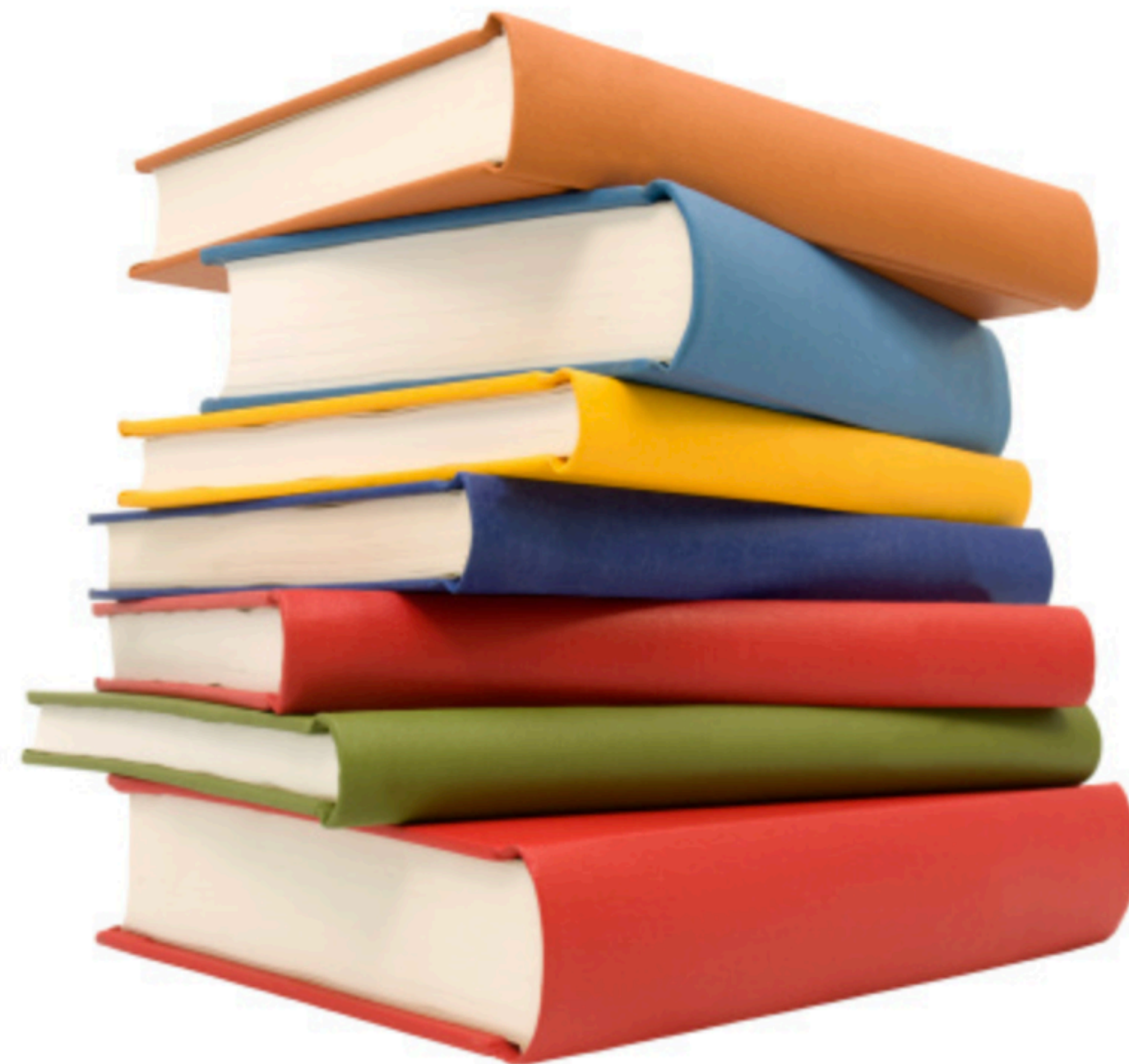


COSC 2436: Stacks

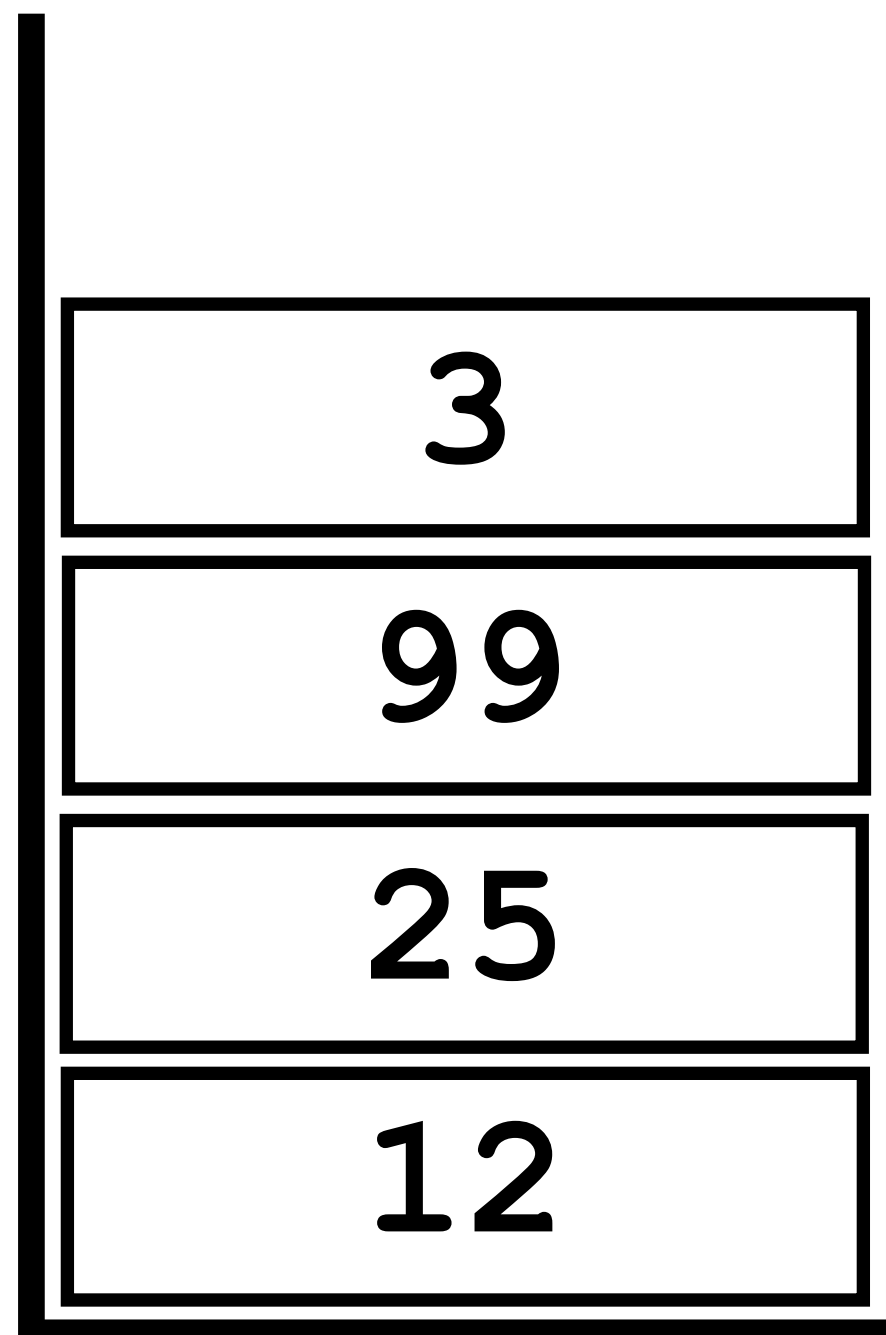
What is a stack?

A stack is a linear data structure where elements are inserted and removed in a last-in-first-out (LIFO) order.



How can we implement a stack?

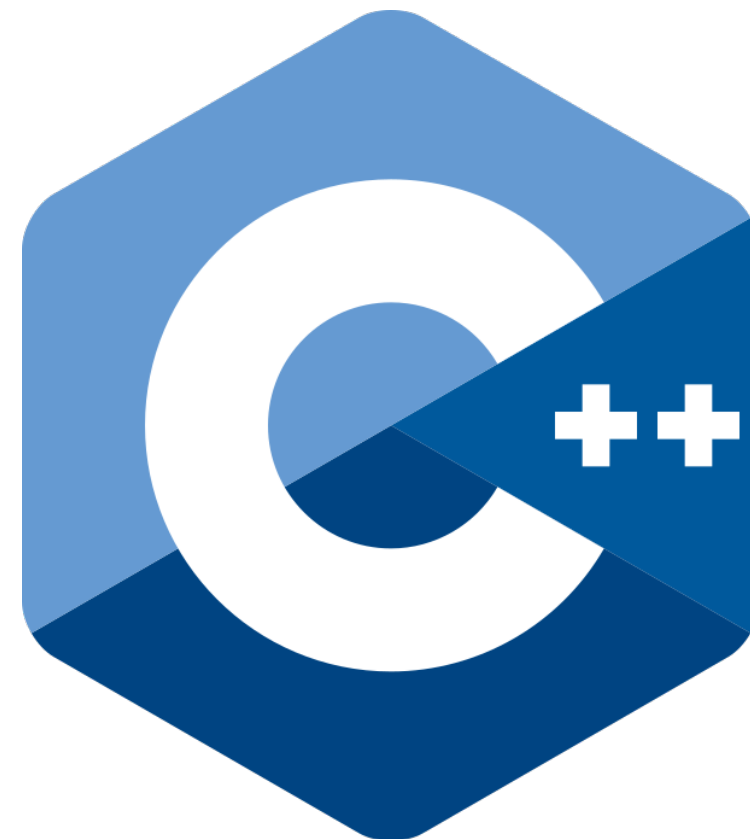
- **Linked List**
- **Array**
- **C++ stack container**



Stack Operations

- **push** - insert element at top of stack
- **pop** - remove element from top of stack
- **top** - return the element at top of stack
- **empty** - returns true if stack is empty, else returns false

All of these operations have a time complexity of $O(1)$



What is Infix and Postfix?

- Infix places operators between operands

$$2 + 6 \times 3$$

- Postfix places operators after operands

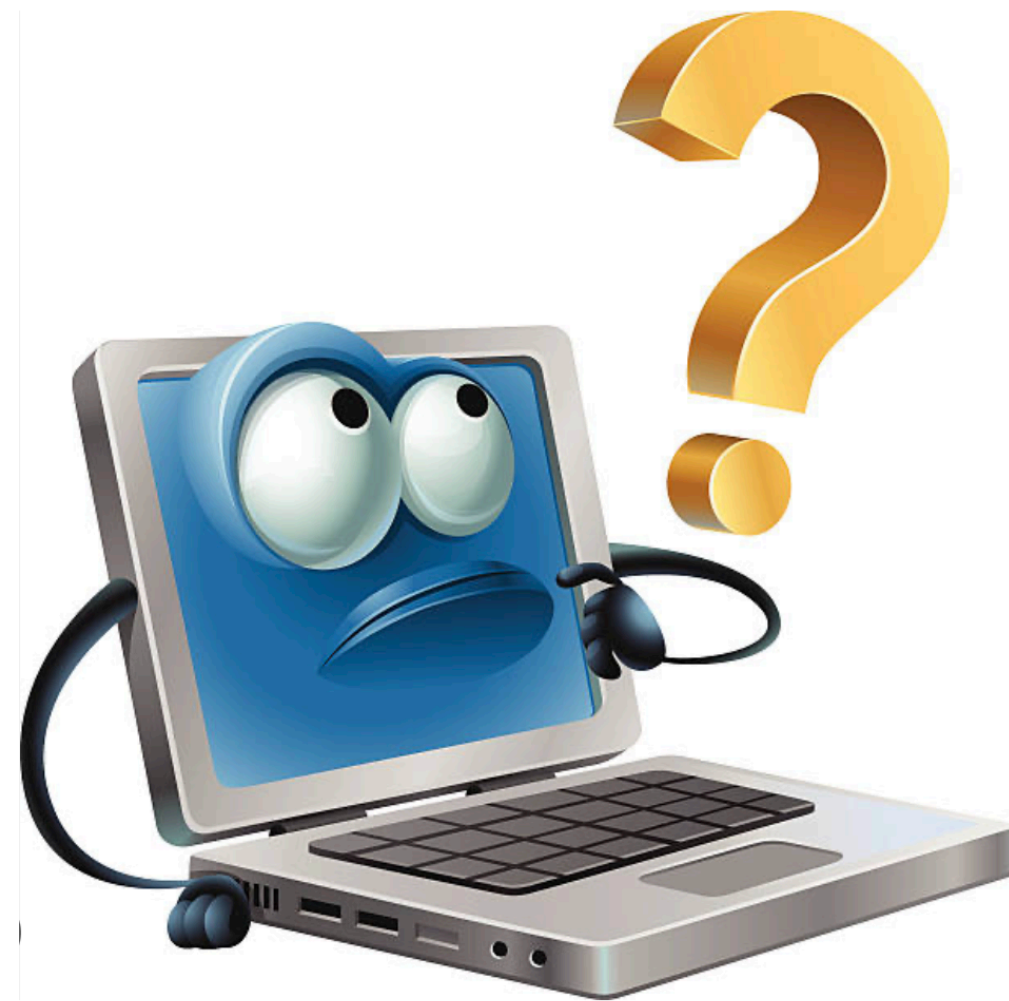
$$2\ 6\ 3\ \times\ +$$

- Prefix places operators before operands

$$+ 2\ \times\ 6\ 3$$

What does this have to do with stacks?

- Infix relies on parentheses for the order of operations
- Postfix & Prefix does not need parentheses
- Infix is intuitive for humans but not for computers
- Postfix & Prefix expressions can be evaluated using stacks



Infix to Postfix

```
string infixToPostfix(string exp) {
    stack<char> s;
    string str;
    for(int i = 0; i < exp.length(); i++) {
        if(isdigit(exp[i]))
            str += exp[i];
        else if(exp[i] == '(')
            s.push('(');
        else if(exp[i] == ')') {
            while(s.top() != '(')
                str += s.top(); s.pop();
            s.pop();
        }
        else {
            while(!s.empty() && priority(exp[i]) <= priority(s.top()))
                str += s.top(); s.pop();
            s.push(exp[i]);
        }
    }
    while(!s.empty())
        str += s.top(); s.pop();
    return str;
}
```

Postfix to Infix

```
string postfixToInfix(string exp) {  
    stack<string> s;  
    string str1, str2, str3;  
    for(int i = 0; i < exp.length(); i++) {  
        if(isdigit(exp[i]))  
            s.push(exp.substr(i,1));  
        else{  
            str1 = s.top(); s.pop();  
            str2 = s.top(); s.pop();  
            str3 += '(' + str2 + exp[i] + str1 + ')';  
            s.push(str3);  
        }  
    }  
    return s.top();  
}
```


Infix to Prefix

```
string infixToPrefix(string exp) {  
    reverse(exp.begin(), exp.end());  
    for(int i = 0; i < exp.length(); i++) {  
        if(exp[i] == '(')  
            exp[i] = ')';  
        else if(exp[i] == ')')  
            exp[i] = '(';  
    }  
    string prefix = infixToPostfix(exp);  
    reverse(prefix.begin(), prefix.end());  
    return prefix;  
}
```

Evaluate Prefix

```
int evaluatePrefix(string exp) {
    stack<int> s;
    for(int i = exp.length() - 1; i >= 0; i--){
        if(isdigit(exp[i]))
            s.push(exp[i]-48);
        else{
            int val1 = s.top(); s.pop();
            int val2 = s.top(); s.pop();
            switch(exp[i]){
                case '+': s.push(val1 + val2); break;
                case '-': s.push(val1 - val2); break;
                case '*': s.push(val1 * val2); break;
                case '/': s.push(val1 / val2); break;
            }
        }
    }
    return s.top();
}
```

Evaluate Postfix

```
int evaluatePostfix(string exp) {
    stack<int> s;
    for(int i = 0; i < exp.length(); i++) {
        if(isdigit(exp[i])
            s.push(exp[i] - 48);
        else{
            int val1 = s.top(); s.pop();
            int val2 = s.top(); s.pop();
            switch(exp[i]){
                case '+': s.push(val2 + val1); break;
                case '-': s.push(val2 - val1); break;
                case '*': s.push(val2 * val1); break;
                case '/': s.push(val2 / val1); break;
            }
        }
    }
    return s.top();
}
```

Stack Practice: Valid Parentheses

Given a string `exp` containing the characters `(,), {, }, [, and]`, determine if the input string is valid. An input string is valid if:

- Open brackets must be closed by same bracket type
- Open brackets must be closed in the correct order
- Every close bracket has a corresponding open bracket of the same type

Input: `{ [()] }` **=>** **Output:** True

Input: `(} [))` **=>** **Output:** False

Stacks Practice: Valid Parentheses

```
bool validParentheses(string exp){
    stack<char> s;
    for(int i = 0; i < exp.length(); i++){
        if(exp[i] == '(' || exp[i] == '[' || exp[i] == '{')
            s.push(exp[i]);
        else if(exp[i] == ')'){
            if(s.empty() || s.top() != '(')
                return false;
            s.pop();
        }
        else if(exp[i] == ']'){
            if(s.empty() || s.top() != '[')
                return false;
            s.pop();
        }
        else if(exp[i] == '}'){
            if(s.empty() || s.top() != '{')
                return false;
            s.pop();
        }
    }
    return s.empty();
}
```