

Infix	PostFix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	?	?

A+B*C in postfix

- Applying the rules of precedence, we obtained

A+B*C

A+(B*C) Parentheses for emphasis

A+(BC*) Convert the multiplication,

ABC*+ Postfix Form

((A+B)*C-(D-E)) $ (F+G)

Conversion to Postfix Expression

((AB+)*C-(DE-)) $ (FG+)

((AB+C*)-(DE-)) $ (FG+)

(AB+C*DE--) $ (FG+)

AB+C*DE- -FG+$

Exercise: Convert the following to Postfix

(A + B) * (C – D)

A $ B * C – D + E / F / (G + H)

Why do we need PREFIX/POSTFIX?

- Appearance may be misleading, INFIX notations are not as simple as they seem
- To evaluate an infix expression we need to consider
  - Operators' Priority
  - Associative property
  - Delimiters

Why do we need PREFIX/POSTFIX?

- Infix Expression Is Hard To Parse and difficult to evaluate.
- Postfix and prefix do not rely on operator priority and are easier to parse.

Why do we need PREFIX/POSTFIX?

- An expression in infix form is thus converted into prefix or postfix form and then evaluated without considering the operators priority and delimiters.

Conversion of Infix Expression to postfix

A+B*C = ABC*+

There must be a precedence function. `prcd(op1, op2), where op1 and op2 are chars representing operators.`

This function returns TRUE if op1 has precedence over op2 when op1 appears to the left of op2 in an infix expression without parenthesis. prcd(op1,op2) returns FALSE otherwise.

`prcd('*','+') and prcd('+','+') are TRUE whereas prcd('+','*') is FALSE.`

Algorithm to Convert Infix to Postfix

```
opstk = the empty stack;
while (not end of input) {
    symb = next input character;
    if (symb is an operand)
        add symb to the postfix string
    else {
        while (!empty(opstk) &&
    prcd(stacktop(opstk),symb) ) {
            topsymb = pop(opstk);
            add topsymb to the postfix
    string;
        } /* end while */
        push(opstk, symb);
    } /* end else */
} /* end while */
/* output any remaining operators */
while (!empty(opstk) ) {
    topsymb = pop(opstk);
    add topsymb to the postfix string;
} /* end while */
```

Example-1: A+B*C

symb	Postfix string	opstk
A	**A**	
+	**A**	**+**
B	**AB**	**+**
*****	**AB**	**+ ***
C	**ABC**	**+ ***
	ABC*	**+**
	ABC*+	

Algorithm to Convert Infix to Postfix

```
opstk = the empty stack;
while (not end of input) {
    symb = next input character;
    if (symb is an operand)
        add symb to the postfix string
    else {
        while (!empty(opstk) &&
    prcd(stacktop(opstk),symb) ) {
            topsymb = pop(opstk);
            add topsymb to the postfix
    string;
        } /* end while */
        push(opstk, symb);
    } /* end else */
} /* end while */
/* output any remaining operators */
while (!empty(opstk) ) {
    topsymb = pop(opstk);
    add topsymb to the postfix string;
} /* end while */
```

Example-1: A*B+C

symb	Postfix string	opstk
A	**A**	
*****	**A**	*****
B	**AB**	*****
+	**AB***	**+**
C	**AB*C**	**+**
	AB*C+	

Questions?