ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

Module 5

Stacks

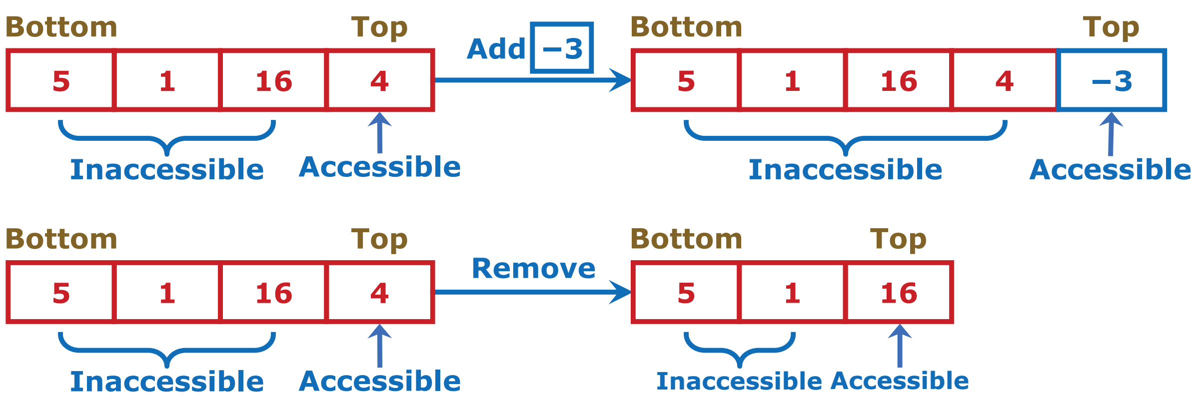
Dayu Wang

- **Stacks**

→ **Stacks** are abstract data structures **(ADT)** with the property that **only the top element is accessible**.

- Each time, you add a new element **onto the top** of the stack.
- Only the **top element** is accessible.
- Each time, you can only remove an element **from the top** of the stack.

→ Stacks are based on the **LIFO (Last-In-First-Out)** principle.



- **Class-Member Functions in Stack Class**

→ **Theoretically, stacks only support 7 functions.**

Functions	Behavior
`size_t size() const;`	Returns the number of elements stored in the stack.
`bool empty() const;`	Tests whether the stack is empty.
`T& top();`	Returns the top element in the stack (l-value).
`const T& top() const;`	Returns the top element in the stack (r-value).
`void push(const T&);`	Adds a new element onto the top of the stack.
`void pop();`	Removes the top element from the stack.
`void clear();`	Removes all elements from the stack.

- **Using vector to implement stack**

→ **The top of the stack is the rear end of the vector.**

- `.size()`, `.empty()`, **and** `clear()` **functions are the same as vector.**
- `.top()` ➡ `.back()`
- `.pop()` ➡ `.pop_back()`
- `.push(item)` ➡ `.push_back(item)`

[See sample code]

- **Using linked list to implement stack**

→ **Singly-linked list or doubly-linked list?**

Singly-linked list

→ **In class data fields, we need to keep reference of top node only or top and bottom nodes?**

Top only

[See sample code]

- **Discussion**

→ **What are the applications of stacks?**

Undo/redo system

Back/forward system

Code compilation

→ **Graph traversal**

Depth-First Search (DFS)

Stack is an intermediate data structure in DFS.

Breadth-First Search (BFS)

Queue is an intermediate data structure in BFS.

→ **Stacks are useful to solve algorithm problems.**

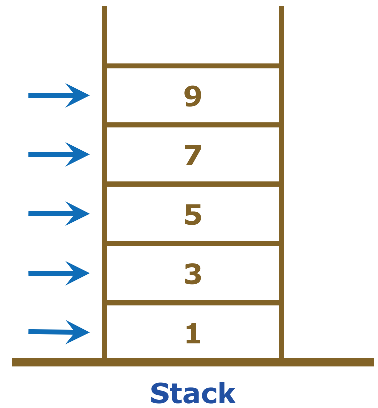
- **[Example 1] Reversing elements in a linear container**

→ **Using a stack to reverse a vector of integers**

Vector | 1 | 3 | 5 | 7 | 9 |

1) **Create an empty stack.**
2) **Push each element in the vector onto the stack.**
3) **Pop each element from the stack to form the reversed vector.**

Reversed Vector | 9 | 7 | 5 | 3 | 1 |




```
/** Reverses a vector of integers.
    @param vec: vector of integers to reverse
*/
void reverse(vector<int>& vec) {
    List_Stack<int> stk;
    for (size_t i = 0; i < vec.size(); i++) { stk.push(vec.at(i)); }
    for (size_t j = 0; j < vec.size(); j++) {
        vec[j] = stk.top();
        stk.pop();
    }
}  // Time complexity: O(n)
```


- **[Example 2] Finding palindromes**

→ **A palindrome is a string that reads the same in either direction—left to right or right to left.**

e.g., "I saw I was I"

→ **Using a stack to test whether a string is palindromic**

1) **Create an empty stack.**
2) **Push the characters in the string onto the stack.**
3) **Construct a string using the popped characters from the stack.**
4) **Compare the constructed string against the original string.**

```
/** Tests whether a string is a palindrome.
    @param s: string to test
    @return: {true} if the string is palindromic; {false} otherwise
*/
bool is_palindromic(const string& s) {
    List_Stack<char> stk;
    for (string::const_iterator it = s.begin(); it != s.end(); it++) { stk.push(*it); }
    string reversed;
    while (!stk.empty()) {
        reversed.push_back(stk.top());
        stk.pop();
    }
    return reversed == s;
}  // Time complexity: O(n)
```

• **[Example 3] Testing balanced parentheses in expressions**

```
c[d]        // Correct
a{b[c]d}e  // Correct
a{b(c]d}e  // Incorrect (']' does not match '('.)
a{b[c}d]e  // Incorrect (order of '}' and ']' is incorrect.)
```

→ **General idea**

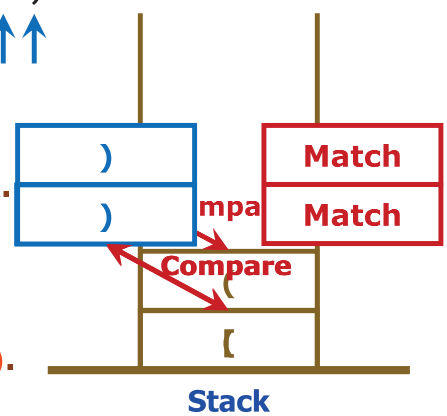
1) **Parse the expression and push the opening delimiters ('(', '[', and '{') onto the stack.**

2) **When a closing delimiter (')', ']', and '}') is found, compare the top of the stack with the closing delimiter and pop it from the stack.**

[Example] "[(3 + x) / (2 - z)] * (y + 3)"

→ **Cases that parentheses are not ~~balanced~~:**

- **Current closing delimiter does not match the opening delimiter at the top of the stack.**
- **Stack is empty when a closing delimiter is found (e.g., "3 + x) + 2").**
- **Stack is non-empty when the expression is completely parsed (e.g., "[(3 + x) * 2 - 5").**




```
/** Tests whether parentheses are balanced in an expression.
    @param exp: expression to test
    @return: {true} if parentheses are balanced; {false} otherwise
*/
bool is_balanced(const string& exp) {
    List_Stack<char> stk;
    for (string::const_iterator it = exp.begin(); it != exp.end(); it++) {
        if (*it == '(' || *it == '[' || *it == '{') { stk.push(*it); }
        if (*it == ')' || *it == ']' || *it == '}') {
            if (stk.empty()) { return false; }
            if (*it == ')' && stk.top() != '(') { return false; }
            if (*it == ']' && stk.top() != '[') { return false; }
            if (*it == '}' && stk.top() != '{') { return false; }
            stk.pop();
        }
    }
    return stk.empty();
}  // Time complexity: O(n)
```

- **Arithmetic Expressions**

→ **Arithmetic expressions consist of operands and operators.**

→ **3 types of arithmetic expressions**

Prefix Expression	Infix Expression	Postfix Expression
*c+ab	c*(a+b)	ab+c*

→ **Evaluating arithmetic expressions**

Prefix Expression	Infix Expression	Postfix Expression	Value
* 4 7	4 * 7	4 7 *	28
* 4 + 7 2	4 * (7 + 2)	4 7 2 + *	36
- * 4 7 20	4 * 7 - 20	4 7 * 20 -	8
+ 3 / * 4 7 2	3 + 4 * 7 / 2	3 4 7 * 2 / +	17

→ **Advantages of using postfix expressions instead of ~~infix expressions~~:**

- **There are no ~~parentheses~~ in postfix expressions.**
- **User does not need to use ~~precedence rules~~ to evaluate postfix expressions.**

- **[Example 4] Evaluating postfix expressions**

→ **Algorithm**

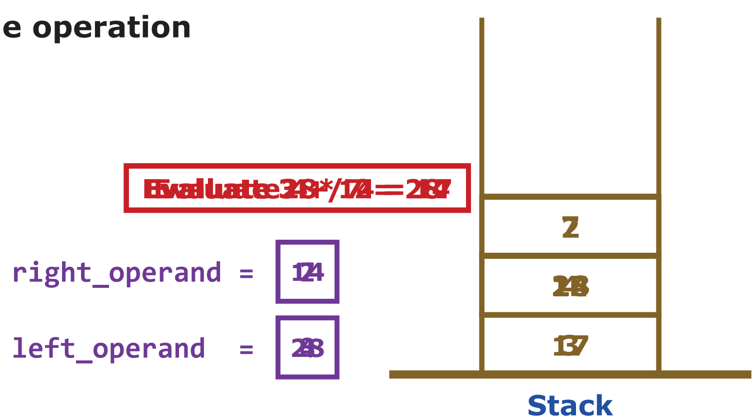
```
stack ← Ø
while there are more tokens
   token ← Get next token
   if token.is_operand()
      stack.push(token)
   else  // "token" is an operator.
      right_operand ← stack.pop()
      left_operand ← stack.pop()
      result ← Evaluate the operation
      stack.push(result)
   endif
endwhile
return stack.top()
```

[Example]

"3 4 7 * 2 / +"

↑↑↑↑↑↑↑

Evaluation result: 17



```cpp
/** Evaluates a postfix expression.
    @param postfix: postfix expression to evaluate
    @return: evaluation result
    @throws exception: divide-by-zero
*/
int eval_postfix(const string& postfix) {
    istringstream iss(postfix);
    List_Stack<int> stk;
    string token;  // Current token
    while (iss >> token) {
        if (isdigit(token.front())) { stk.push(stoi(token)); }
        else {
            int right = stk.top();
            stk.pop();
            int left = stk.top();
            stk.pop();

            // Supported operators
            if (token == "+") { stk.push(left + right); }
            if (token == "-") { stk.push(left - right); }
            if (token == "*") { stk.push(left * right); }
            if (token == "/") {
                if (!right) { throw exception("Divide by zero"); }
                stk.push(left / right);
            }
        }
    }
    return stk.top();
}  // Time complexity: O(n)
```


- **[Example 5] Converting from infix expression to postfix expression**

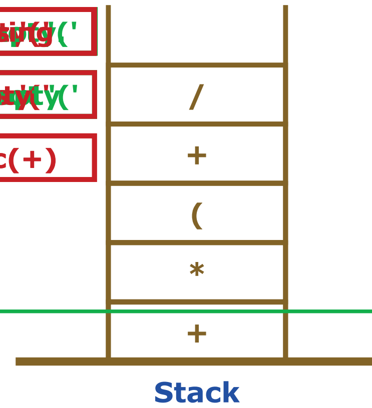
➔ **Algorithm**

```
while there are more tokens in infix expression
   token ← Get next token
   if token.is_operand()
      Append token to postfix expression
   else if token == '('
      Push token onto the stack
   else if token.is_operator()
      while !stack.empty() and stack.top() != '('
            and precedence(token) ≥ precedence(stack.top())
         Pop top off stack and append to postfix expression
      endwhile
      Push token onto the stack
   else  // "token" == ')'
      while stack.top() != '('
         Pop top off stack and append to postfix expression
      endwhile
      stack.pop()  // Remove the opening parenthesis from the stack.
   endif
endwhile
while !stack.empty()
   Append stack.top() to postfix expression
   stack.pop()
endwhile
```

- **[Example]** "a + b * (c + d / e)"

Postfix: a b c d e / + * +

Prec(*) > Prec(+)



→ **In the infix-to-postfix process, precedence rules must be used.**

The `precedence()` **function**

```
/** Returns the precedence of an operator.
    @param oper: operator to return its precedence
    @return: precedence
    @throws exception: operator unsupported
*/
int precedence(const string& oper) {
    if (oper == "*" || oper == "/") { return 5; }
    if (oper == "+" || oper == "-") { return 6; }
    throw exception("Unsupported operator");
}  // Time complexity: O(1)
```

→ **In order to correctly parse the infix expression, all tokens must be surrounded by spaces on the left and on the right.**

" (3 + 2) * 5 " **is correct.**

"(3 + 2) * 5" **is incorrect.**

→ **All the sample code assumes that the input arithmetic expression is valid, without ~~errors~~.**


```
/** Converts an infix expression to postfix expression.
    @param infix_exp: infix expression to convert
    @return: postfix expression converted from the infix expression
*/
string infix_to_postfix(const string& infix_exp) {
    istringstream iss(infix_exp);
    ostringstream oss;
    List_Stack<string> stk;
    string token;
    while (iss >> token) {
        if (isdigit(token.front())) { oss << ' ' << token; }
        else if (token == "(") { stk.push(token); }
        else if (token == ")") {
            while (stk.top() != "(") {
                oss << ' ' << stk.top();
                stk.pop();
            }
            stk.pop();
        } else {
            while (!stk.empty() && stk.top() != "(" && precedence(token) >= precedence(stk.top())) {
                oss << ' ' << stk.top();
                stk.pop();
            }
            stk.push(token);
        }
    }
    while (!stk.empty()) {
        oss << ' ' << stk.top();
        stk.pop();
    }
    return oss.str();
}  // Time complexity: O(n)
```