Stack Cheatsheet

A stack is a linear data structure that stores multiple elements in a specific order, following the **LIFO (Last In, First Out)** principle. This means the last element added to the stack is the first one to be removed. Think of a stack of plates: you add and remove plates from the top.

Unlike vectors, elements in a stack are not accessed by index numbers. You can only access the element at the top of the stack.

To use a stack in C++, you need to include the `<stack>` header file:

```
#include <stack>
```

Creating a Stack

To create a stack, use the `stack` keyword, specify the data type it will store within angle brackets (`<>`), and then provide the stack's name. For example:

```
stack<string> cars;
```

Note: You cannot add elements to the stack at the time of declaration, unlike vectors.

Basic Stack Operations

1. `push()` : Adding Elements

To add elements to the stack, use the `.push()` function. Elements are always added to the top of the stack.

```
stack<string> cars;

cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");
```

After these operations, the stack would conceptually look like this (top element is the last one pushed):

`Mazda (top element) <- Ford <- BMW <- Volvo`

2. `top()`: Accessing the Top Element

You can only access the top element of the stack using the `.top()` function.

```
// Access the top element
cout << cars.top(); // Outputs "Mazda"
```

You can also use `.top()` to change the value of the top element:

```
cars.top() = "Tesla";
cout << cars.top(); // Now outputs "Tesla"
```

3. `pop()`: Removing Elements

To remove the top element from the stack, use the `.pop()` function. This function removes the last element that was added.

```
stack<string> cars;
cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");

cars.pop(); // Removes "Mazda"

cout << cars.top(); // Now outputs "Ford"
```

4. `size()`: Getting the Stack Size

To find out how many elements are in the stack, use the `.size()` function:

```
cout << cars.size(); // Outputs 3 (after the pop operation above)
```

5. `empty()`: Checking if the Stack is Empty

The `.empty()` function returns `true` (1) if the stack is empty and `false` (0) otherwise.

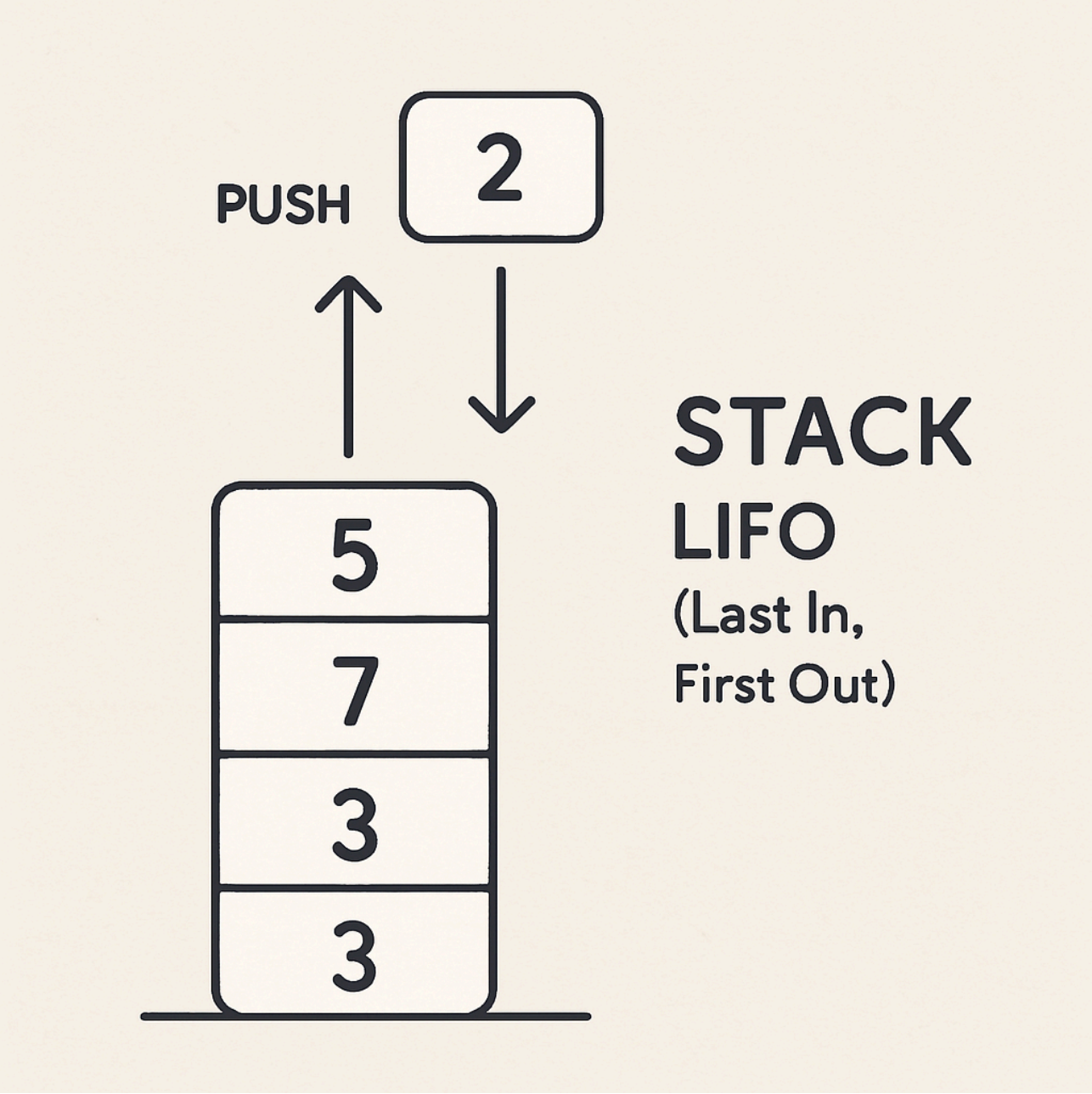
```
 stack<string> emptyCars;
cout << emptyCars.empty(); // Outputs 1 (true)

stack<string> cars;
cars.push("Volvo");
cout << cars.empty(); // Outputs 0 (false)
```

Common Use Cases for Stacks

Stacks are widely used in computer science for various applications, including:

- **Function Call Management:** Compilers use stacks to manage function calls and local variables.
- **Expression Evaluation:** Used in evaluating arithmetic expressions (e.g., infix to postfix conversion).
- **Undo/Redo Functionality:** Many applications implement undo/redo features using stacks.
- **Backtracking Algorithms:** Essential for algorithms that involve exploring multiple paths and backtracking (e.g., maze solving, depth-first search).
- **Browser History:** Storing the history of visited web pages.

This cheatsheet provides a fundamental understanding of the stack data structure and its basic operations in C++. For more advanced concepts and implementations, refer to comprehensive data structure resources.



Stack Implementations

Stacks can be implemented using various underlying data structures. The most common implementations are using arrays (static or dynamic) or linked lists.

1. Implementation using Dynamic Array (e.g., `std::vector`)

A dynamic array provides a contiguous block of memory that can grow or shrink as needed. In C++, `std::vector` is an excellent choice for implementing a stack.

Advantages: * **Cache Locality:** Elements are stored contiguously, leading to better cache performance. * **Simple Implementation:** `push_back()` and `pop_back()` operations are efficient and straightforward. * **Memory Efficiency:** Generally uses less memory overhead per element compared to linked lists.

Disadvantages: * **Reallocation Overhead:** When the array capacity is exceeded, a new, larger array must be allocated, and all elements copied, which can be an expensive operation (though amortized constant time for `push_back`). * **Fixed Capacity (if not dynamic):** If using a static array, the size is fixed, leading to potential overflow or underutilization.

C++ Example (using `std::vector` as underlying storage):

```cpp
#include <iostream>
#include <vector>
#include <stdexcept> // For std::out_of_range

template <typename T>
class MyStackArray {
private:
    std::vector<T> data;

public:
    // Push operation: adds an element to the top
    void push(const T& value) {
        data.push_back(value);
        std::cout << value << " pushed to stack (Array-based)\n";
    }

    // Pop operation: removes the top element
    T pop() {
        if (isEmpty()) {
            throw std::out_of_range("Stack is empty, cannot pop (Array-
based)");
        }
        T value = data.back();
        data.pop_back();
        std::cout << value << " popped from stack (Array-based)\n";
        return value;
    }

    // Top operation: returns the top element without removing it
    T top() const {
        if (isEmpty()) {
            throw std::out_of_range("Stack is empty, no top element (Array-
based)");
        }
        return data.back();
    }

    // Check if the stack is empty
    bool isEmpty() const {
        return data.empty();
    }

    // Get the number of elements in the stack
    size_t size() const {
        return data.size();
    }
};

/*
int main() {
    MyStackArray<int> s_array;
    s_array.push(10);
    s_array.push(20);
    s_array.push(30);

    std::cout << "Top element: " << s_array.top() << "\n";
    s_array.pop();
    std::cout << "Size: " << s_array.size() << "\n";
    s_array.pop();
    s_array.pop();
```

```
    try {
        s_array.pop(); // This will throw an exception
    } catch (const std::out_of_range& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
    return 0;
}
*/
```

2. Implementation using Linked List

A linked list-based stack uses nodes, where each node contains data and a pointer to the next node. The `top` of the stack is typically represented by the head of the linked list.

Advantages: * **Dynamic Size:** Easily grows and shrinks without the need for reallocations or fixed capacity limits. * **Efficient Insertions/Deletions:** `push()` and `pop()` operations (at the head) are O(1) as they only involve updating a few pointers.

Disadvantages: * **Memory Overhead:** Each node requires extra memory for pointers, which can be significant for small data types. * **No Random Access:** Accessing elements other than the top requires traversing the list, which is O(n). * **Poor Cache Performance:** Elements are not stored contiguously, leading to more cache misses.

C++ Example (using Linked List):

```cpp
#include <iostream>
#include <stdexcept> // For std::out_of_range

template <typename T>
class Node {
public:
    T data;
    Node* next;

    Node(T d) : data(d), next(nullptr) {}
};

template <typename T>
class MyStackLinkedList {
private:
    Node<T>* top_node;
    size_t current_size;

public:
    MyStackLinkedList() : top_node(nullptr), current_size(0) {}

    // Destructor to free memory
    ~MyStackLinkedList() {
        while (top_node != nullptr) {
            Node<T>* temp = top_node;
            top_node = top_node->next;
            delete temp;
        }
    }

    // Push operation: adds an element to the top
    void push(const T& value) {
        Node<T>* newNode = new Node<T>(value);
        newNode->next = top_node;
        top_node = newNode;
        current_size++;
        std::cout << value << " pushed to stack (Linked List-based)\n";
    }

    // Pop operation: removes the top element
    T pop() {
        if (isEmpty()) {
            throw std::out_of_range("Stack is empty, cannot pop (Linked List-
based)");
        }
        Node<T>* temp = top_node;
        T value = temp->data;
        top_node = top_node->next;
        delete temp;
        current_size--;
        std::cout << value << " popped from stack (Linked List-based)\n";
        return value;
    }

    // Top operation: returns the top element without removing it
    T top() const {
        if (isEmpty()) {
            throw std::out_of_range("Stack is empty, no top element (Linked
List-based)");
        }
        return top_node->data;
```

```
    }

    // Check if the stack is empty
    bool isEmpty() const {
        return top_node == nullptr;
    }

    // Get the number of elements in the stack
    size_t size() const {
        return current_size;
    }
};

/*
int main() {
    MyStackLinkedList<int> s_linkedlist;
    s_linkedlist.push(100);
    s_linkedlist.push(200);
    s_linkedlist.push(300);

    std::cout << "Top element: " << s_linkedlist.top() << "\n";
    s_linkedlist.pop();
    std::cout << "Size: " << s_linkedlist.size() << "\n";
    s_linkedlist.pop();
    s_linkedlist.pop();

    try {
        s_linkedlist.pop(); // This will throw an exception
    } catch (const std::out_of_range& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
    return 0;
}
*/
```

Comparison of Stack Implementations

Feature	Dynamic Array (e.g., `std::vector`)	Linked List
Memory Allocation	Contiguous	Non-contiguous
Dynamic Sizing	Yes (with reallocations)	Yes (node by node)
`push()` / `pop()`	Amortized O(1)	O(1)
`top()`	O(1)	O(1)
Random Access	O(1) (to any element)	O(n) (only top is O(1))
Memory Overhead	Lower (no per-node pointer overhead)	Higher (pointers for each node)
Cache Performance	Good (due to locality)	Poor (due to scattered memory)
Use Case	Preferred for general-purpose stacks where reallocations are acceptable.	Preferred when frequent insertions/deletions at the ends are critical, and random access is not needed. Avoids reallocation overhead.

In C++, `std::stack` (from the `<stack>` header) is a container adaptor that uses `std::deque` by default, which is a double-ended queue that combines the advantages of both dynamic arrays and linked lists for efficient insertions/deletions at both ends. It can also be adapted to use `std::vector` or `std::list` as its underlying container.

Expression Evaluation and Notations

Stacks play a crucial role in evaluating and converting expressions between different notations. The three common notations are Infix, Postfix, and Prefix.

1. Infix Notation

This is the most common way we write mathematical expressions, where operators are placed **between** the operands.

Example: `A + B * C`

Characteristics: * Easy for humans to read and write. * Requires operator precedence and associativity rules for evaluation. * Parentheses are often used to override default precedence.

2. Postfix Notation (Reverse Polish Notation - RPN)

In postfix notation, operators are placed **after** their operands. This notation does not require parentheses or operator precedence rules, making it easier for computers to parse and evaluate.

Example: `A B C * +`

Characteristics: * Operators appear after their operands. * No parentheses needed. * Evaluation is straightforward using a stack: * Scan the expression from left to right. * If an operand is encountered, push it onto the stack. * If an operator is encountered, pop the required number of operands from the stack, perform the operation, and push the result back onto the stack.

3. Prefix Notation (Polish Notation)

In prefix notation, operators are placed **before** their operands.

Example: `+ A * B C`

Characteristics: * Operators appear before their operands. * No parentheses needed. * Evaluation typically involves scanning the expression from right to left, or recursively.

Conversion Between Notations (using Stacks)

Stacks are fundamental for converting expressions from one notation to another, especially from infix to postfix or prefix. The general idea involves:

- **Infix to Postfix:** Use a stack to temporarily hold operators and manage their precedence. Operands are output directly.
- **Infix to Prefix:** Similar to infix to postfix, but often involves reversing the infix expression, converting to postfix, and then reversing the result.

Example: Infix to Postfix Conversion Algorithm (Simplified)

1. Initialize an empty stack for operators and an empty string/list for the postfix expression.
2. Scan the infix expression from left to right.
3. If an operand is found, append it to the postfix expression.
4. If an opening parenthesis `(` is found, push it onto the stack.
5. If a closing parenthesis `)` is found, pop operators from the stack and append them to the postfix expression until an opening parenthesis is encountered. Pop and discard the opening parenthesis.
6. If an operator is found:
    - Pop operators from the stack and append them to the postfix expression as long as the stack is not empty, the top of the stack is not an opening parenthesis, and the operator at the top of the stack has higher or equal precedence than the current operator.
    - Push the current operator onto the stack.
7. After scanning the entire expression, pop any remaining operators from the stack and append them to the postfix expression.

This is a simplified overview, and a full implementation would require handling operator precedence rules and associativity.