

# CSC 212: Data Structures and Abstractions

## 07: Stacks

Prof. Marco Alvarez

Department of Computer Science and Statistics  
University of Rhode Island

Fall 2025



## Stacks, queues, dequeues

- Fundamental **collection** data structures
  - ✓ store and manage collections of elements with specific access patterns
  - ✓ data is manipulated in controlled, predictable order
  - ✓ used in various applications, including algorithm design, data processing, and system design
- Why using specialized data structures?
  - ✓ clear, restricted interfaces prevent misuse and express algorithmic purpose
  - ✓ enforced access patterns reduce programming mistakes
  - ✓ optimized  $\Theta(1)$  operations vs. linear-time overhead in general containers
- Available in many programming languages and libraries
  - ✓ STL C++: `std::stack`, `std::queue`, and `std::deque`
  - ✓ Python: `collections.deque` (more efficient than lists)
  - ✓ Java: `java.util` provides `Stack` and `Queue` interfaces, as well as `ArrayDeque` and `LinkedList`

2

## Stacks

## Stacks

- Last-in-first-out
  - ✓ a **stack** is a linear data structure that follows the (LIFO) principle
  - ✓ the last element added to the stack is the first one to be removed
- Main operations
  - ✓ **push**: add element to the top
  - ✓ **pop**: remove element from the top
- Applications
  - ✓ expression evaluation, backtracking algorithms, undo mechanisms in applications, browser history navigation, etc.



4

# Implementation

## Using arrays

- ✓ **push** and **pop** at the end of the array (easier and efficient)
- ✓ array can be either fixed-length or a dynamic array (additional cost)

## Considerations

- ✓ underflow: throw an error when calling pop on an empty stack
- ✓ overflow: throw an error when calling push on a full stack

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	3	-5	0	9	-2	7	1								

top

<https://www.cs.usfca.edu/~galles/visualization/StackArray.html>

5

```
#ifndef __STACK_H__
#define __STACK_H__

#include <cstdlib>

// class implementing a Stack of integers
// fixed-length array (not a dynamic array)
class Stack {
private:
    // array to store stack elements
    int *array;
    // maximum number of elements stack can hold
    size_t length;
    // current number of elements in stack
    size_t top;

public:
    // IMPORTANT: need to add copy constructor and
    // overload assignment operator
    Stack(size_t);
    ~Stack();

    // pushes an element onto the stack
    void push(int);
    // returns/removes the top element from the stack
    int pop();
    // check if stack is empty
    bool empty() const { return top == 0; }

};

#endif // __STACK_H__
```

6

```
#include "stack.h"
#include <stdexcept>

Stack::Stack(size_t len) {
    if (len < 1) {
        throw std::invalid_argument("Can't create an empty stack");
    }
    length = len;
    array = new int[length];
    top = 0;
}

Stack::~Stack() {
    delete [] array;
}

void Stack::push(int value) {
    if (top == length) {
        throw std::out_of_range("Stack is full");
    } else {
        array[top] = value;
        top++;
    }
}

int Stack::pop() {
    if (top == 0) {
        throw std::out_of_range("Stack is empty");
    } else {
        top--;
        return array[top];
    }
}
```

```
class Stack {
private:
    int *array;
    size_t length;
    size_t top;

public:
    Stack(size_t);
    ~Stack();

    void push(int);
    int pop();
    bool empty();
};
```

7

## Practice

### What is the output of this code?

```
#include <iostream>
#include "stack.h"

int main() {
    Stack s1(10), s2(10);

    s1.push(100);
    s2.push(s1.pop());
    s1.push(200);
    s1.push(300);
    s2.push(s1.pop());
    s2.push(s1.pop());

    s1.push(s2.pop());
    s1.push(s2.pop());

    while (!s1.empty()) {
        std::cout << s1.pop() << std::endl;
    }

    while (!s2.empty()) {
        std::cout << s2.pop() << std::endl;
    }

    return 0;
}
```

8

## Example application

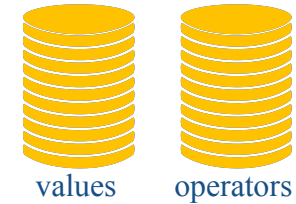
- Fully parenthesized infix expressions
  - infix expression: operators are placed between two operands
  - fully parenthesized: every operator and its operands are contained in parentheses
  - operator precedence and associativity don't matter
  - parentheses dictate exact computation order
- Consider an algorithm for evaluating fully parenthesized infix expressions

$((5 + ((10 - 4) * (3 + 2))) + 25)$

9

## Dijkstra's two-stack algorithm

- Create two stacks:
  - values (for operands) and operators (for operators)
- Process the expression from left to right, token by token:
  - if left parenthesis, ignore it
  - if operand, push it onto values stack
  - if operator, push it onto operators stack
  - if right parenthesis:
    - pop operator from operators stack
    - pop two elements from values stack
    - apply operator to those operands in the correct order
      - <result = second-popped operator first-popped>
    - push the result back onto values stack



10

## Practice

- Trace the 2-stack algorithm with the following expression

$((5 + ((10 - 4) * (3 + 2))) + 25)$

11

## Practice

- Design an algorithm using a single stack to verify if the following code has balanced parenthesis or not
  - consider the following characters as parenthesis: `()`, `{}`, `[]`

```
int foo(int x) { return (x > 0 ? new int[x]{x}[0] : x * (2)); }
```

12

# Quick detour (C++ templates)

## Templates

- How to modify the code to support adding floats, or other data types?

```
#include <iostream>

int add_int(int a, int b) {
    return a + b;
}

double add_double(double a, double b) {
    return a + b;
}

int main() {
    std::cout << "Sum (int): " << add_int(5, 3) << "\n";
    std::cout << "Sum (double): " << add_double(2.5, 1.7) << "\n";

    return 0;
}
```

14

## Templates

```
#include <iostream>

template <typename T>
T add(T a, T b) {
    return a + b;
}


int main() {
    std::cout << "Sum (int): " << add<int>(5, 3) << "\n";
    std::cout << "Sum (double): " << add<double>(2.5, 1.7) << "\n";

    return 0;
}
```

Template functions/classes allow writing **generic code** that can work with different data types without the need to write separate code for each type. The compiler generates the appropriate instantiation based on the data type specified to the function/class.

15

## Class templates

<pre>class Stack { private:     int *array;     int length;     int top;  public:     Stack(int);     ~Stack();      void push(int);     void pop();     int peek(); };</pre>		<pre>template &lt;typename T&gt; class Stack { private:     T *array;     size_t length;     size_t top;  public:     Stack(size_t);     ~Stack();      void push(T);     void pop();     T peek(); };</pre>
---	---	--

16