Stack

This C++ file is a complete and practical tutorial on the **Stack data structure**. It's brilliantly structured to build your understanding from the ground up, starting with the fundamental implementations and then moving on to showcase powerful, real-world applications. 🏗️

The file begins by demonstrating three distinct ways to build a stack:

1. **Using an Array:** A simple, fixed-size implementation that's fast but limited in capacity.
2. **Using a Linked List (C-style):** A flexible, dynamic-size implementation using procedural C code.
3. **Using a C++ Class:** A modern, object-oriented approach that encapsulates the linked list logic for better safety and resource management.

After establishing these foundations, the file dives into classic computer science problems where the Stack's **Last-In, First-Out (LIFO)** nature is essential. It provides clear, commented code for:

* **Parenthesis Matching:** Validating mathematical expressions.
* **Infix to Postfix Conversion:** Translating human-readable expressions into a format that's easier for computers to evaluate.
* **Postfix Evaluation:** Calculating the final result of a postfix expression.

This logical progression from theory to application makes the file an outstanding resource for mastering the Stack data structure and its uses in programming and algorithm design.

**Stack using Arrays**

This section implements a **Stack** using a basic array. It's a static approach where the size of the stack is fixed at the time of creation. It's very fast but can lead to a "Stack Overflow" if you try to add more elements than its capacity.

* struct Stack{ int size; int top; int \*S; }; This struct defines the Stack. It contains the maximum **size** (capacity), an integer top that acts as an index to the top element, and a pointer S to the array in heap memory.
* st->top = -1; The top is initialized to **-1**. This is a standard convention indicating that the stack is **empty**. The first element pushed will be at index 0.
* if(st->top == st->size-1){ printf("Stack Overflow\n"); } The push operation first checks for **Stack Overflow**. This occurs when top has reached the last valid index of the array (size-1), meaning there's no more room.
* x = st.S[st.top]; st.top--; The pop operation retrieves the element at the top index and then simply **decrements the top index**. The data isn't actually erased; it's just no longer accessible.
* x = st.S[st.top-index+1]; The peek function lets you look at an element at a certain position from the top without removing it. This formula correctly converts the index (e.g., 1st from top, 2nd from top) into the correct 0-based array index.

**Stack using Linked List**

This section implements a **Stack** using a dynamic Linked List. This approach is more flexible than the array version because the stack can grow or shrink as needed, so there's no risk of overflow unless the system runs out of memory.

* struct Node{ ... }\*top=NULL; Instead of an array, we use a global pointer top that always points to the **head of the linked list**. An empty stack is represented by top being NULL.
* t->data = x; t->next = top; top = t; This is the core of the push operation. A new node t is created. Its next pointer is set to the current top, and then the global top pointer is updated to point to the new node. This is effectively an "insert at beginning" operation, which is very fast (O(1)).
* t = top; top = top->next; x = t->data; free(t); The pop operation is an "delete from beginning" operation. It saves the current top node, moves top to the next node in the list, and then frees the memory of the original top node.

**Stack using Linked List in C++**

This section refactors the C-style linked list stack into a modern **C++ class**. This is a much better design because it encapsulates the data and logic, preventing direct, unsafe access to the top pointer.

* class Stack{ private: Node \*top; public: ... }; The Stack class keeps the top pointer **private**, meaning only the class's own methods (push, pop, display) can modify it. This is a key principle of Object-Oriented Programming called **encapsulation**.
* Stack(){top = NULL;} This is the **constructor**. It's a special function that automatically runs when a Stack object is created, ensuring that top is always initialized to NULL for a new, empty stack.
* void Stack::push(int x){ ... } The push method is part of the Stack class (as indicated by Stack::). It contains the exact same logic as the C-style version but operates on the private top member of the class instance.

**Parenthesis Matching**

This is a classic application of a stack to check if an expression has **balanced parentheses**. The LIFO (Last-In, First-Out) property of a stack is perfect for matching opening brackets with their corresponding closing ones. ✅

* if(str[i]=='('){ push(str[i]); } The logic is simple: when the code scans the string and finds an **opening parenthesis (**, it pushes it onto the stack.
* else if(str[i]==')'){ if(top==NULL){ return 0; } pop(); } When a **closing parenthesis )** is found, we try to pop from the stack.
  + If the stack is empty (top==NULL), it means we found a closing bracket with no matching opener, so the expression is **unbalanced**.
  + If we can successfully pop, it means we've found a matching pair, and that pair is now resolved.
* if(top==NULL){ return 1; } After the entire string has been scanned, the stack must be **empty** for the expression to be balanced. If any parentheses are left on the stack, it means there are unmatched opening brackets.

**Infix to Postfix conversion**

This algorithm uses a stack to convert a standard mathematical expression (Infix) into a Postfix (or Reverse Polish Notation) expression. Postfix is easier for computers to evaluate because it removes ambiguity and the need for parentheses.

* int precedence(char x){ ... } This helper function is crucial. It assigns a **priority level** to operators. Multiplication \* and division / have a higher precedence (2) than addition + and subtraction - (1).
* if (isOperand(infix[i])) { postfix[j++] = infix[i++]; } If the character is an **operand** (a number or letter like 'a', 'b', 'c'), it is immediately appended to the output postfix string.
* while (top != NULL && precedence(top->data) >= precedence(infix[i])) { postfix[j++] = pop(); } This is the key logic. When an **operator** is found, the code looks at the operator on top of the stack. It pops all operators from the stack that have a **higher or equal precedence** than the current one and appends them to the postfix string.
* push(infix[i++]); After popping any higher-precedence operators, the current, lower-precedence operator is pushed onto the stack to wait its turn.

**Infix to Postfix using STLS (actually Postfix Evaluation)**

*(The heading in the file is a bit misleading; this code doesn't convert to postfix. It* ***evaluates*** *an existing postfix expression.)*

This section shows how to calculate the final numerical result of a postfix expression. It provides two versions: one using the custom C++ Stack class and another using the built-in C++ Standard Template Library (std::stack), which is the common practice in modern C++.

* stk.push(postfix[i]-'0'); The algorithm scans the postfix string. When it finds an **operand** (a digit character), it converts that character to its integer equivalent by subtracting the ASCII value of '0' and pushes the integer onto the stack.
* y = stk.top(); stk.pop(); x = stk.top(); stk.pop(); When it finds an **operator** (+, -, \*, /), it pops the top **two operands** from the stack. Note that the second operand (y) is popped first.
* result = operation(postfix[i], x, y); stk.push(result); The operation is performed on the two popped numbers (x and y), and the **result is pushed back** onto the stack.
* result = stk.top(); After the entire expression is processed, the stack will contain only one number: the final result.

**Evaluation of Postfix**

This is the final, active code block in your file. It's another implementation of the **Postfix Evaluation** algorithm, this time using the C-style linked list stack. The logic is identical to the previous section.

* for (i = 0; postfix[i] != '\0'; i++) { ... } The function iterates through the entire postfix expression character by character.
* if (isOperand(postfix[i])) { push(postfix[i] - '0'); } If the character is a digit (operand), it's converted to an integer and **pushed** onto the stack. For example, the character '4' becomes the integer 4.
* else { x2 = pop(); x1 = pop(); ... } If the character is an operator, the top two numbers are **popped** from the stack. The first pop gives x2 (the right-hand side of the operation), and the second pop gives x1 (the left-hand side).
* switch(postfix[i]) { ... } push(r); A switch statement efficiently determines which mathematical operation to perform. The result of the operation (r) is then **pushed back** onto the stack, ready to be used in the next calculation.
* return top->data; After the loop finishes, the final answer is the only element remaining on the stack, which is returned as the result.