Algerian Olympiad
in Informatics

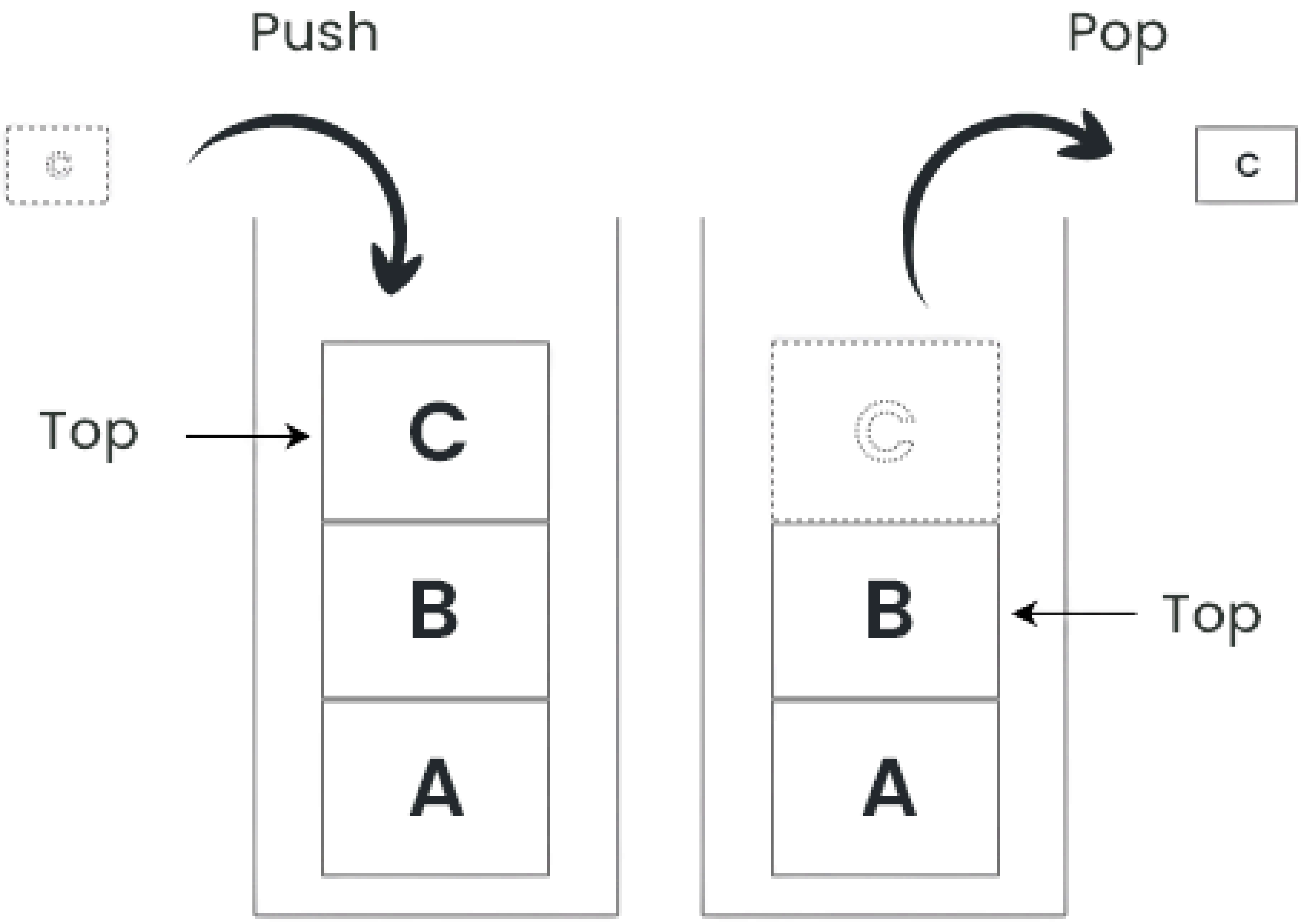
Linear Data Structures

Stacks and Queues

Presented by Hiba Hamidi

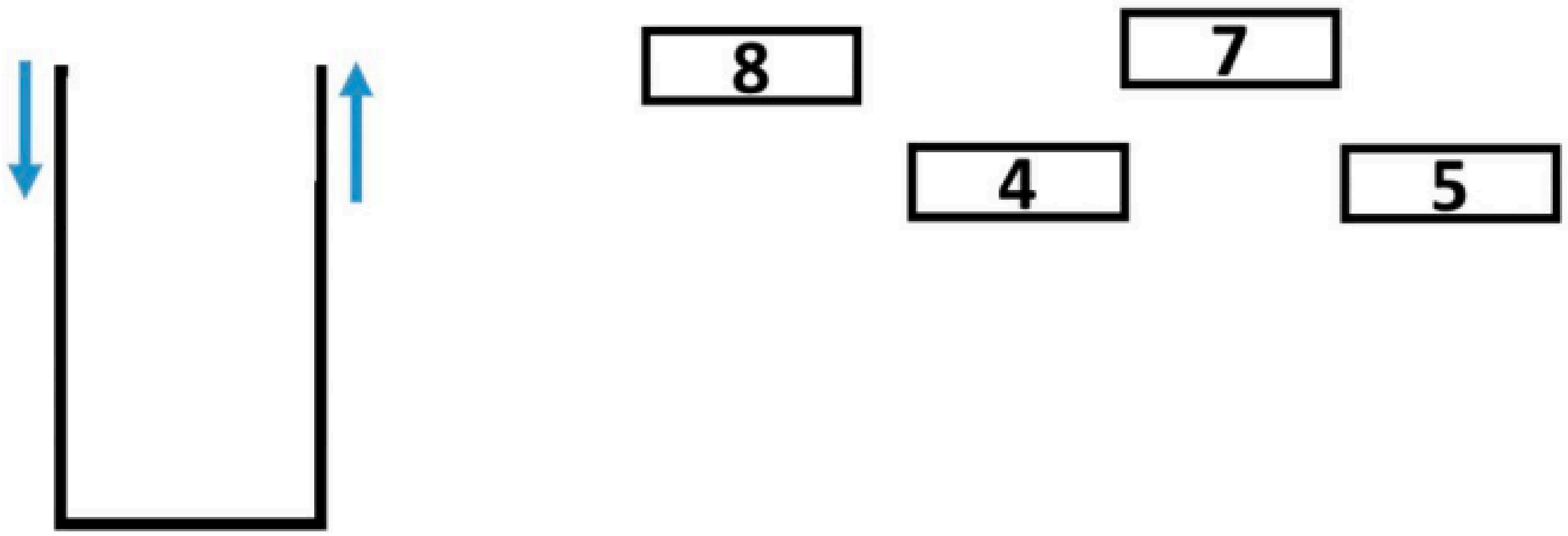
Stack

Data structure



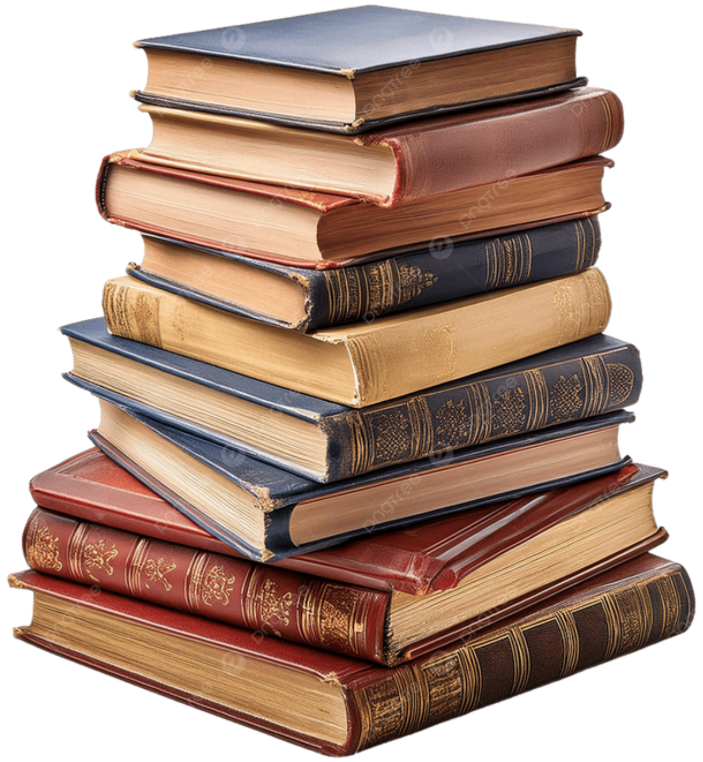
Introduction to stacks

Stack data structure is a linear data structure that accompanies a principle known as LIFO (Last In First Out) or FILO (First In Last Out).



Real life examples

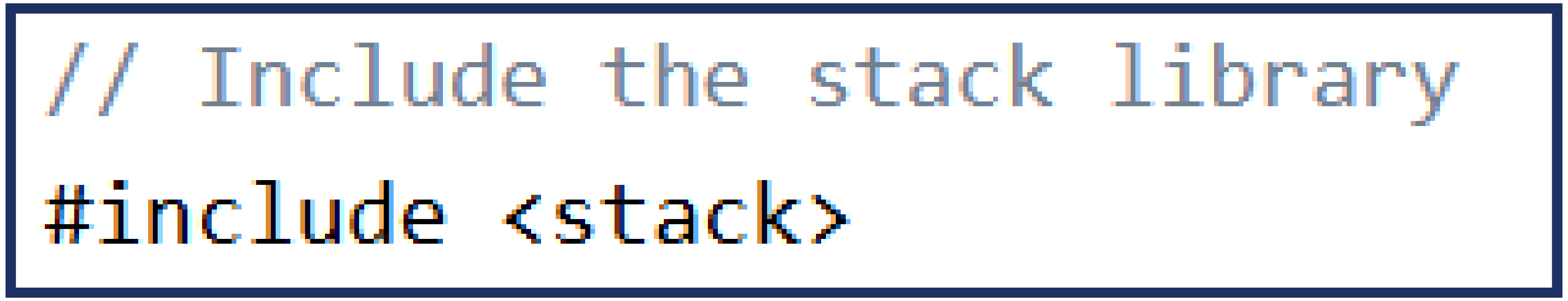
Stack of plates



Stack of books

Library

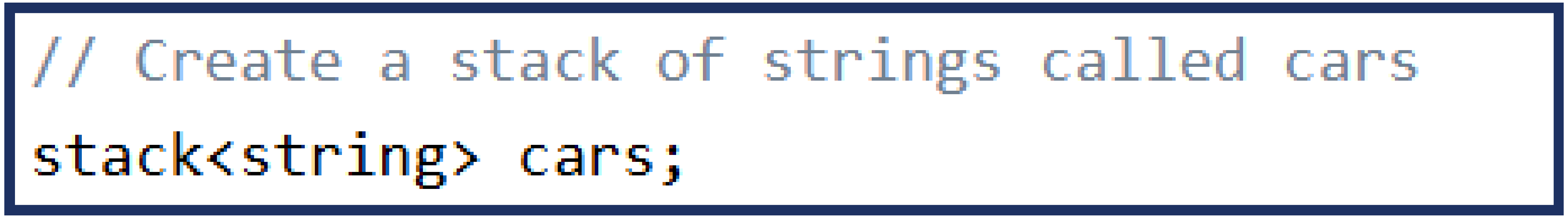
To use a stack, you have to include the <stack> header file:



```
// Include the stack library
#include <stack>
```

Create a stack

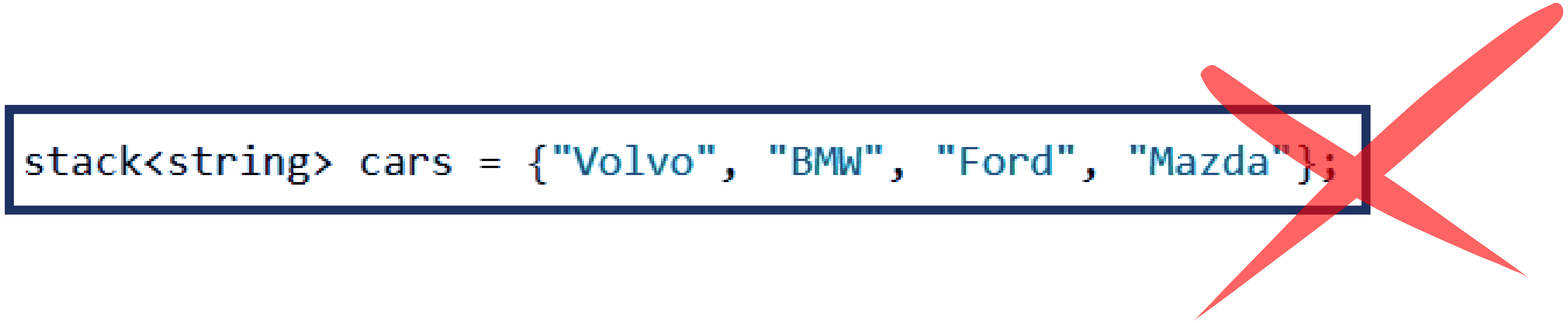
To create a stack, use the stack keyword, and specify the type of values it should store within angle brackets <> and then the name of the stack, like: stack<type> stackName.



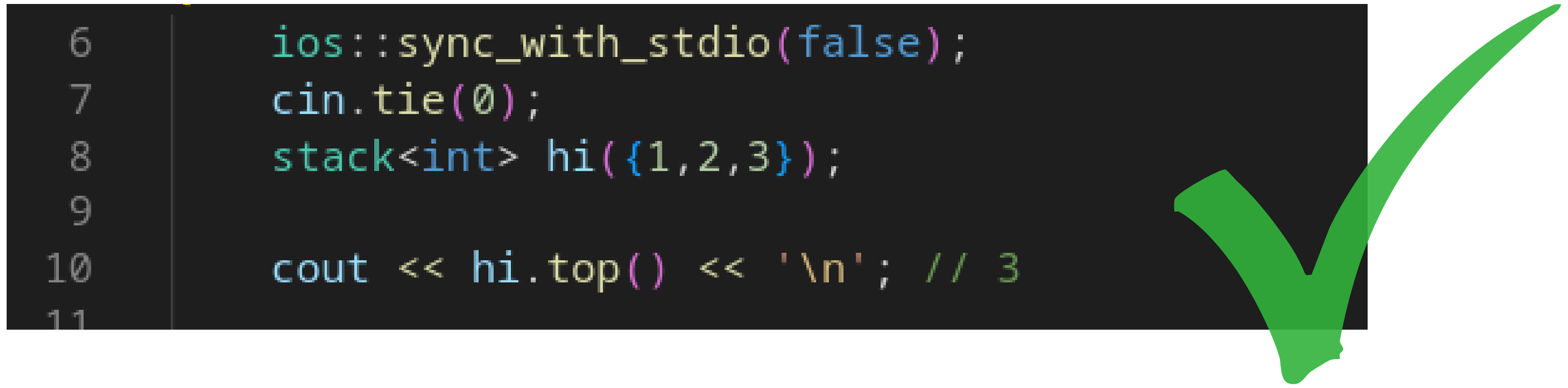
```
// Create a stack of strings called cars
stack<string> cars;
```

NOTE !!!

You CANNOT add elements to the stack at the time of declaration, like you can with vectors:

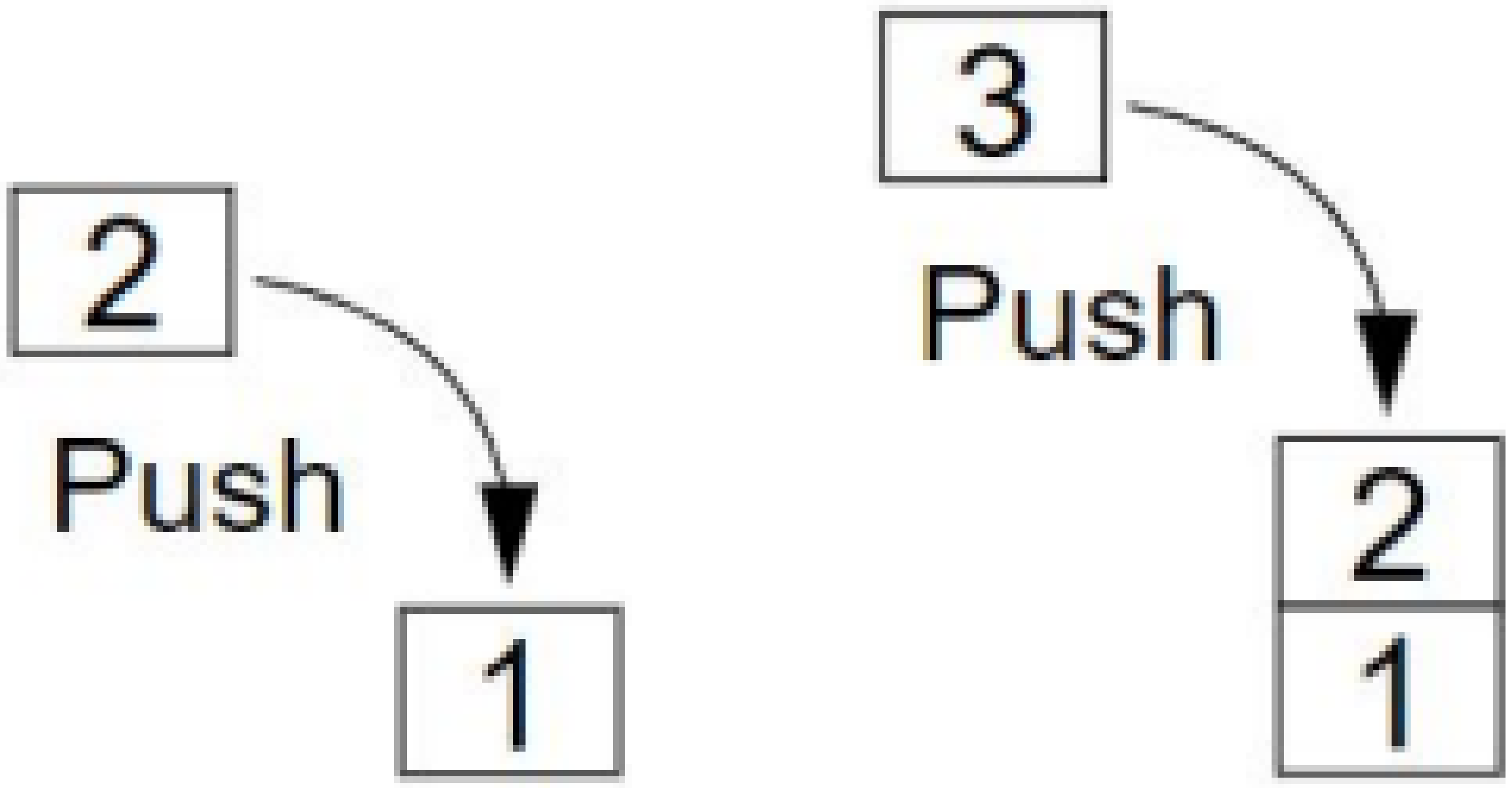


```
stack<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
```



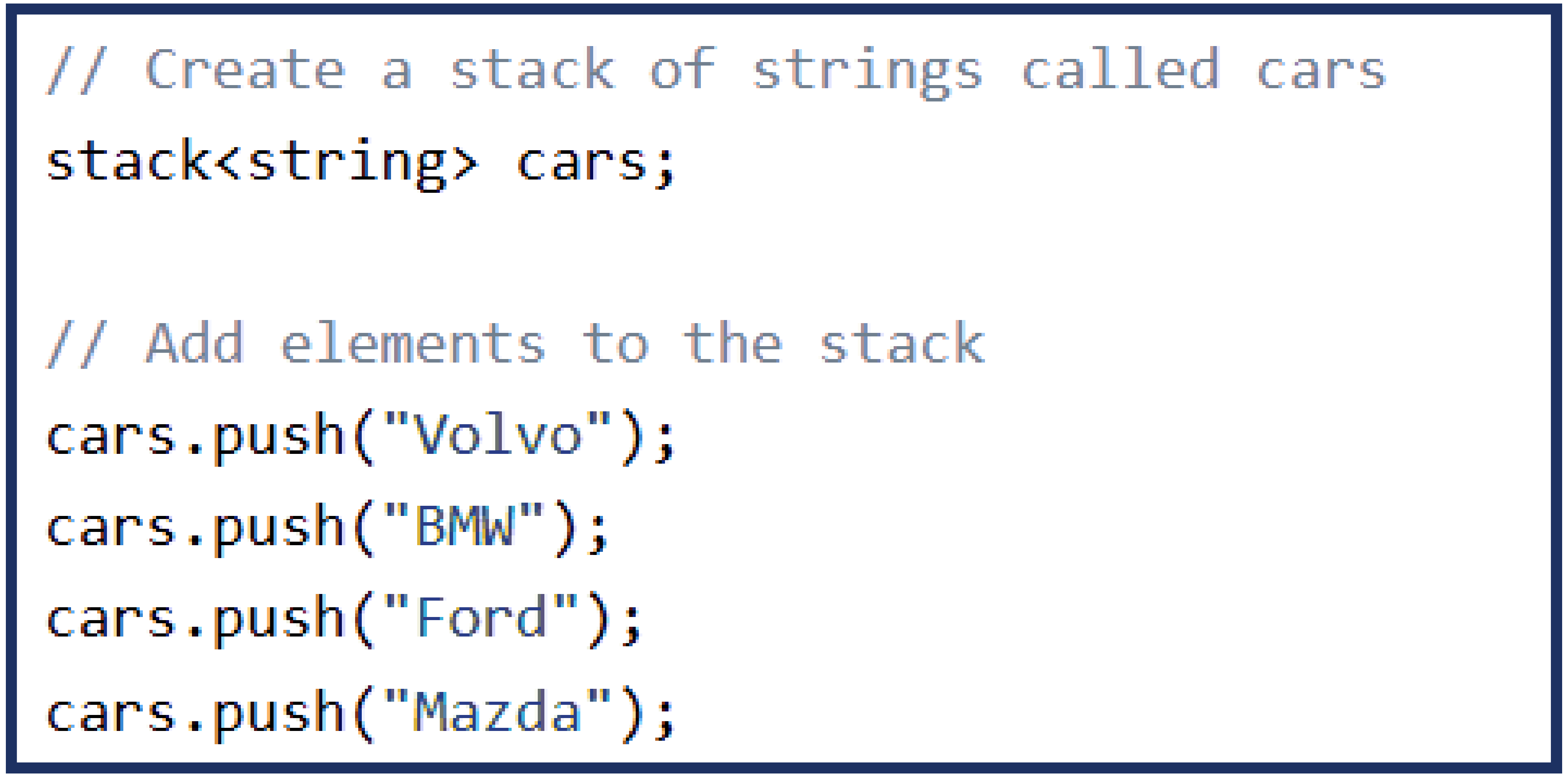
```
    ios::sync_with_stdio(false);
    cin.tie(0);
    stack<int> hi({1,2,3});

    cout << hi.top() << '\n'; // 3
```

Add Elements



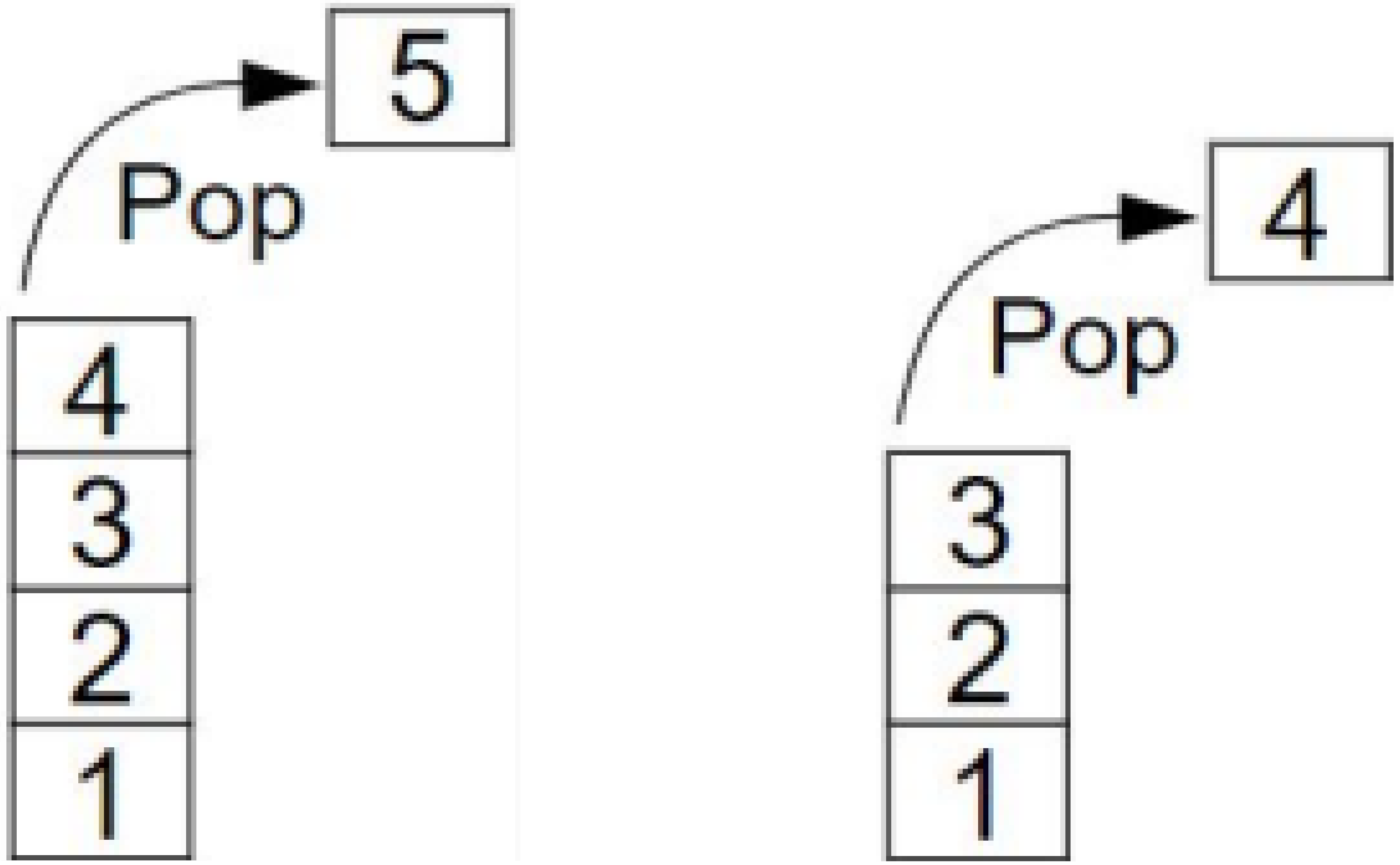
Add Elements

To add elements to the stack, use the .push() function, after declaring the stack:



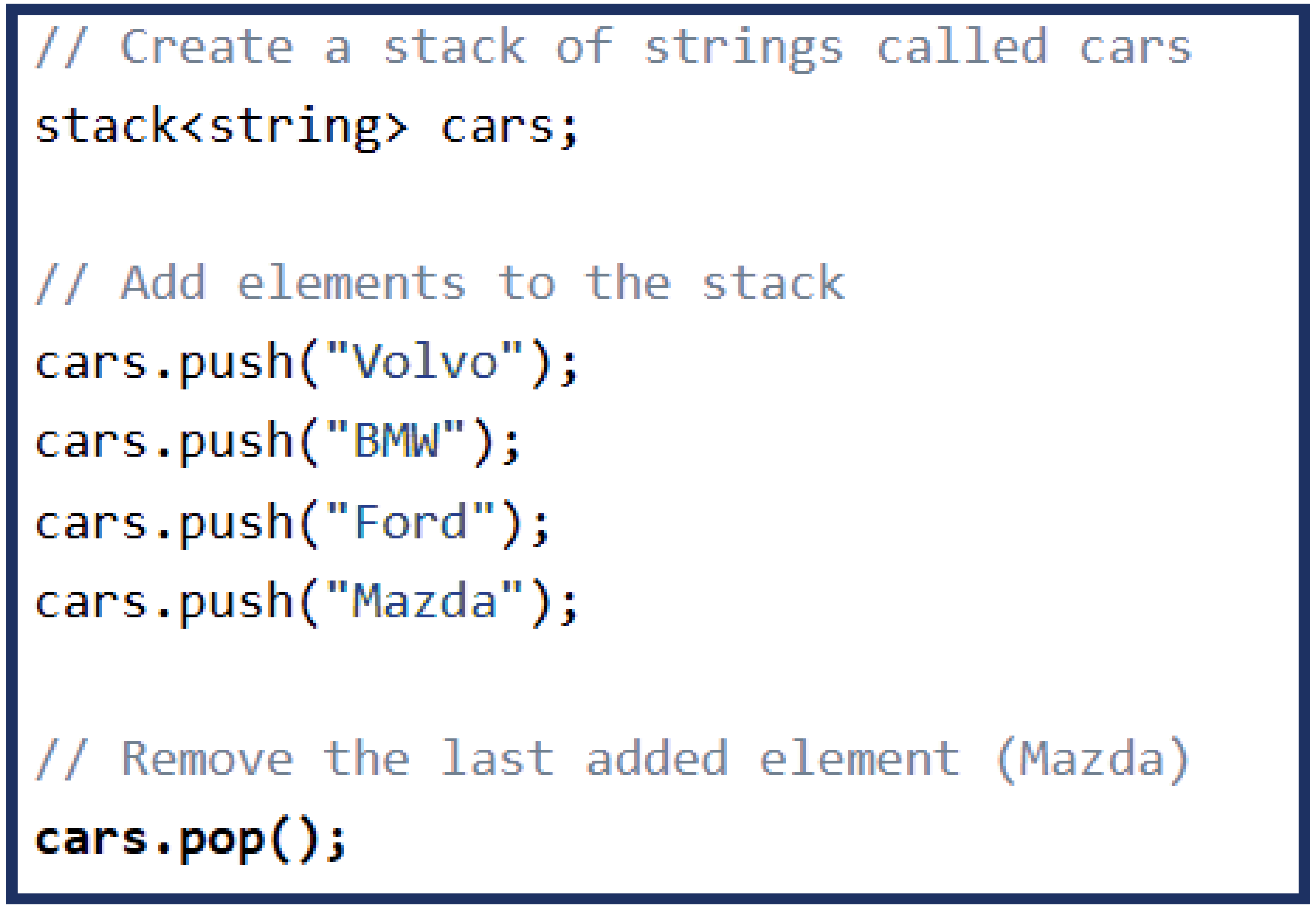
```
// Create a stack of strings called cars
stack<string> cars;

// Add elements to the stack
cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");
```

Remove Elements



Remove Elements

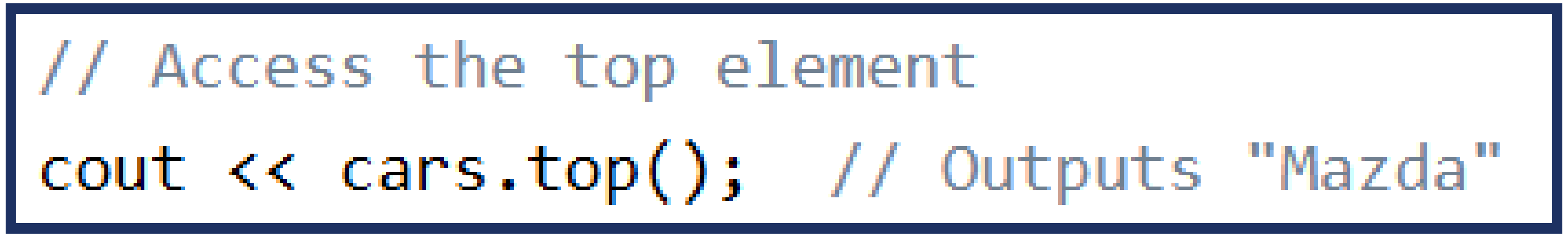
You can use the .pop() function to remove an element from the stack.



```
// Create a stack of strings called cars
stack<string> cars;

// Add elements to the stack
cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");

// Remove the last added element (Mazda)
cars.pop();
```

Access Stack Elements

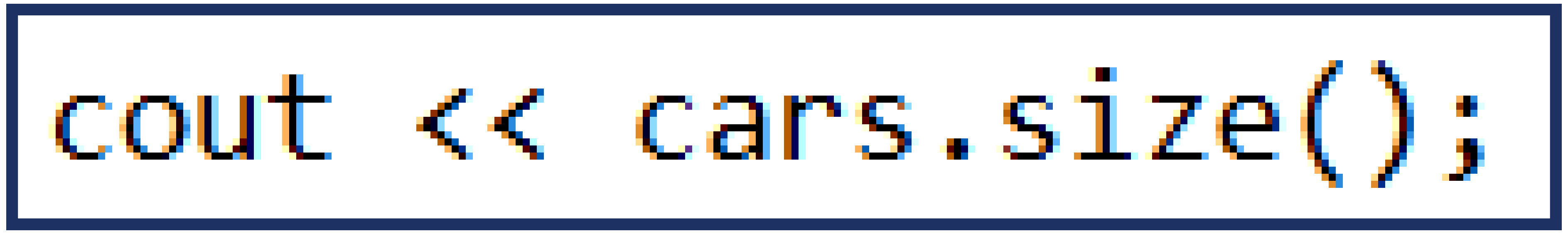
In a stack, you can only access the top element, which is done using the .top() function:



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

Get the Size of the Stack

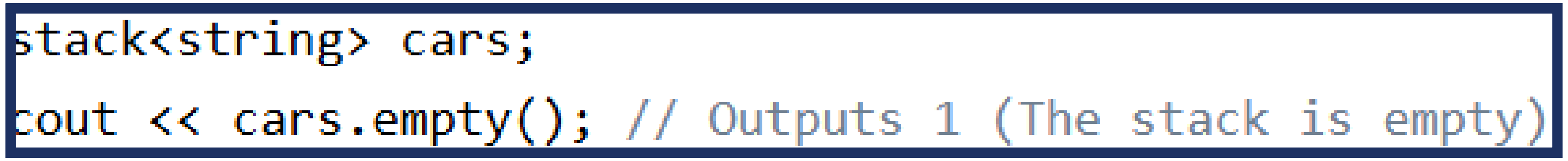
To find out how many elements a stack has, use the .size() function:



```
cout << cars.size();
```

Check if the Stack is Empty

Use the .empty() function to find out if the stack is empty or not.
The .empty() function returns 1 (true) if the stack is empty and 0 (false) otherwise:



```
stack<string> cars;
cout << cars.empty(); // Outputs 1 (The stack is empty)
```

Time and space complexity

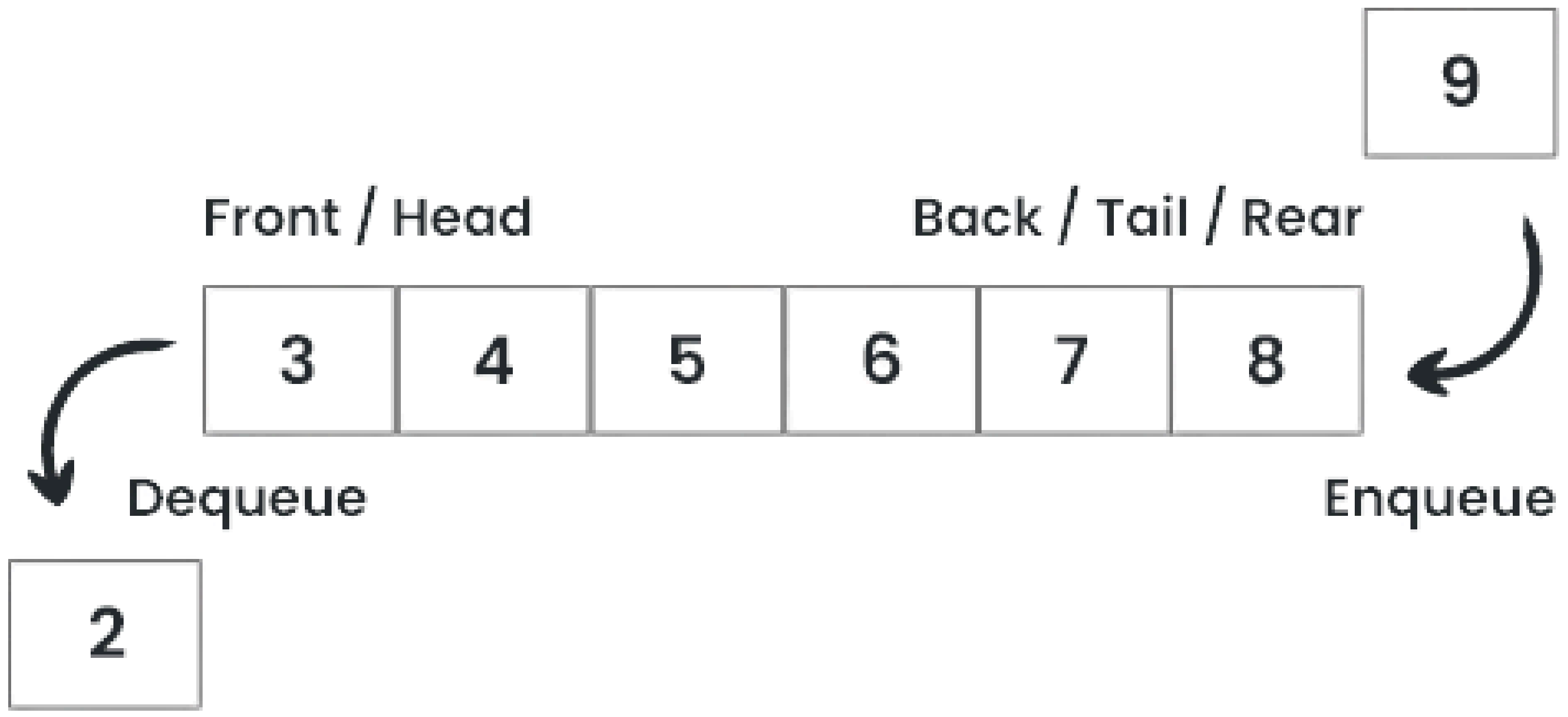
- Push
    - Time complexity - ___?
- Pop
    - Time complexity - ___?
- Peek
    - Time complexity - ___?
- isEmpty()
    - Time complexity - ___?

Time and space complexity

- Push
  - Time complexity - O(1)
- Pop
  - Time complexity - O(1)
- Peek
  - Time complexity - O(1)
- isEmpty()
  - Time complexity - O(1)

Queue

Data structure

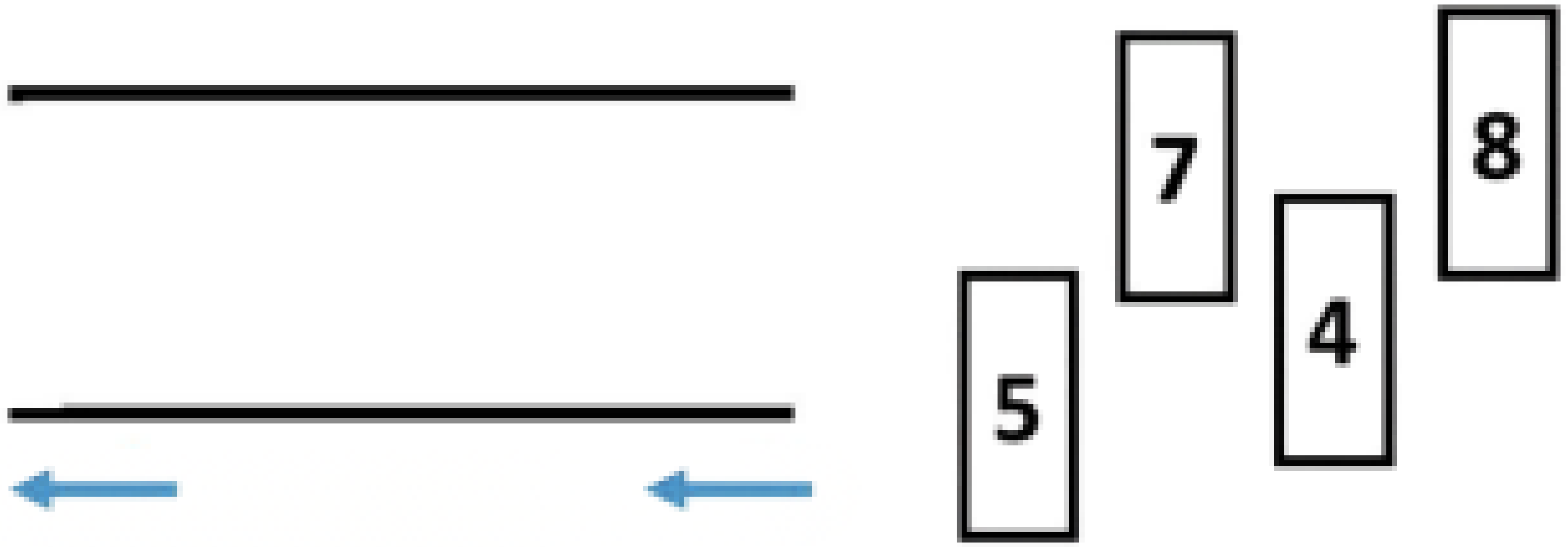


Front / Head

Back / Tail / Rear

Introduction to queues

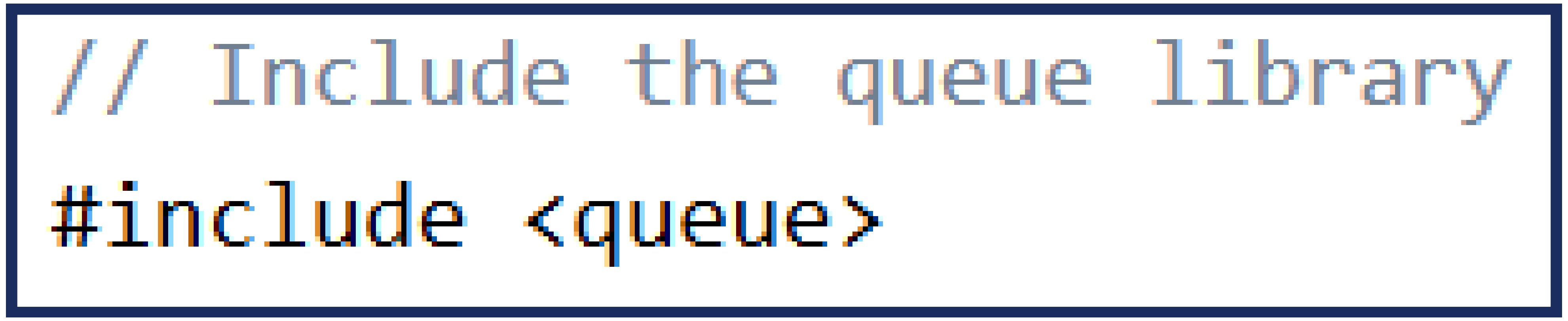
A collection whose elements are added at one end (the back) and removed from the other end (the front). Uses FIFO data handling



Real life examples

Library

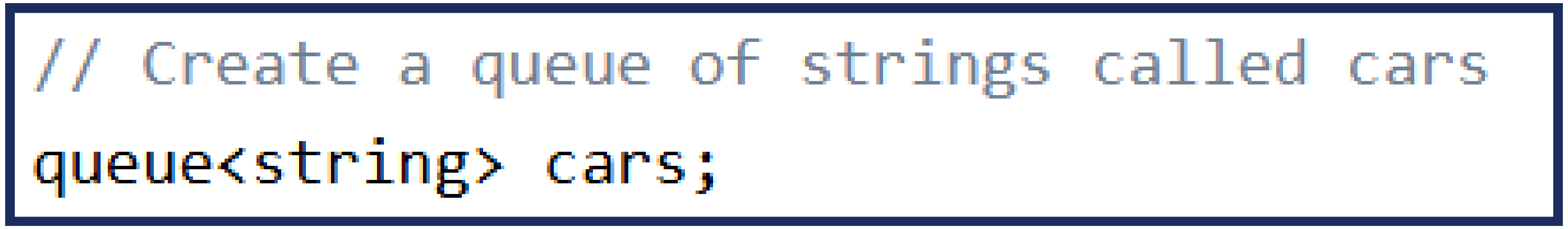
To use a stack, you have to include the <queue> header file:



```
// Include the queue library
#include <queue>
```

Create a queue

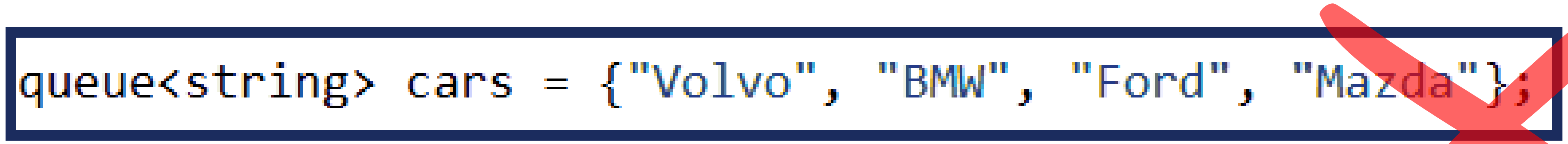
To create a queue, use the queue keyword, and specify the type of values it should store within angle brackets <> and then the name of the queue, like: queue<type> queueName.



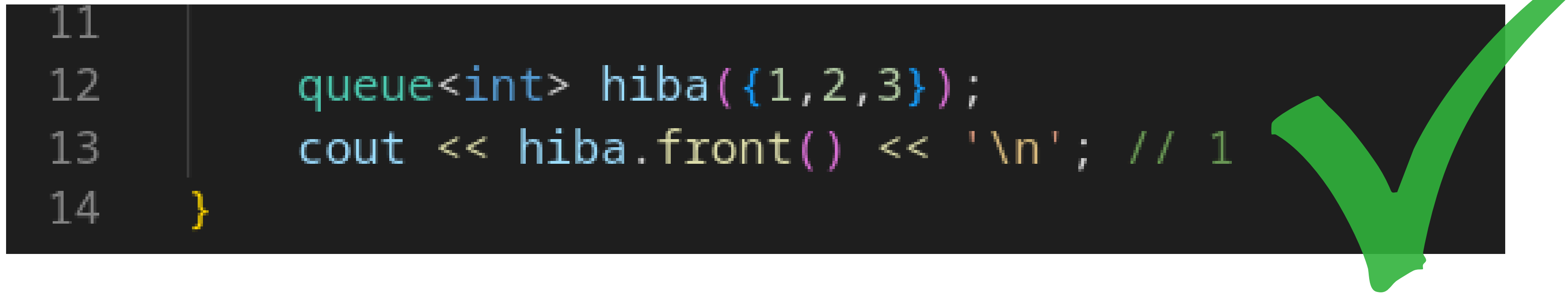
```
// Create a queue of strings called cars
queue<string> cars;
```

NOTE !!!

You **CANNOT** add elements to the stack at the time of declaration, like you can with vectors:

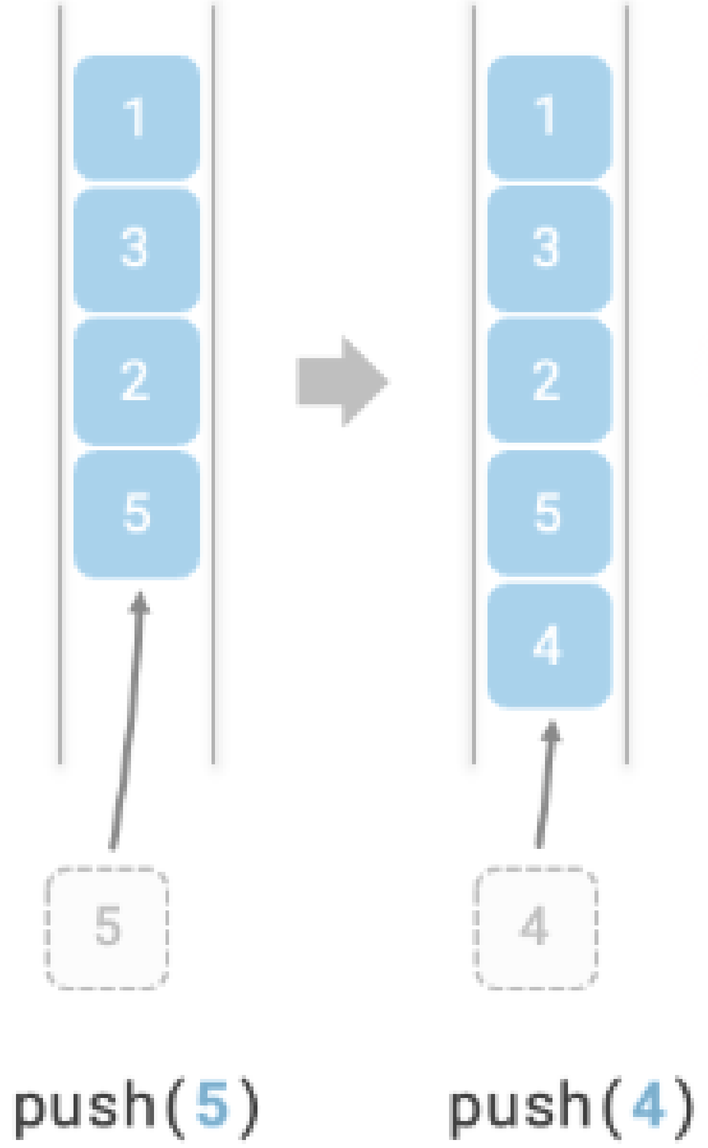


```
queue<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
```



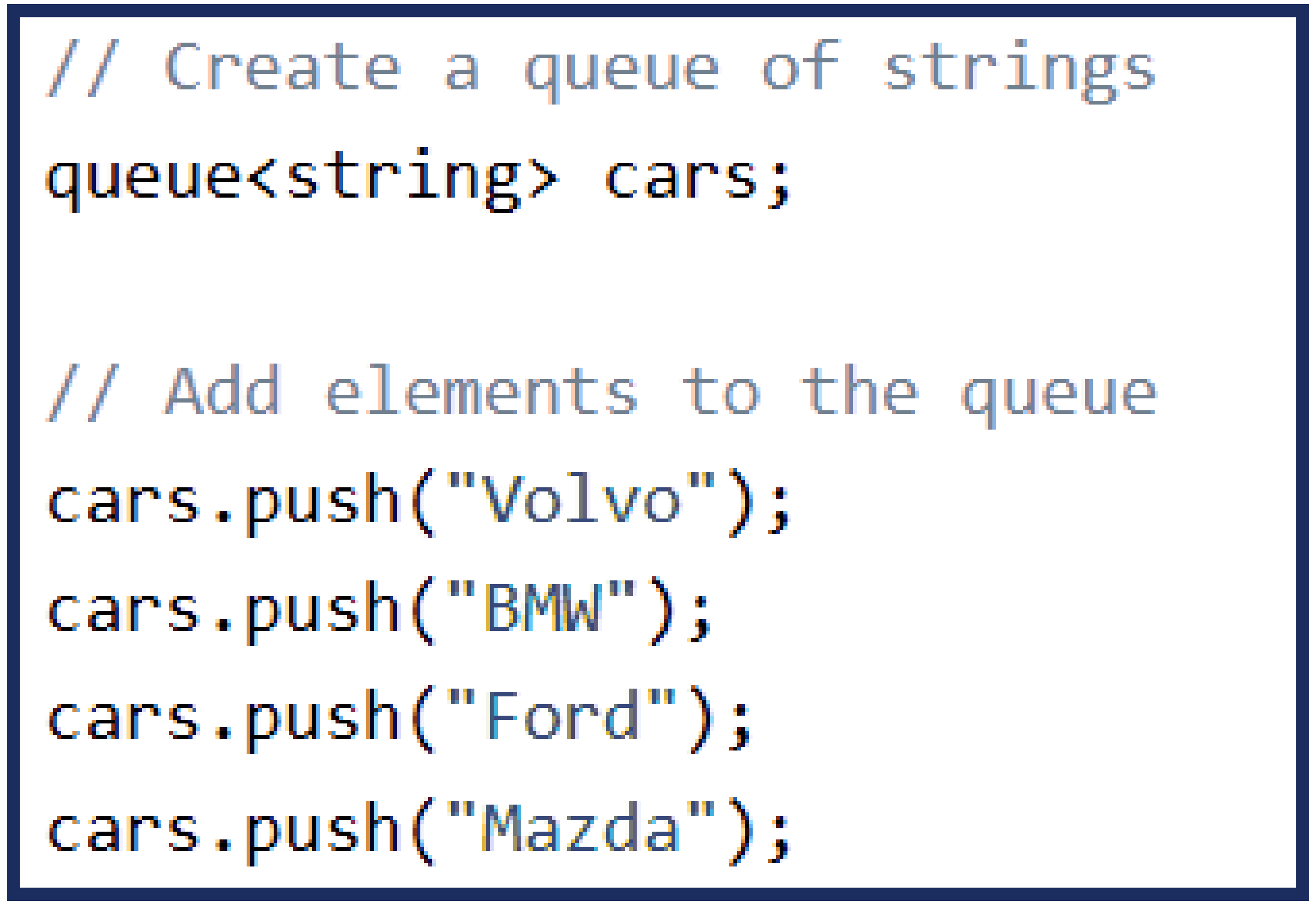
```

    queue<int> hiba({1,2,3});
    cout << hiba.front() << '\n'; // 1
}
```

Add Elements

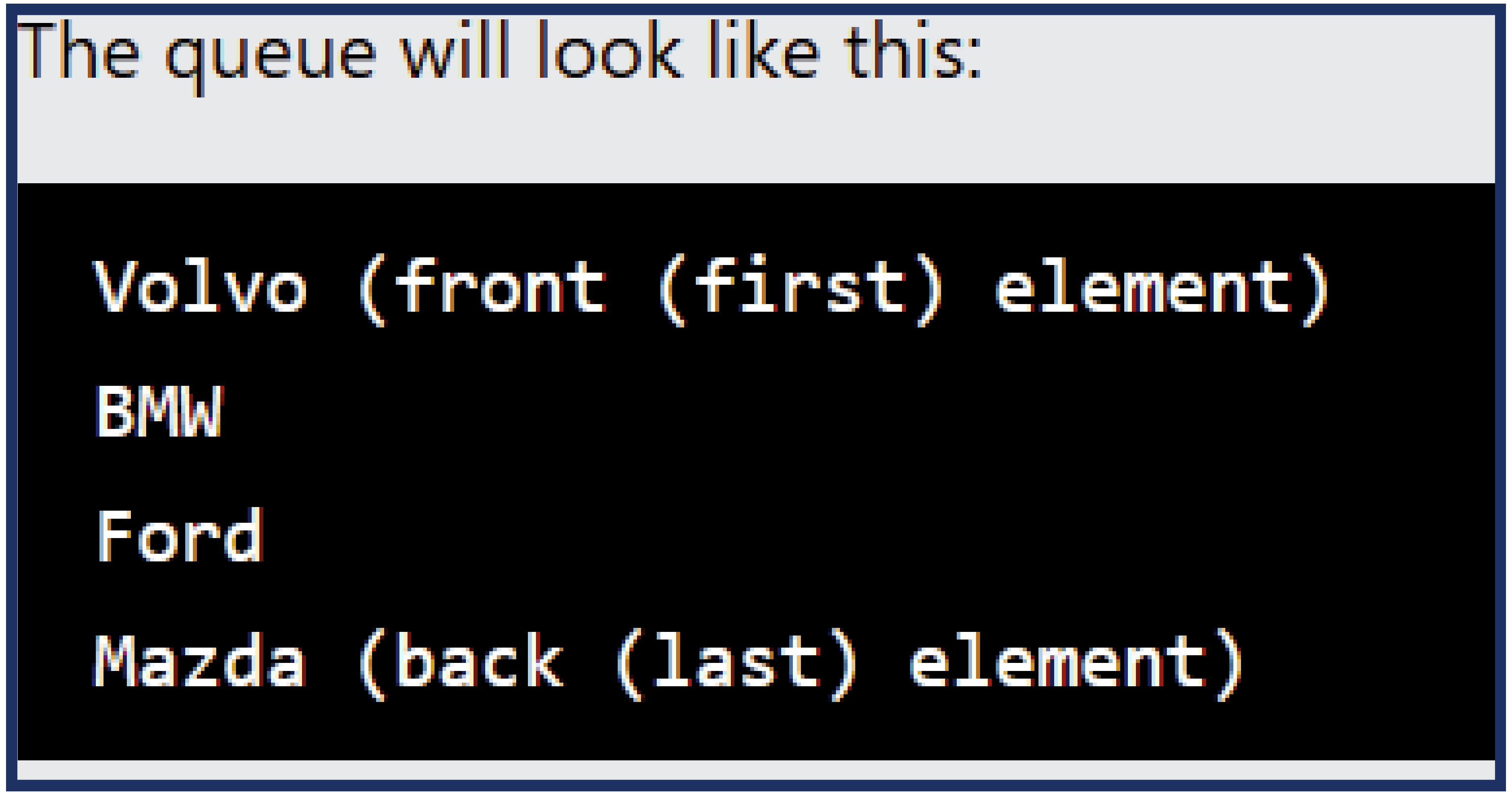


Add Elements

To add elements to the queue, you can use the .push() function after declaring the queue.



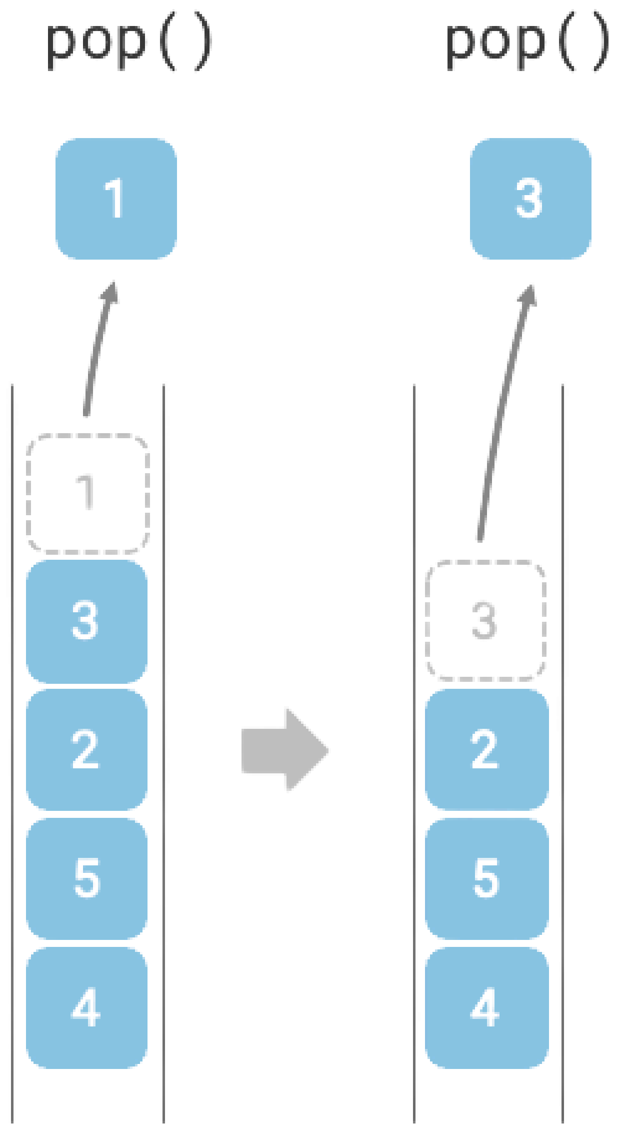
```
// Create a queue of strings
queue<string> cars;

// Add elements to the queue
cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");
```



The queue will look like this:

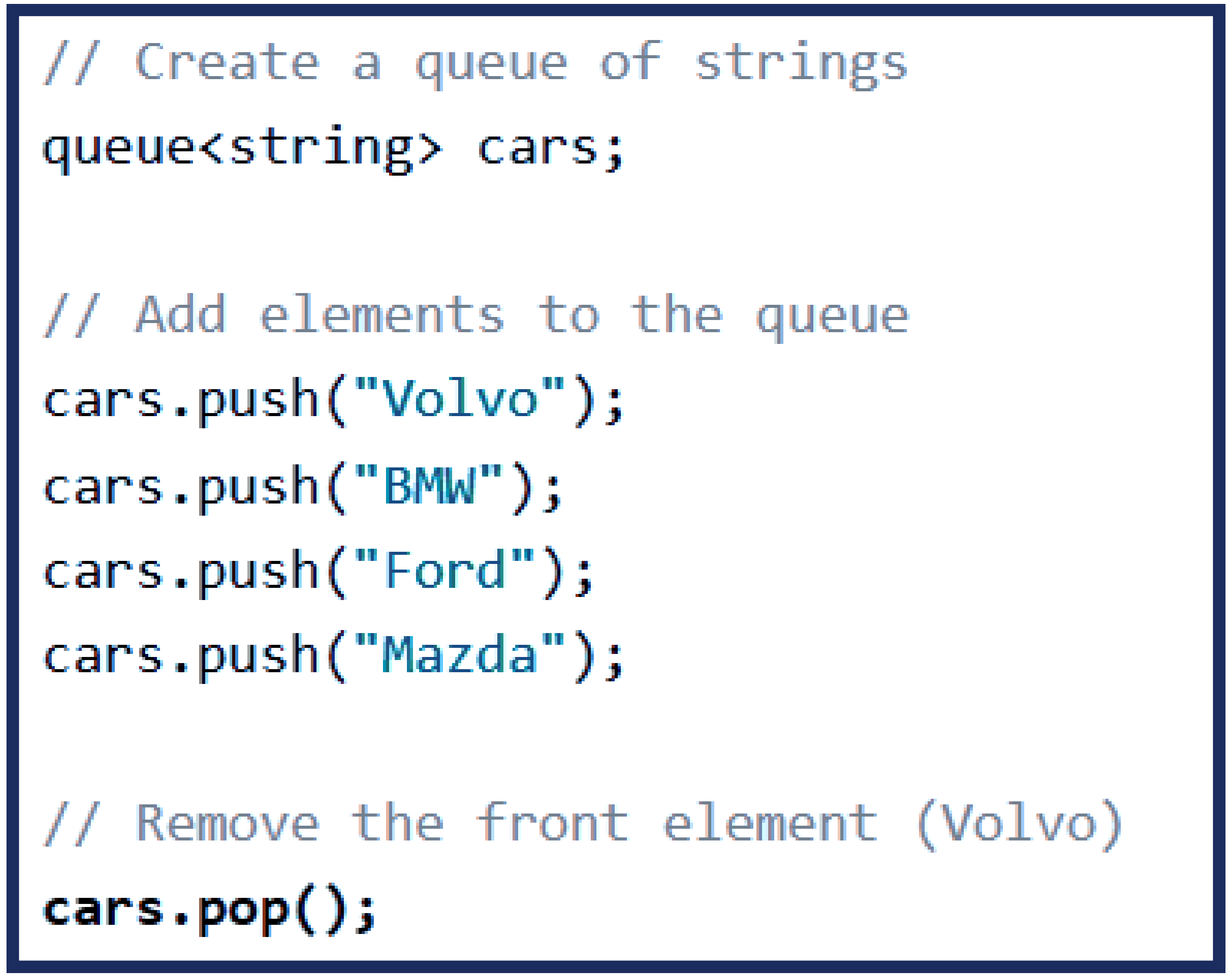
```
Volvo (front (first) element)
BMW
Ford
Mazda (back (last) element)
```

Remove Elements



Remove Elements

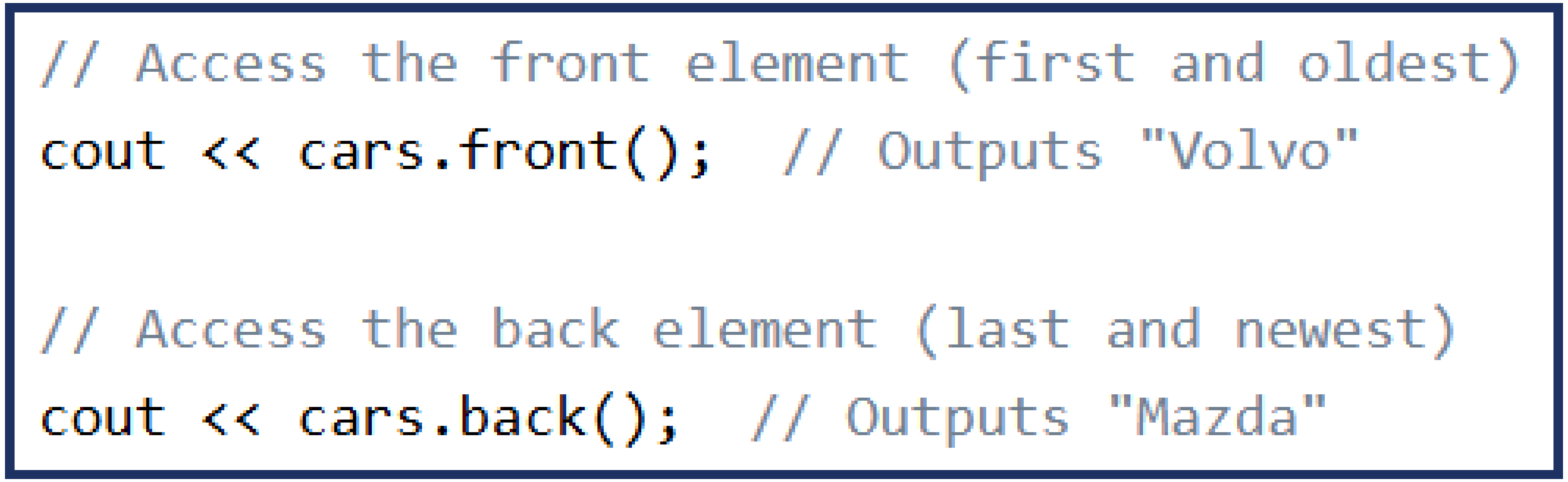
You can use the .pop() function to remove an element from the queue.



```
// Create a queue of strings
queue<string> cars;

// Add elements to the queue
cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");

// Remove the front element (Volvo)
cars.pop();
```

Access Queue Elements

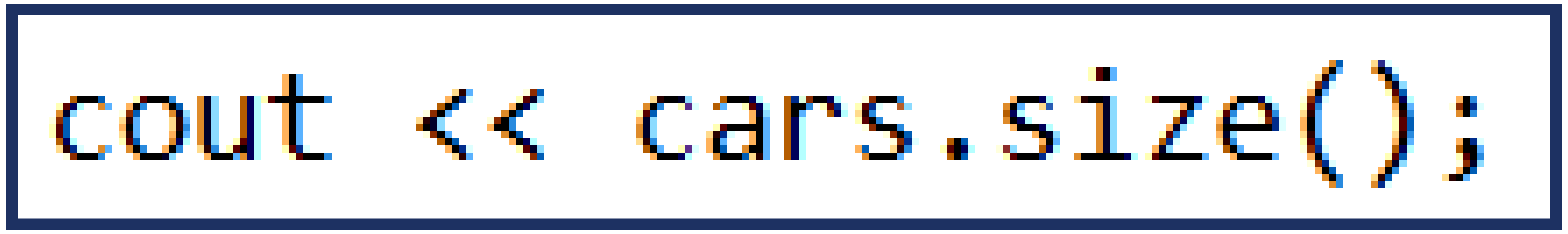
In a queue, you can only access the element at the front or the back, using .front() and .back() respectively:



```
// Access the front element (first and oldest)
cout << cars.front();  // Outputs "Volvo"

// Access the back element (last and newest)
cout << cars.back();  // Outputs "Mazda"
```

Get the Size of the Queue

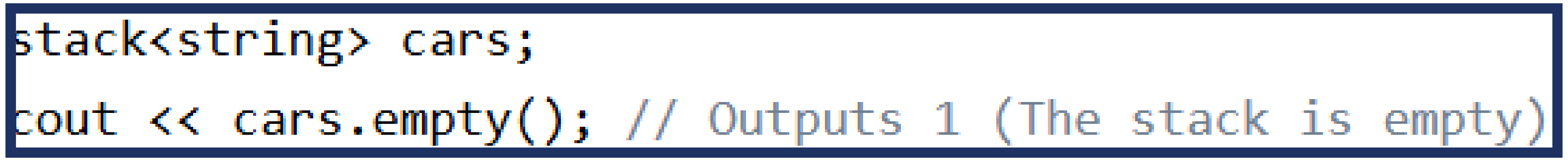
To find out how many elements a queue has, use the .size() function:



```
cout << cars.size();
```

Check if the Queue is Empty

Use the .empty() function to find out if the queue is empty or not.
The .empty() function returns 1 (true) if the queue is empty and 0 (false) otherwise:



```
stack<string> cars;
cout << cars.empty(); // Outputs 1 (The stack is empty)
```

Time and space complexity

- Push
  - Time complexity - ___?
- Pop
  - Time complexity - ___?
- Peek
  - Time complexity - ___?
- isEmpty()
  - Time complexity - ___?

Time and space complexity

- Push
  - Time complexity - O(1)
- Pop
  - Time complexity - O(1)
- Peek
  - Time complexity - O(1)
- isEmpty()
  - Time complexity - O(1)