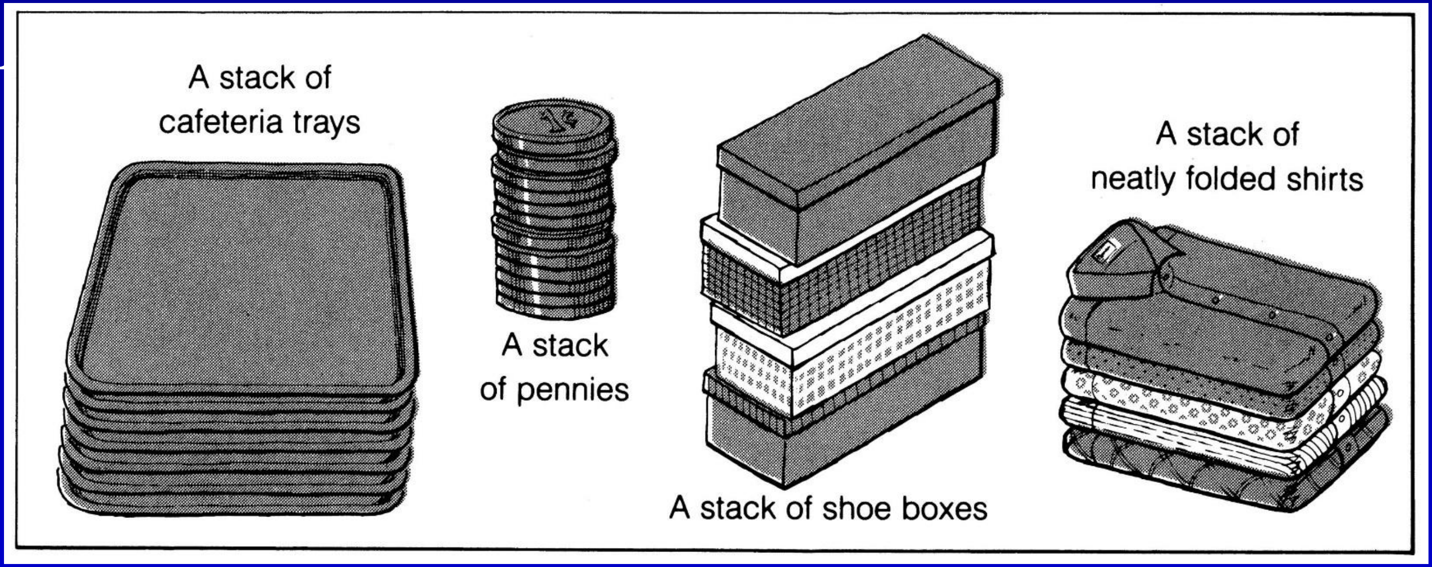
Stacks

CS 308 – Data Structures

What is a stack?

- It is an ordered group of homogeneous items of elements.
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).
- The last element to be added is the first to be removed (L



Stack Specification

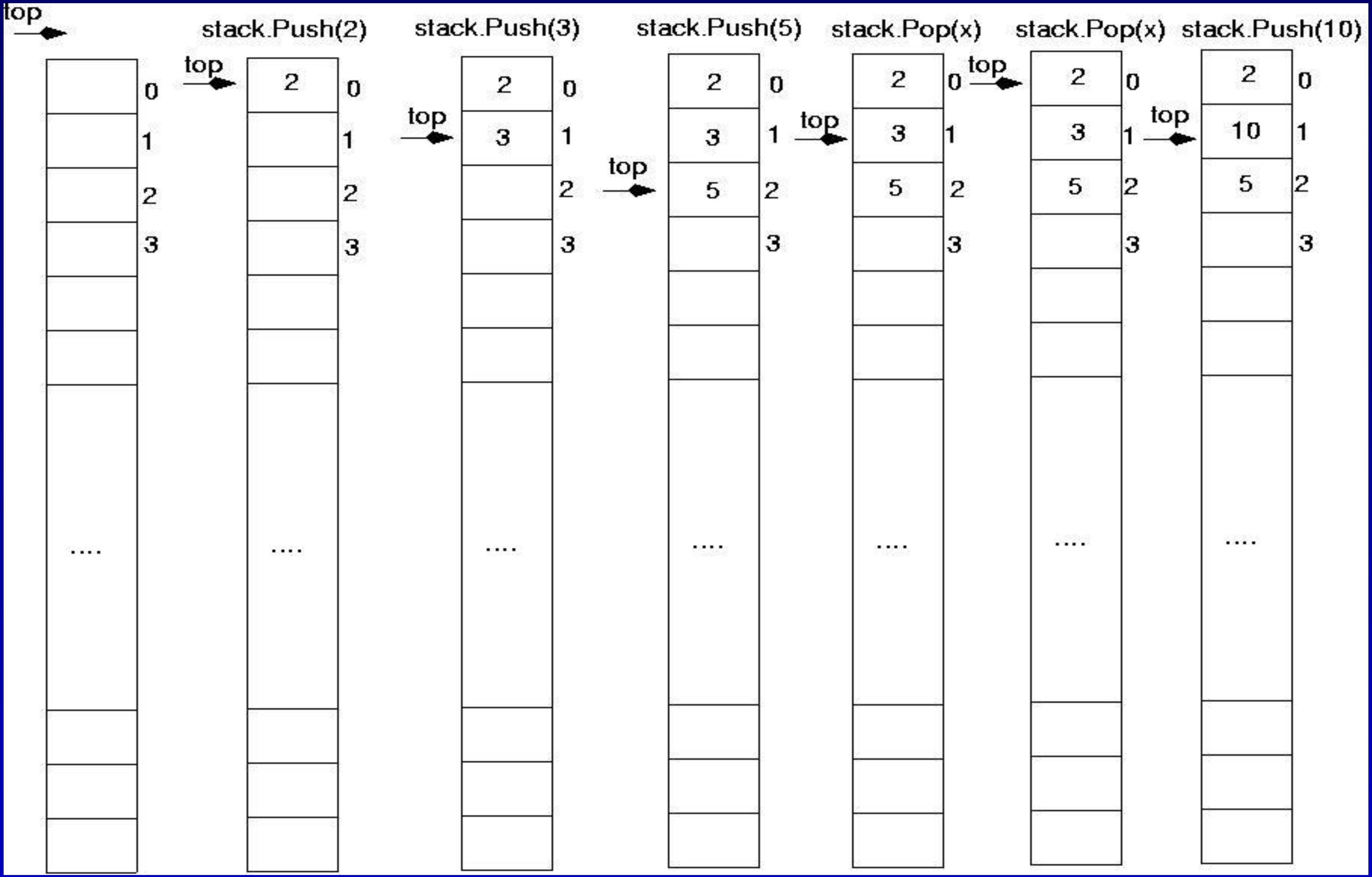
- Definitions: (provided by the user)
  - *MAX_ITEMS*: Max number of items that might be on the stack
  - *ItemType*: Data type of the items on the stack
- Operations
  - MakeEmpty
  - Boolean IsEmpty
  - Boolean IsFull
  - Push (ItemType newItem)
  - Pop (ItemType& item)

Push (ItemType newItem)

- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.

Pop (ItemType& item)

- *Function*: Removes topItem from stack and returns it in item.
- *Preconditions*: Stack has been initialized and is not empty.
- *Postconditions*: Top element has been removed from stack and item is a copy of the removed element.


top
0
1
2
3
....
stack.Push(2)
top
2
stack.Push(3)
top
3
stack.Push(5)
top
5
stack.Pop(x)
stack.Pop(x)
stack.Push(10)
10

Stack Implementation

```
#include "ItemType.h"
// Must be provided by the user of the class
// Contains definitions for MAX_ITEMS and ItemType

class StackType {
 public:
     StackType();
     void MakeEmpty();
     bool IsEmpty() const;
     bool IsFull() const;
     void Push(ItemType);
     void Pop(ItemType&);
private:
     int top;
     ItemType items[MAX_ITEMS];
};
```

Stack Implementation (cont.)

```
StackType::StackType()
{
 top = -1;
}

void StackType::MakeEmpty()
{
 top = -1;
}

bool StackType::IsEmpty() const
{
 return (top == -1);
}
```

Stack Implementation (cont.)

```
bool StackType::IsFull() const
{
 return (top == MAX_ITEMS-1);
}

 void StackType::Push(ItemType newItem)
{
 top++;
 items[top] = newItem;
}

 void StackType::Pop(ItemType& item)
{
 item = items[top];
 top--;
}
```

Stack overflow

- The condition resulting from trying to push an element onto a full stack.

```
if(!stack.IsFull())
    stack.Push(item);
```

Stack underflow

- The condition resulting from trying to pop an empty stack.

```
if(!stack.IsEmpty())
    stack.Pop(item);
```

Implementing stacks using templates

- Templates allow the compiler to generate multiple versions of a class type or a function by allowing parameterized types.
- It is similar to passing a parameter to a function (we pass a data type to a class !!)

Implementing stacks using templates

(cont.)

```
template<class ItemType>
class StackType {
 public:
    StackType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType);
    void Pop(ItemType&);
 private:
    int top;
    ItemType items[MAX_ITEMS];
};
```

Example using templates

```
// Client code
StackType<int> myStack;
StackType<float> yourStack;
StackType<StrType> anotherStack;

myStack.Push(35);
yourStack.Push(584.39);
```

The compiler generates distinct class types and gives its own internal name to each of the types.

Function templates

- The definitions of the member functions must be rewritten as function templates.

```
template<class ItemType>
StackType<ItemType>::StackType()
{
 top = -1;
}

template<class ItemType>
void StackType<ItemType>::MakeEmpty()
{
 top = -1;
}
```

Function templates (cont.)

```
template<class ItemType>
bool StackType<ItemType>::IsEmpty() const
{
 return (top == -1);
}

template<class ItemType>
bool StackType<ItemType>::IsFull() const
{
 return (top == MAX_ITEMS-1);
}

template<class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
 top++;
 items[top] = newItem;
}
```

Function templates (cont.)

```
template<class ItemType>
void StackType<ItemType>::Pop(ItemType& item)
{
 item = items[top];
 top--;
}
```

Comments using templates

- The *template<class T>* designation must precede the class method name in the source code for each template class method.
- The word *class* is required by the syntax of the language and does not mean that the actual parameter must be the name of a class.
- Passing a parameter to a template <u>has an effect at compile time</u>.

Implementing stacks using dynamic array allocation

```
template<class ItemType>
class StackType {
 public:
    StackType(int);
    ~StackType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType);
    void Pop(ItemType&);
private:
    int top;
    int maxStack;
    ItemType *items;
};
```

Implementing stacks using dynamic array allocation (cont.)

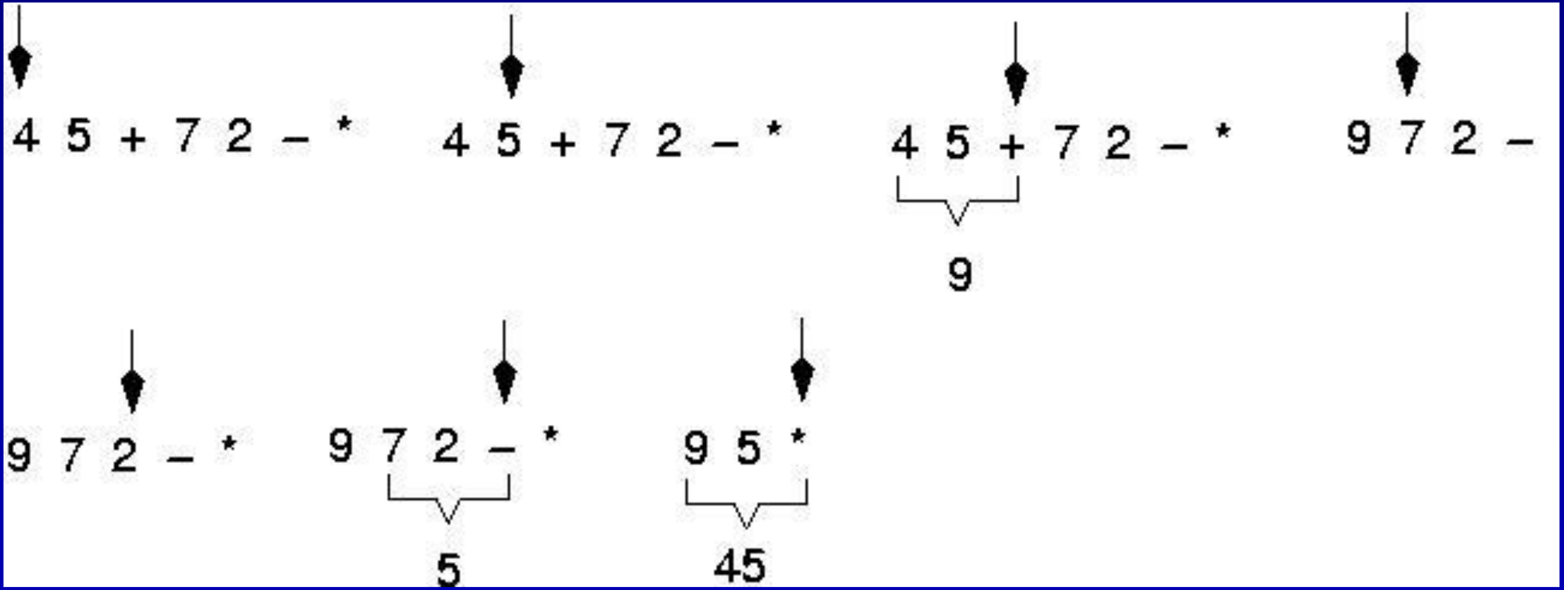
```
template<class ItemType>
StackType<ItemType>::StackType(int max)
{
 maxStack = max;
 top = -1;
 items = new ItemType[max];
}

template<class ItemType>
StackType<ItemType>::~StackType()
{
 delete [ ] items;
}
```

Example: postfix expressions

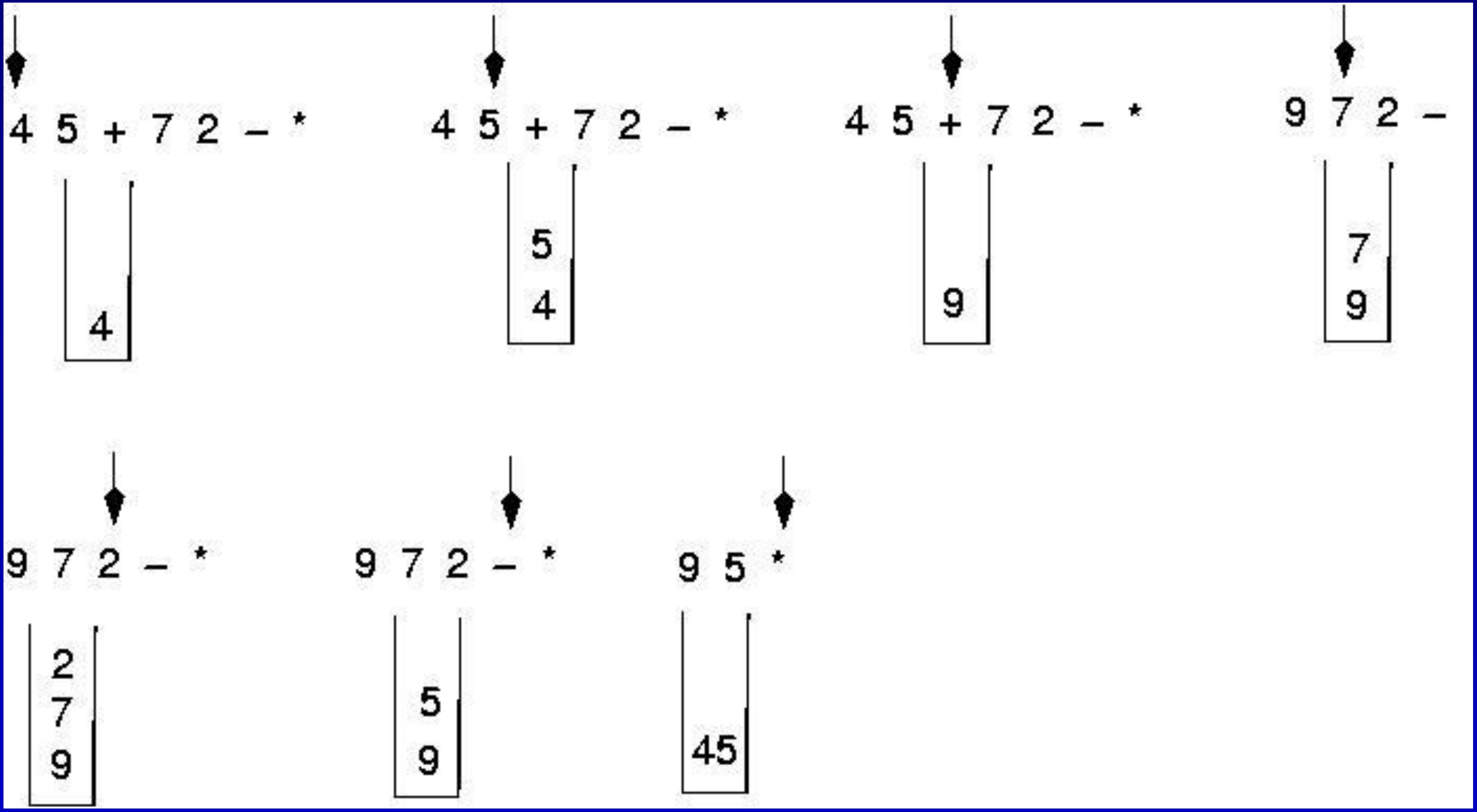
- Postfix notation is another way of writing arithmetic expressions.
- In postfix notation, the operator is written after the two operands.

  infix: 2+5   *postfix*: 2 5 +
- Expressions are evaluated from left to right.
- Precedence rules and parentheses are never needed!!

Example: postfix expressions (cont.)



Postfix expressions: Algorithm using stacks (cont.)



Postfix expressions: Algorithm using stacks

```
WHILE more input items exist
    Get an item
    IF item is an operand
      stack.Push(item)
    ELSE
      stack.Pop(operand2)
      stack.Pop(operand1)
      Compute result
      stack.Push(result)
stack.Pop(result)
```

Write the body for a function that replaces each copy of an item in a stack with another item. Use the following specification. (this function is a client program).

ReplaceItem(StackType& stack, ItemType oldItem, ItemType newItem)

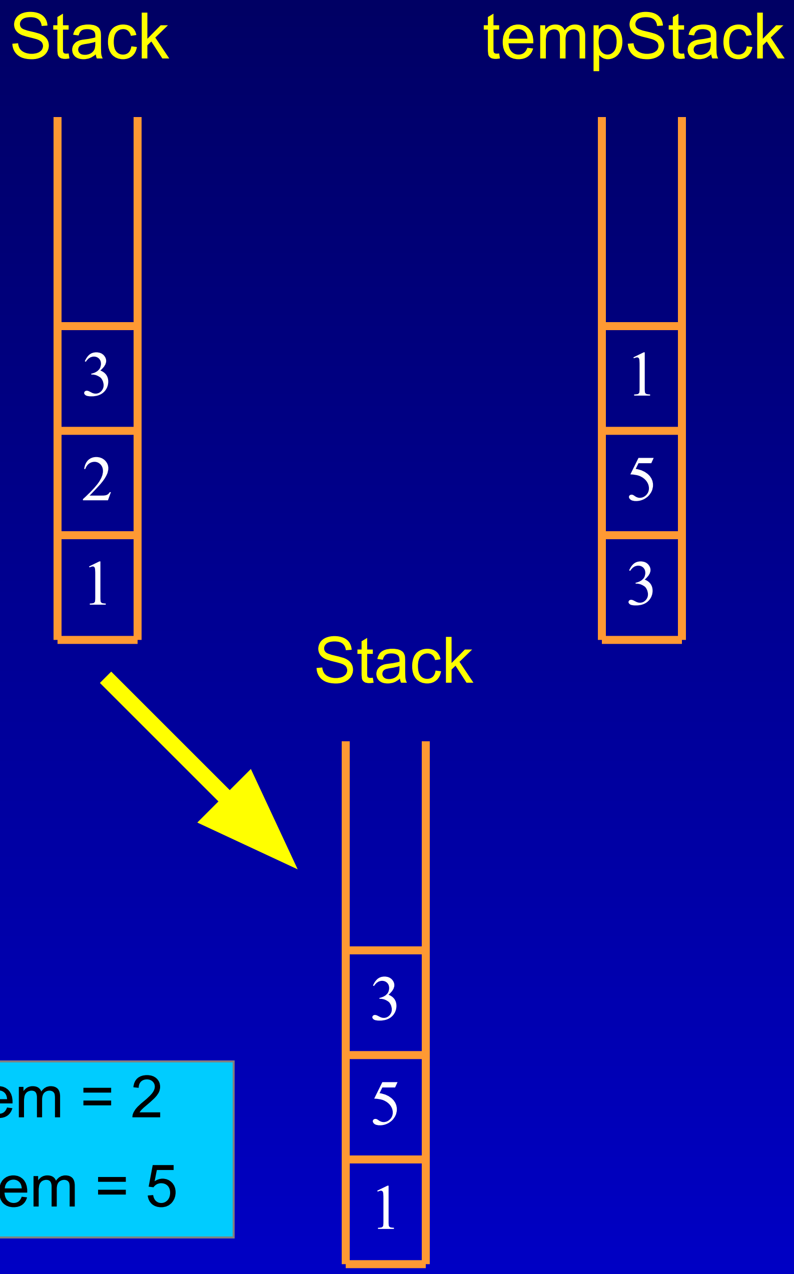
Function: Replaces all occurrences of oldItem with newItem.

Precondition: stack has been initialized.

Postconditions: Each occurrence of oldItem in stack has been replaced by newItem.

(You may use any of the member functions of the StackType, but you may not assume any knowledge of how the stack is implemented).

```
{
  ItemType item;
  StackType tempStack;

  while (!Stack.IsEmpty())  {
  Stack.Pop(item);
  if (item==oldItem)
     tempStack.Push(newItem);
  else
     tempStack.Push(item);
  }
  while (!tempStack.IsEmpty()) {
  tempStack.Pop(item);
  Stack.Push(item);
  }
}
```



oldItem = 2

newItem = 5

Exercises

- 1, 3-7, 14, 12, 15, 18, 19