CSC 212: Data Structures and Abstractions

07: Stacks

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

Quick detour (C++)

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;
}
```


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 <u>without the need to write separate code for each type</u>. The compiler generates the appropriate instantiation based on the data type specified to the function/class.

Important C++ topics to review

- Memory model and pointers
- Dynamic memory allocation
- Classes and objects
- References
- Templates
- STL containers

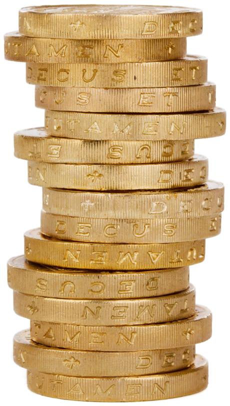
Stacks

Stacks and queues

- Fundamental data structures used to store and manage **collections** of elements
  - provide a way to organize and manipulate data in a specific order
  - used in various applications, including algorithm design, data processing, and system design
  - better to define stacks and queues separately than using existing vectors/arrays/lists (clarity, error-prevention, efficiently)
- Available in many programming languages and libraries
  - in C++ `std::stack` and `std::queue` are the standard library implementations of stacks and queues, respectively
  - in Python, the `collections` module provides `deque` (more efficient than lists), which can be used as a stack or a queue
  - in Java, the `java.util` package provides `Stack` and `Queue` interfaces, as well as implementations such as `ArrayDeque` and `LinkedList`

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 an element to the top of the stack
  - Pop: remove the element from the top of the stack
- Applications
  - expression evaluation, backtracking algorithms, undo mechanisms in applications, browser history navigation, etc.



Implementation

- Using arrays
  - push and pop at the end of the array (easier and efficient)
  - array can be 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


```
// 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
        int length;
        // current number of elements in stack
        int top;

    public:

        Stack(int);
        ~Stack();

        // pushes an element onto the stack
        void push(int);
        // returns/removes the top element from the stack
        int pop();
};
```



```
Stack::Stack(int len) {
    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];
    }
}
```


Using templates

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

    public:

        Stack(int);
        ~Stack();

        void push(int);
        void pop();
        int peek();
};
```

```
template <typename T>
class Stack {
    private:
        T *array;
        size_t length;
        size_t top;

    public:

        Stack(size_t);
        ~Stack();

        void push(T);
        void pop();
        T peek();
};
```

Practice

- What is the output of this code?

```
Stack<int> s1, s2;

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;
}
```


Example application

- Fully parenthesized infix expressions
  - infix arithmetic expression where every operator and its arguments are contained in parentheses
    - infix arithmetic expressions: operators are placed between two operands
  - operator precedence and associativity don't matter
    - every operation is explicitly enclosed in parentheses

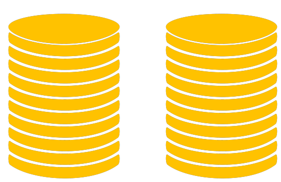
- Consider an algorithm for evaluating fully parenthesized infix expressions

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

Algorithm

- Create two stacks:
  - values (for operands) and operators (for operators)

- Process the expression from left to right, character by character:
  - if left parenthesis, ignore it
  - if operand, push it onto values stack
  - if operator, push onto operators stack
  - if right parenthesis:
    - pop operator from operators stack
    - pop two elements from values stack
      - the second element popped is the first operand
    - apply operator to those operands in the correct order
      - first-operand operator second-operand
    - push the result back onto values stack



Practice

- Trace the 2-stack algorithm with the following expression

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

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)); }
```