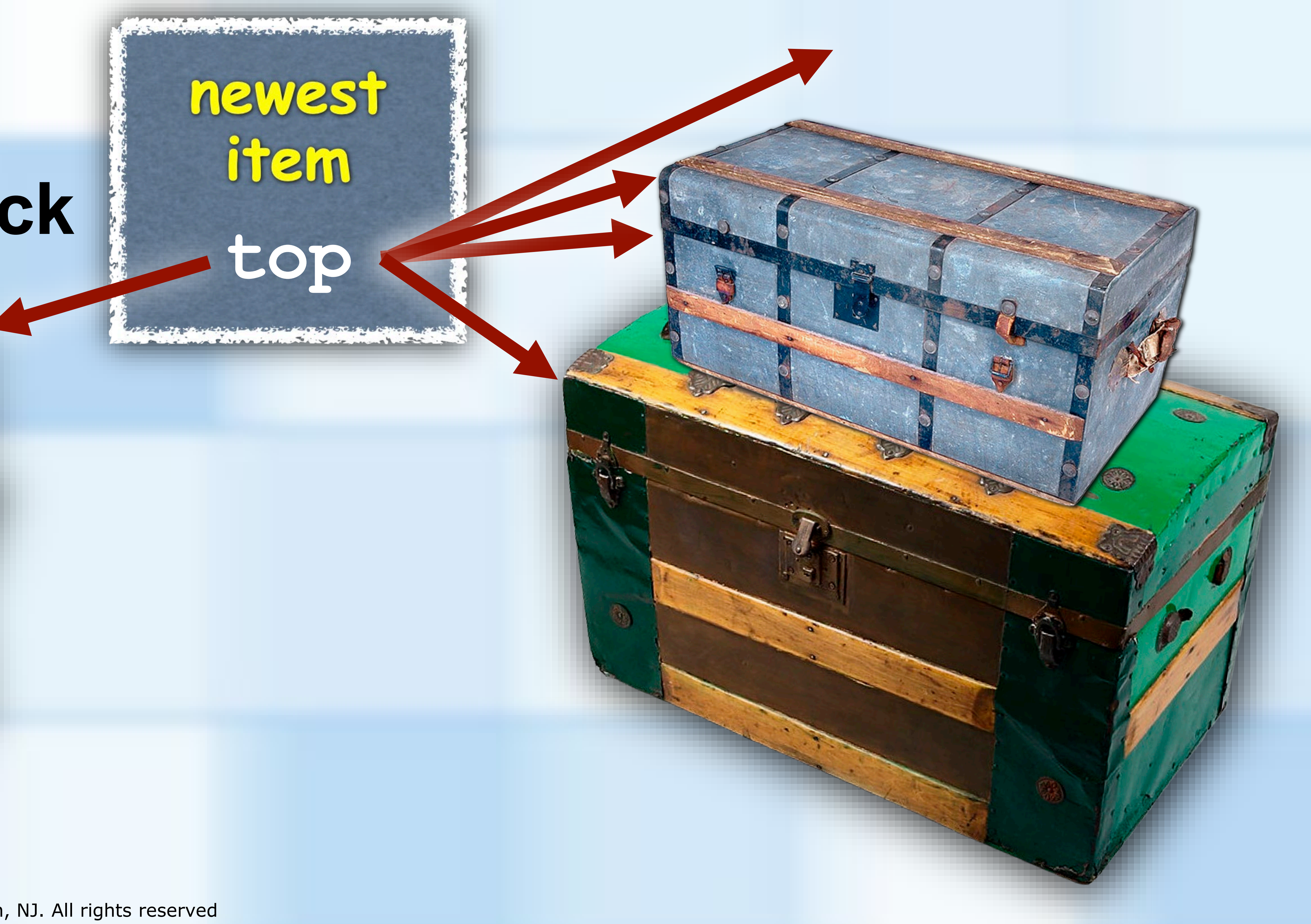
The ADT Stack

The Stack Concept

- Stack Characteristics
  - Last-in, first-out (LIFO) property
  - Last item placed on stack will be first item removed
  - Items placed and removed on top of stack
- Some Real Life Stacks
  - Books on a desk
  - Dishes in a Cafeteria
  - Boxes in an attic



The ADT Stack

- Collection of objects in reverse chronological order with same data type
- ADT Stack operations
  - Add a new item to the stack

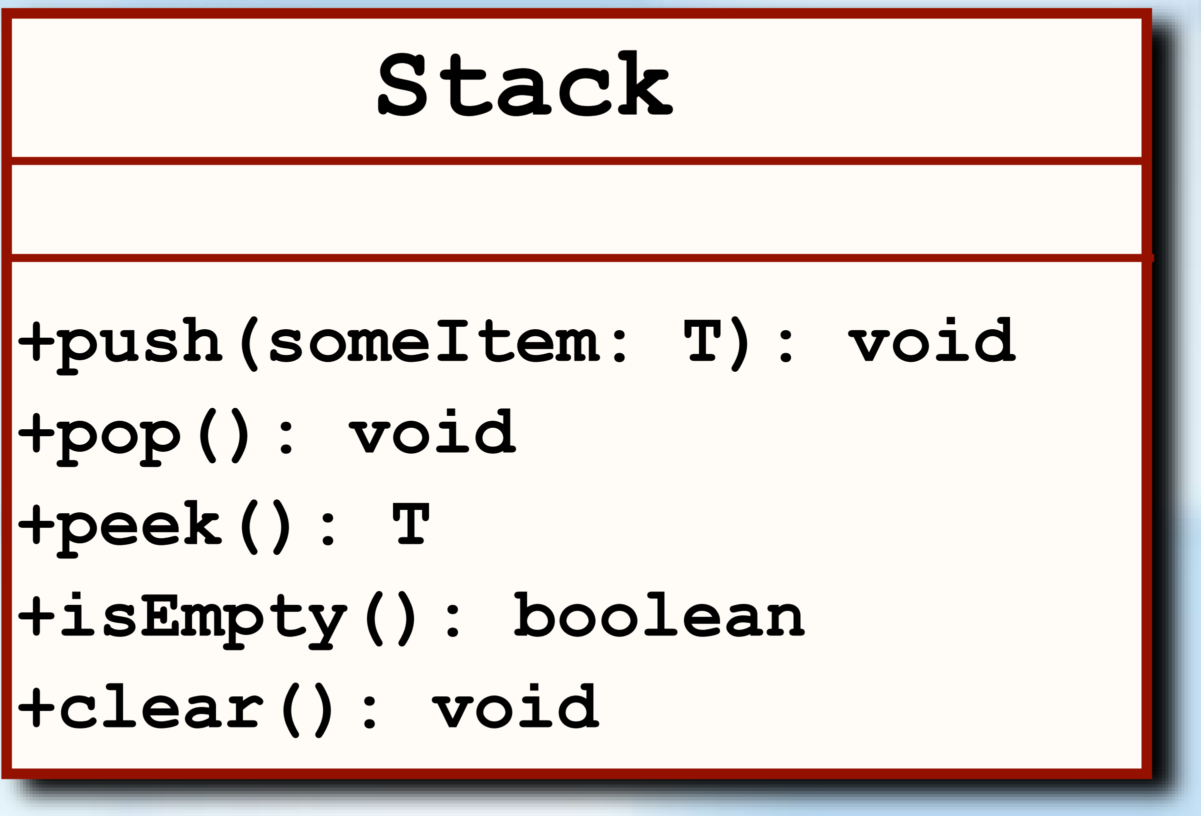
    push(ItemType someItem)
  - Remove item that was added most recently

    pop()
  - Retrieve item that was added most recently

    ItemType peek()
  - Determine whether a stack is empty

    boolean isEmpty()
  - Remove all entries from the stack

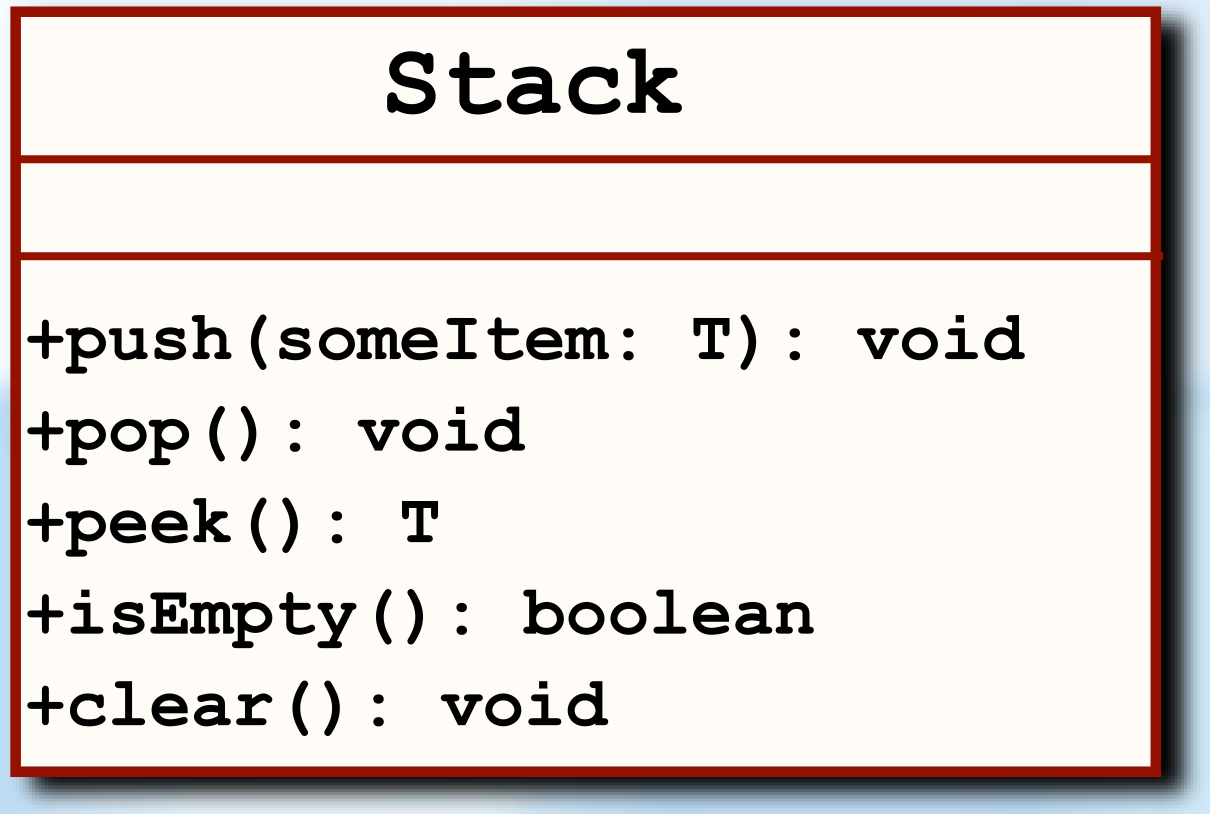
pop()
peek()



Stack

+push(someItem: T): void
+pop(): void
+peek(): T
+isEmpty(): boolean
+clear(): void

The ADT Stack



```
Stack

+push(someItem: T): void
+pop(): void
+peek(): T
+isEmpty(): boolean
+clear(): void
```

```cpp
/** @file StackInterface.h */

#ifndef STACKINTERFACE
#define STACKINTERFACE

template<class ItemType>
class StackInterface
{
public:

  virtual bool isEmpty() const = 0;

  virtual bool push(const ItemType& someItem) = 0;

  virtual bool pop() = 0;

  virtual ItemType peek() const = 0;

  virtual ~StackInterface() {   }
}; // end StackInterface

#endif
```

Using the ADT Stack

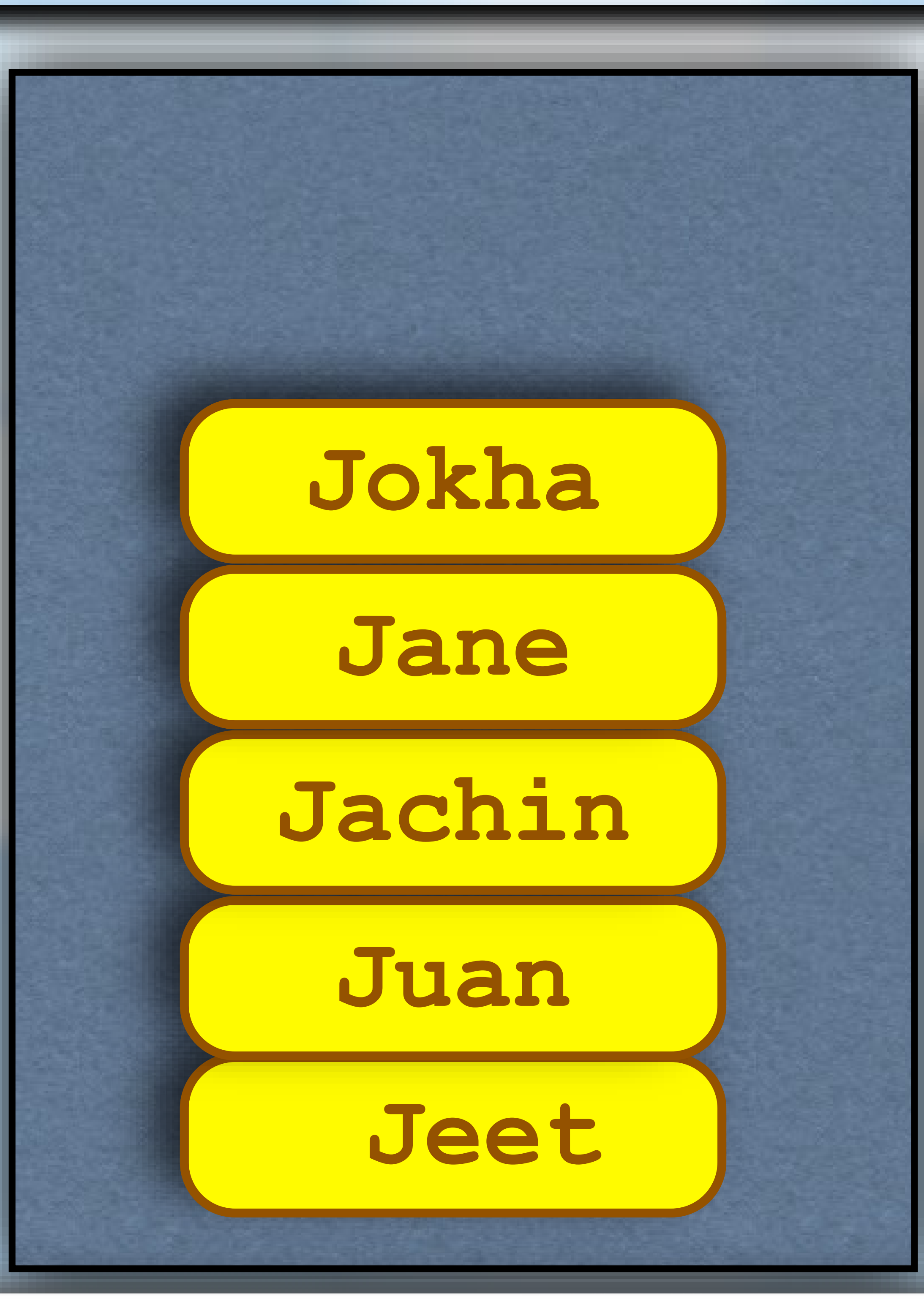
```
template<class ItemType>
class StackInterface  {
public:
  virtual bool isEmpty() const = 0;
  virtual bool push(const ItemType& someItem) = 0;
  virtual bool pop() = 0;
  virtual ItemType peek() const = 0;
  virtual ~StackInterface() {   }
```

```
/** @file StackInterface.h */

#ifndef STACKINTERFACE
#define STACKINTERFACE

template<class ItemType>
class StackInterface
{
public:

  virtual bool isEmpty() const = 0;

  virtual bool push(const ItemType& someItem) = 0;

  virtual bool pop() = 0;

  virtual ItemType peek() const = 0;

  virtual ~StackInterface() {   }
}; // end StackInterface

#endif
```

Pearson

Using the ADT Stac

```
template<class ItemType>
class StackInterface {
public:
  virtual bool isEmpty() const = 0;
  virtual bool push(const ItemType& someItem) = 0;
  virtual bool pop() = 0;
  virtual ItemType peek() const = 0;
  virtual ~StackInterface() {  }
}; // end StackInterface
```



```
Stack<std::string>* stringStack = new Stack<std::string>();
stringStack->push("Jeet");
stringStack->push("Juan");
stringStack->push("Jachin");
stringStack->push("Jane");
stringStack->push("Jokha");

std::string top = stringStack->peek();
std::cout << top << " is at the top of the stack.\n";

if (stringStack->pop())
   std::cout << top << " is removed from the stack.\n";

top = stringStack>peek();
std::cout << top << " is at the top of the stack.\n";

if (stringStack->pop())
     std::cout << top << " is removed from the stack.\n";
```

```
Jokha is at the top of the stack.
Jokha is removed from the stack.
Jane is at the top of the stack.
Jane is removed from the stack.
```

Balanced Expressions

Pearson

Balanced Expressions

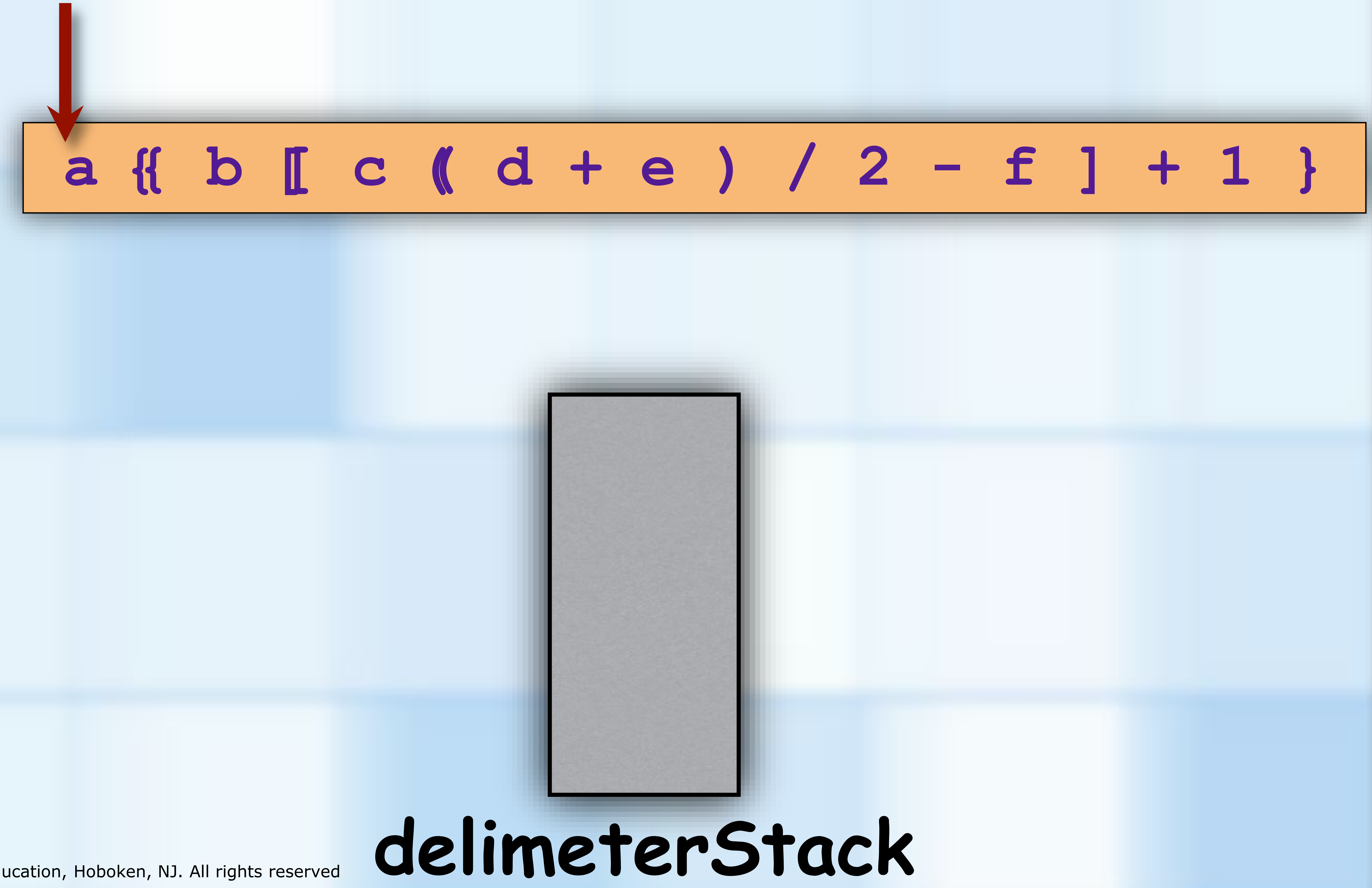
- Checking for Balanced Expressions
  - Scan expression:
    - Ignore characters that are not delimiters
  - When open delimiter is encountered
    - **push** it on the stack
  - When close delimiter is encountered
    - check to see if it matches top of stack
    - if yes, **pop** off top of stack
    - if not, expression is not balanced

Balanced

{ [( ) ( )] ( ) }

Not Balanced

[( ])



a { b [c (d + e) / 2 - f] + 1 }

delimeterStack

Balanced Expressions

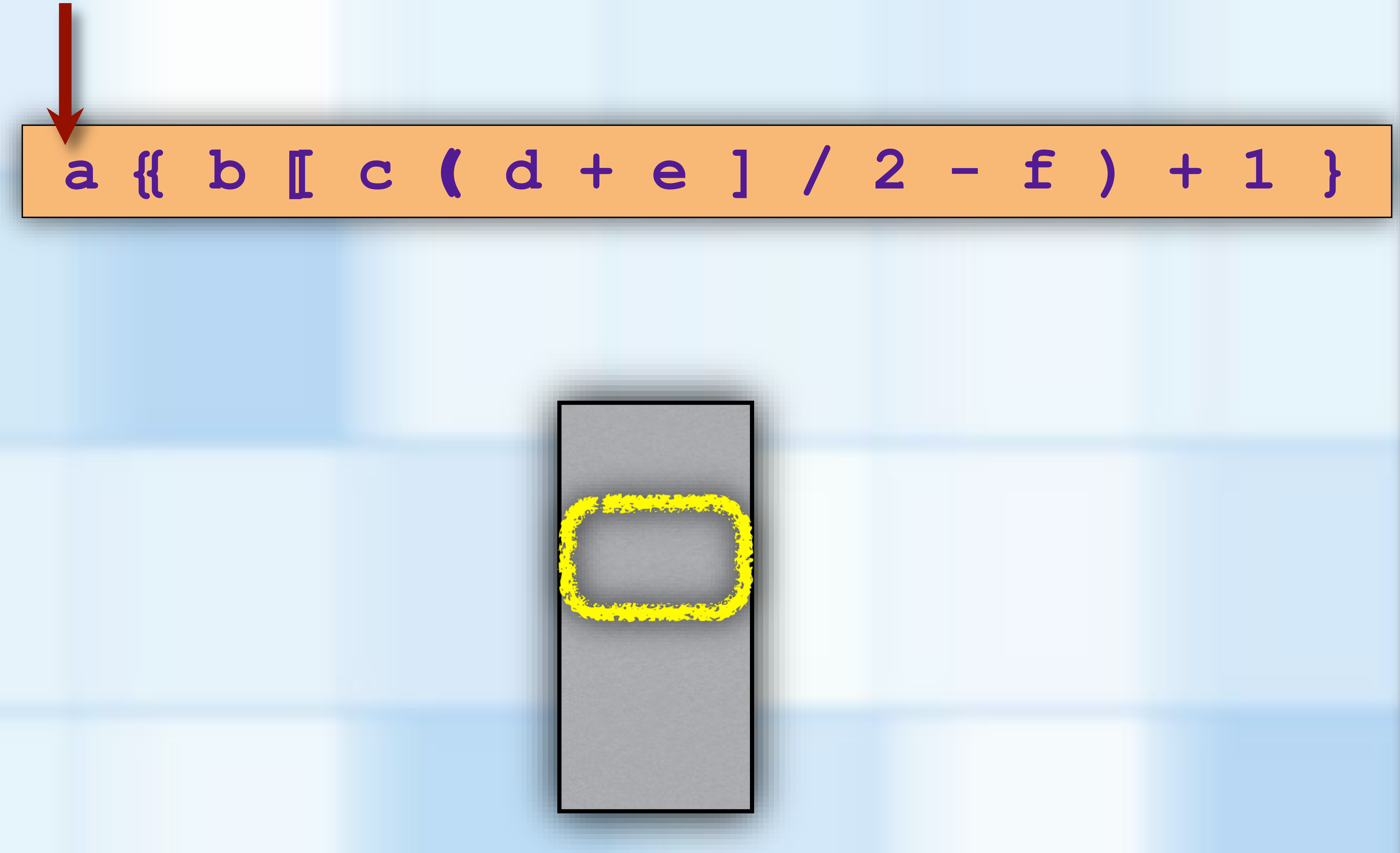
- **Checking for Balanced Expressions**
  - **Scan expression:**
    - Ignore characters that are not delimiters
  - **When open delimiter is encountered**
    - **push** it on the stack
  - **When close delimiter is encountered**
    - check to see if it matches top of stack
    - if yes, **pop** off top of stack
    - if not, expression is not balanced

Balanced

{ [( ) ( )] ( ) }

Not Balanced

[( ])



a { b [c (d + e] / 2 - f) + 1 }

Check Balance Implementation

```
bool checkBalance(string expression) {
  Stack<char> openDelimiterStack = new Stack<char>();
  int characterCount = expression.length();
  bool isBalanced = true;
  int index = 0;
  char nextCharacter = ' ';

 while (isBalanced && (index < characterCount)) {
   nextCharacter = expression.charAt(index);
   switch (nextCharacter){
    case '(': case '[': case '{':
      openDelimiterStack->push(nextC
      break;
    case ')': case ']': case '}':
      if (openDelimiterStack->isEmpt
        isBalanced = false;
      else
      {
        char openDelimiter = openDe
        openDelimiterStack->pop();
        isBalanced = isPaired(openDe
      } // end if
      break;
    default:
      break;
   } // end switch
  index++;
  } // end while
 if (!openDelimiterStack->isEmpty())
    isBalanced = false;
 return isBalanced;
} // end checkBalance
```

```
// Returns true if the given characters, open and close,
//form a pair of parentheses, brackets, or braces.
bool isPaired(char open, char close)
{
   return (open == '(' && close == ')') ||
          (open == '[' && close == ']') ||
          (open == '{' && close == '}');
} // end isPaired
```

Algebraic Expressions

Algebraic Expressions

- Operator Precedence

( )	Parenthesis
- +	Unary
^	Exponentiation
* / %	Multiplicative
+ -	Additive

- Binary Operators
  - Require two operands

4 + 5

- Unary Operators
  - Single operand

-6

- *Infix*
- *Common Notation*

5 + 6

5 + 6 * 7

(5 + 6) * 7

- *Prefix*
- *Functional Languages*

+ 5 6

+ * 7 6 5

* + 5 6 7

- *Postfix*
- *Reverse Polish Notation*

5 6 +

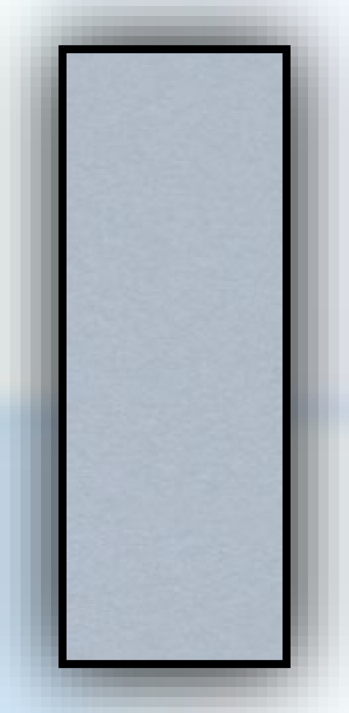
5 6 7 * +

7 5 6 + *

40 - 16* 8 ^ 3

5 + 6 * 7

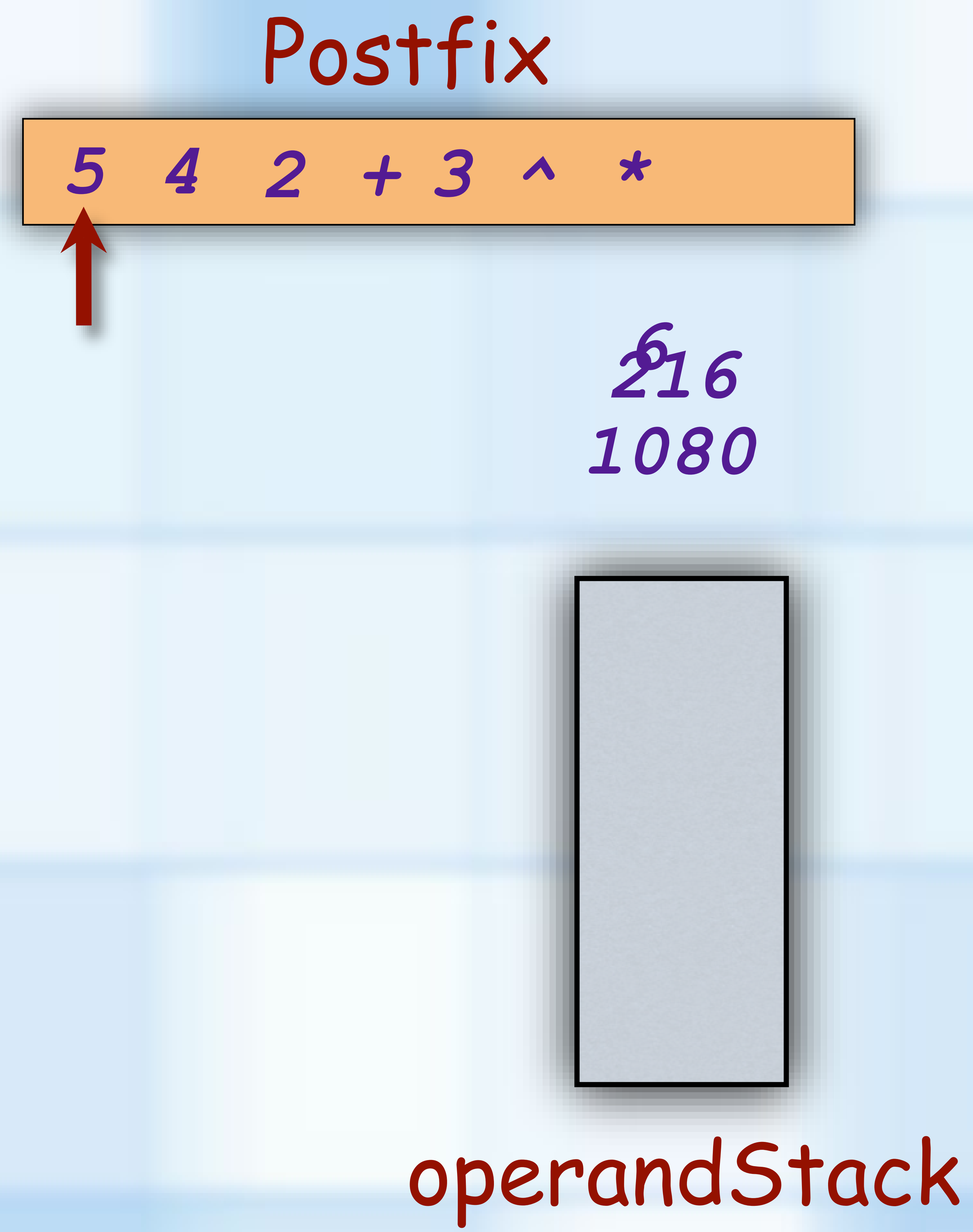
5 + 6 * 7



operatorStack

Evaluating Postfix Expressions

- **Scan characters in the Postfix Expression**
  - **When an *operand* is encountered,**
    - **push** it onto the **operandStack**
  - **When an *operator* is encountered,**
    - apply it to the top two ***operands*** of the **operandStack**
      - **pop** the ***operands*** from the operandStack
    - **push** the result of the operation onto the **operandStack**


Postfix
5 4 2 + 3 ^ *
6
216
1080
operandStack