CSC 212: Data Structures and Abstractions

Stacks and Queues

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

THINK BIG WE DO℠

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 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.

Important C++ topics to review

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

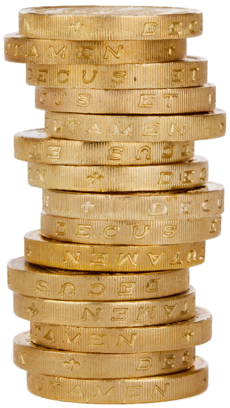
Stacks

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

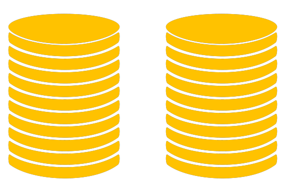
- Design an algorithm for evaluating fully parenthesized infix expressions

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

Algorithm

- Create two stacks (for operands and operators)

- Process the string from left to right
  - if left parenthesis, ignore
  - if value, push onto values stack
  - if operator, push onto operators stack
  - if right parenthesis:
    - pop operator and two values
    - apply operator to those values in the order they are popped
    - push result back onto values stack



Practice

- Trace the 2-stack algorithm with the following expression

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

Practice

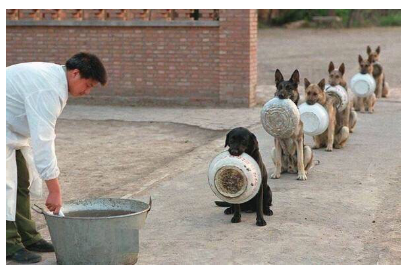
- Design an algorithm using a single stack to verify if the following code has <u>balanced parenthesis</u> or not
  - consider the following characters as parenthesis: `()`, `{}`, `[]`

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


Queues

Queues

- First-in-first-out
  - a **<u>queue</u>** is a linear data structure that follows the (FIFO) principle
  - the first element added to the queue is the first one to be removed
    - analogous to a real-world queue, such as a line of people waiting for service
- Main operations
  - Enqueue: add an element to the end of the queue
  - Dequeue: remove an element from the front of the queue
- Applications
  - scheduling tasks in operating systems, managing requests in web servers, implementing breadth-first search (BFS) in graph algorithms, etc.



Implementation

- Using arrays
  - ensure enqueue and dequeue work at different ends of the array
  - array can be fixed-length or a dynamic array (additional cost)
- Considerations
  - underflow: throw an error when calling dequeue on an empty queue
  - overflow: throw an error when calling enqueue on a full queue

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

https://www.cs.usfca.edu/~galles/visualization/QueueArray.html

```cpp
// implements a queue using a fixed-size array
class Queue {
    private:
        // array to store queue elements
        int *array;
        // maximum number of elements queue can hold
        int length;
        // index of the first element in the queue
        int base;
        // index of the last element in the queue
        int top;

    public:

        Queue(int);
        ~Queue();

        // adds an element to the end of the queue
        void enqueue(int);
        // removes the first element from the queue
        int dequeue();
};
```


Practice

- What is the output of this code?

```cpp
Queue<int> s1, s2;

s1.enqueue(100);
s2.enqueue(s1.dequeue());
s1.enqueue(200);
s1.enqueue(300);
s2.enqueue(s1.dequeue());
s2.enqueue(s1.dequeue());

s1.enqueue(s2.dequeue());
s1.enqueue(s2.dequeue());

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

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


Practice

- Design an algorithm to:
  - load a number of audio files (songs)
  - play them in a continuous loop

Stacks and queues in the STL (C++)

std::stack

Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

The `std::stack` class is a container adaptor that gives the programmer the functionality of a stack - specifically, a LIFO (last-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

Member functions

(constructor)	constructs the `stack` (public member function)
(destructor)	destructs the `stack` (public member function)
operator=	assigns values to the container adaptor (public member function)
Element access	
top	accesses the top element (public member function)
Capacity	
empty	checks whether the container adaptor is empty (public member function)
size	returns the number of elements (public member function)
Modifiers	
push	inserts element at the top (public member function)
push_range (C++23)	inserts a range of elements at the top (public member function)
emplace (C++11)	constructs element in-place at the top (public member function)
pop	removes the top element (public member function)
swap (C++11)	swaps the contents (public member function)

```
#include <cassert>
#include <stack>

int main()
{
    std::stack<int> stack;
    assert(stack.size() == 0);

    const int count = 8;
    for (int i = 0; i != count; ++i)
        stack.push(i);
    assert(stack.size() == count);
}
```

https://en.cppreference.com/w/cpp/container/stack

std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

The `std::queue` class template is a container adaptor that gives the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes the elements on the back of the underlying container and pops them from the front.

Member functions

(constructor)	constructs the queue (public member function)
(destructor)	destructs the queue (public member function)
operator=	assigns values to the container adaptor (public member function)
Element access	
front	access the first element (public member function)
back	access the last element (public member function)
Capacity	
empty	checks whether the container adaptor is empty (public member function)
size	returns the number of elements (public member function)
Modifiers	
push	inserts element at the end (public member function)
push_range (C++23)	inserts a range of elements at the end (public member function)
emplace (C++11)	constructs element in-place at the end (public member function)
pop	removes the first element (public member function)
swap (C++11)	swaps the contents (public member function)

```
#include <cassert>
#include <queue>

int main()
{
    std::queue<int> queue;
    assert(queue.size() == 0);

    const int count = 8;
    for (int i = 0; i != count; ++i)
        queue.push(i);
    assert(queue.size() == count);
}
```

https://en.cppreference.com/w/cpp/container/queue