

CS-202

# Dynamic Data Structures (Pt.4)

C. Papachristos  
Autonomous Robots Lab  
University of Nevada, Reno



# Course Week

## Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (4 Sections)	
	CLASS	RL – Session	CLASS	
PASS Session	PASS Session	Project DEADLINE	NEW Project	

Your 8<sup>th</sup> Project Deadline is this Wednesday 4/18.

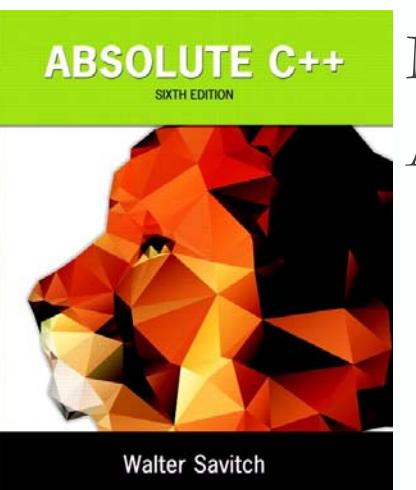
- PASS Sessions held Monday-Tuesday }
- RL Session held Wednesday } get all the help you may need!
  
- 24-hrs delay after Project Deadline incurs 20% grade penalty.
- Past that, NO Project accepted. Better send what you have in time!

# Today's Topics

## Dynamic Data Structures

### Stack(s)

- Array-based
- List-based
- Traversal
- Insertion
- Deletion
- Search



Modified Slides from:

Absolute C++

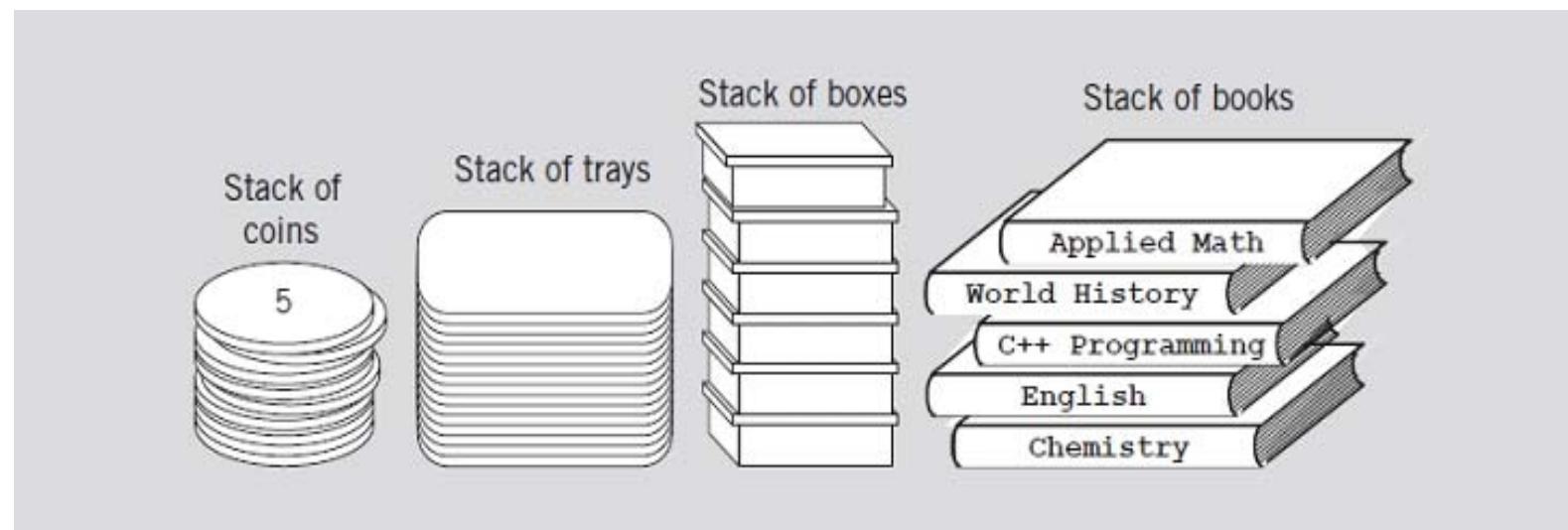
6th Edition

Walter Savitch

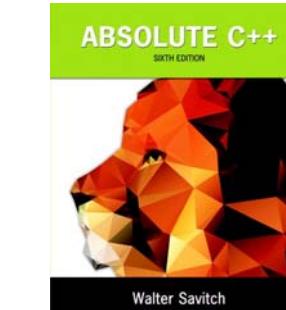
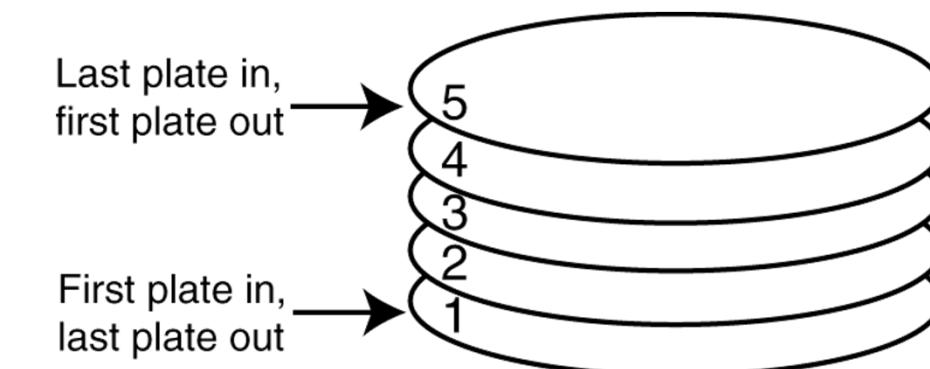
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# What is a Stack?

- Data structure
  - that has homogeneous items of elements.
  - Elements are added and removed from the top of the stack
  - The last element to be added is the first to be removed  
**LIFO: Last In, First Out**

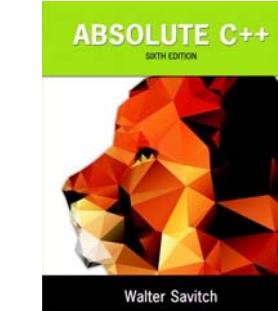
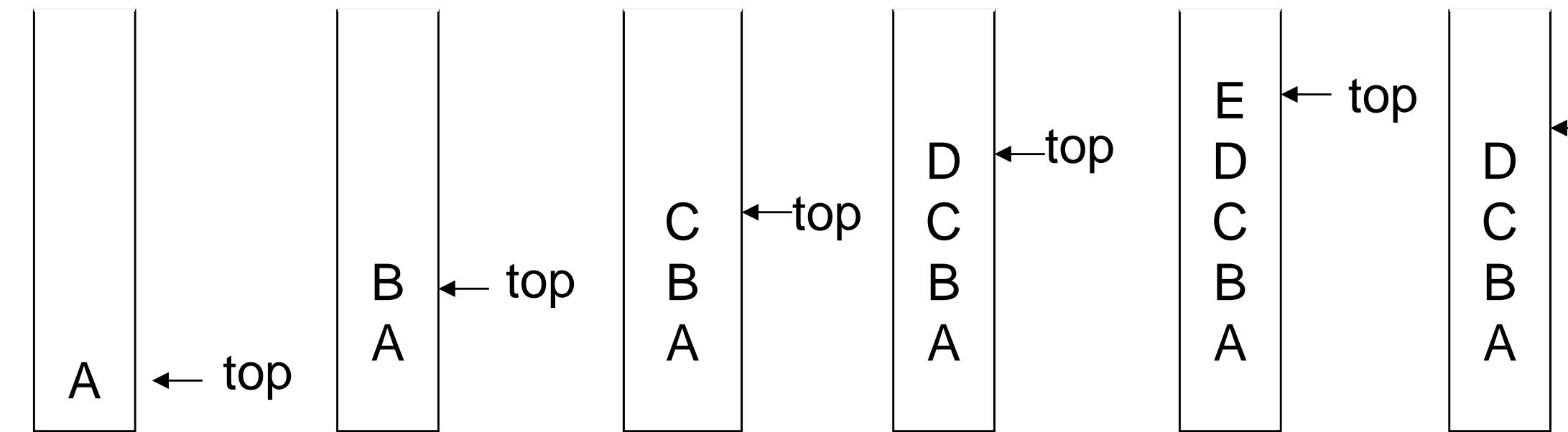


**FIGURE 7-1** Various examples of stacks



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

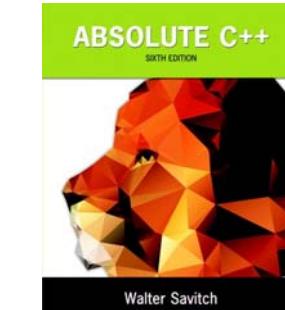
# Last In First Out



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

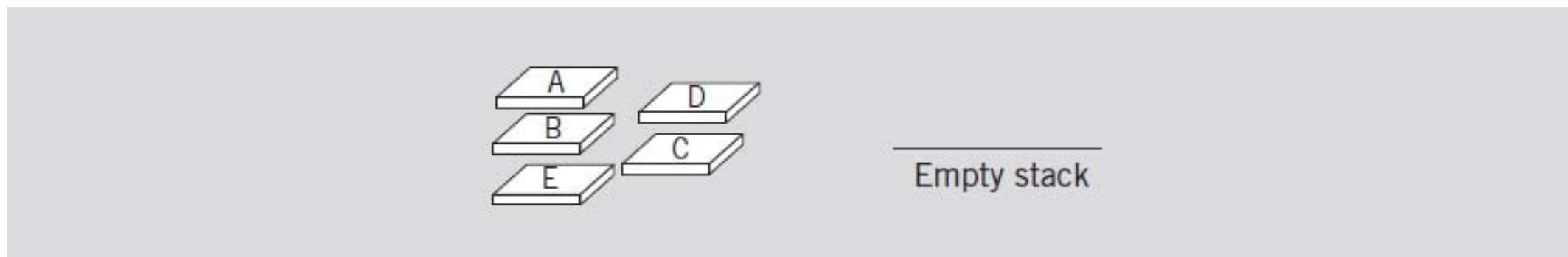
# Stacks

- Some operations on a stack
  - push operation
    - Add element onto the stack
  - top operation
    - Retrieve top element of the stack
  - pop operation
    - Remove top element from the stack

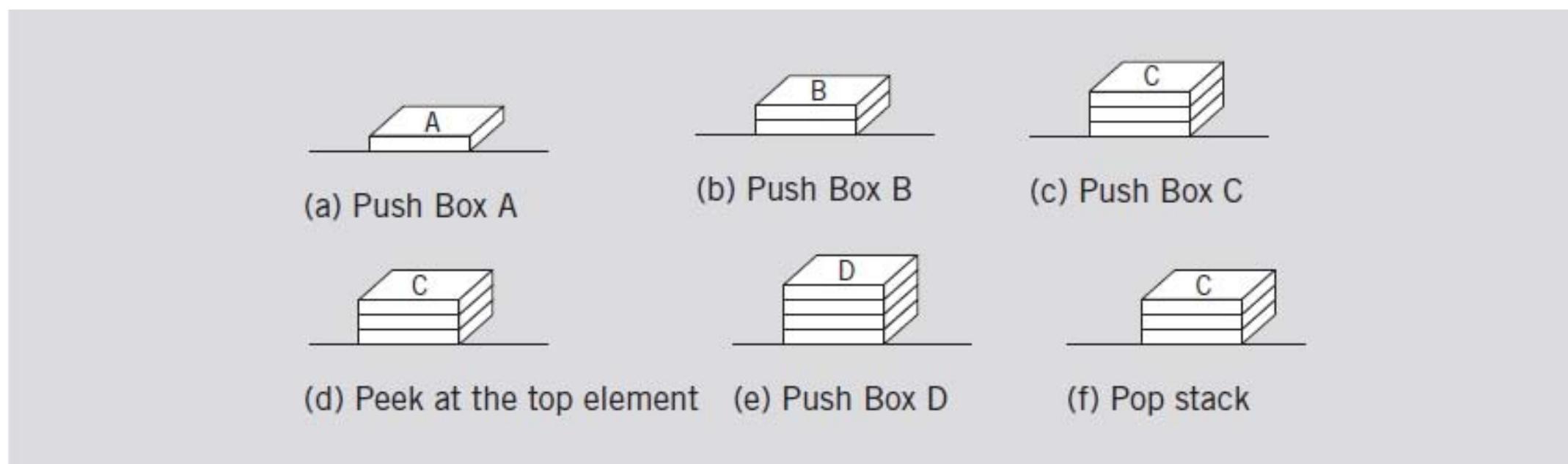


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

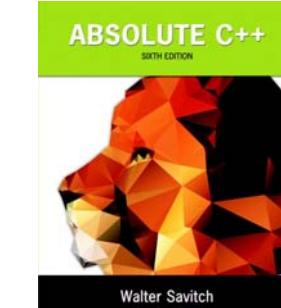
# Stacks (cont'd.)



**FIGURE 7-2** Empty stack

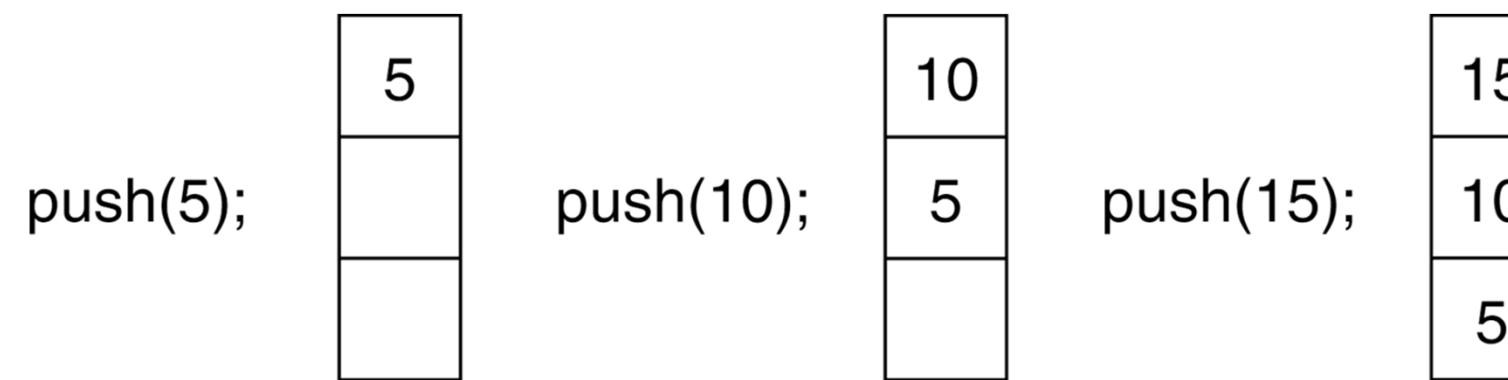


**FIGURE 7-3** Stack operations

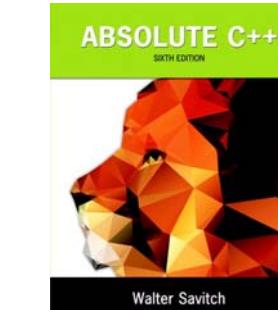
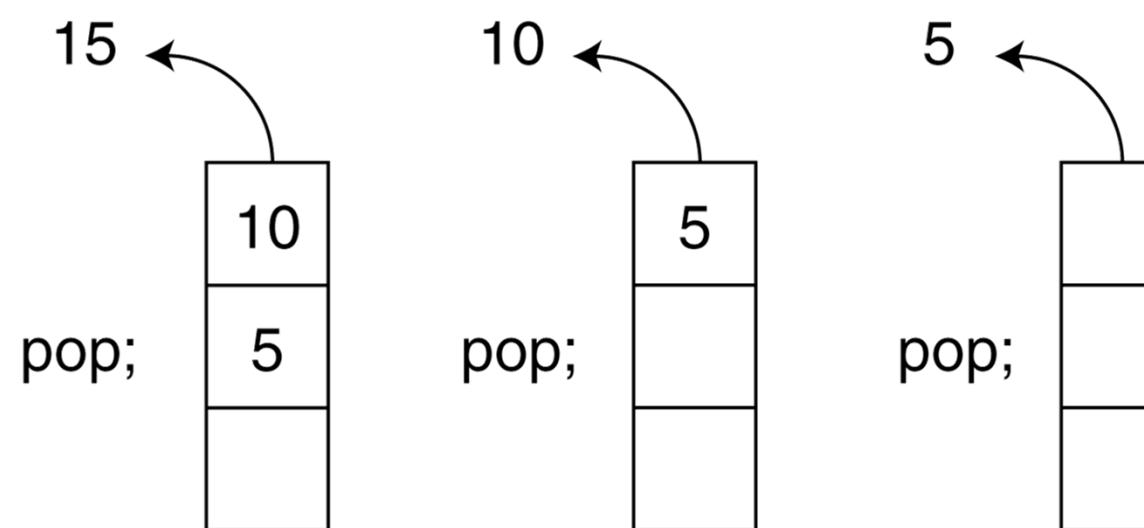


# Push and Pop Example

The state of the stack after each of the push operations:



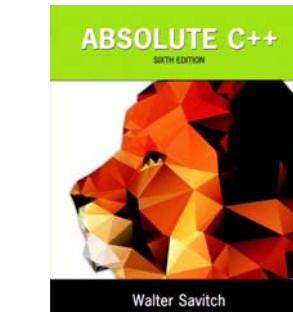
Now, suppose we execute three consecutive pop operations on the same stack:



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

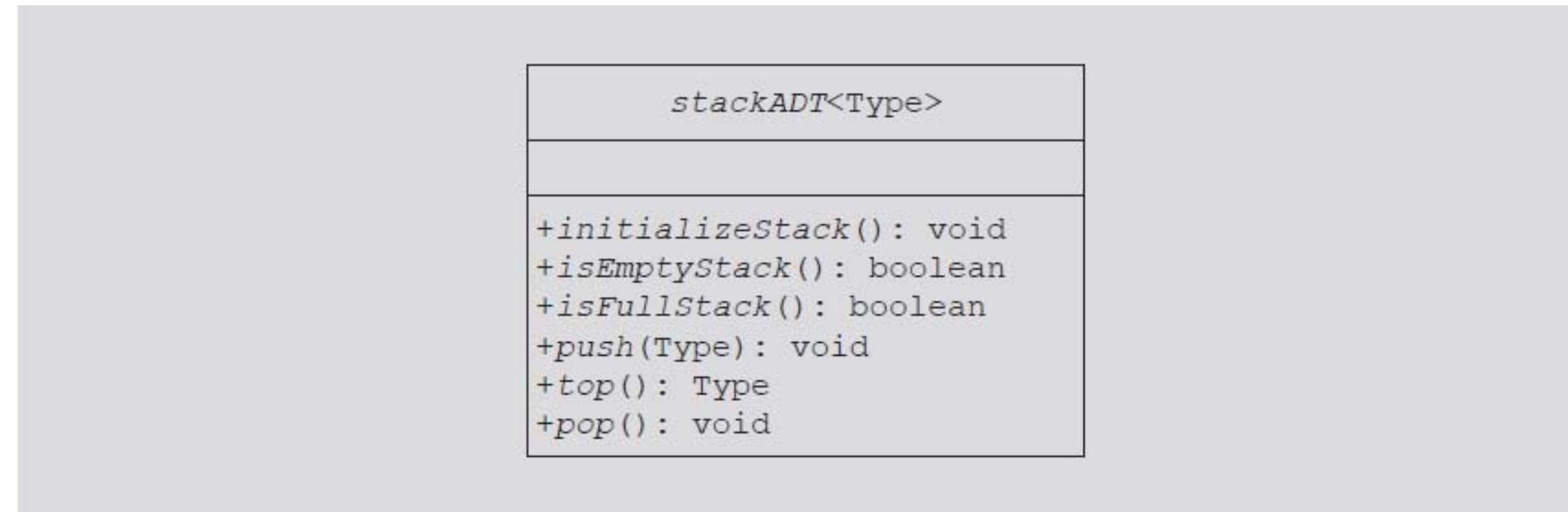
# Stacks (cont'd.)

- Stack element removal
  - Occurs only if something is in the stack
- Stack element added only if room available
- isFullStack operation
  - Checks for full stack
- isEmptyStack operation
  - Checks for empty stack
- initializeStack operation
  - Initializes stack to an empty state

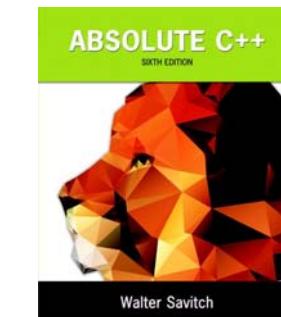


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Stacks (cont'd.)



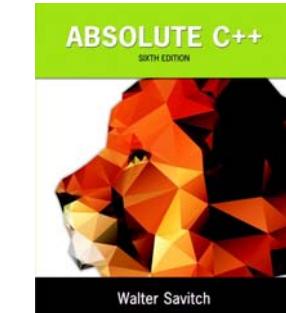
**FIGURE 7-4** UML class diagram of the class `stackADT`



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

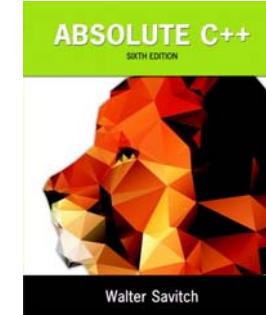
# Static and Dynamic Stacks

- Static Stacks
  - Fixed size
  - Can be implemented with an array
- Dynamic Stacks
  - Grow in size as needed
  - Can be implemented with a linked list



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

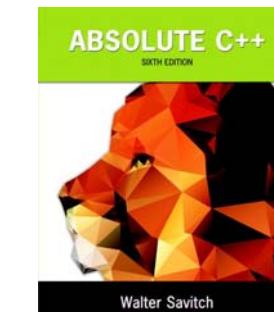
# Implementation of Stacks as Arrays



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Implementation of Stacks as Arrays

- Advantages
  - best performance
- Disadvantage
  - fixed size
- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false
    - otherwise adds it into the correct spot
  - if array is empty, pop() returns null
    - otherwise removes the next item in the stack



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

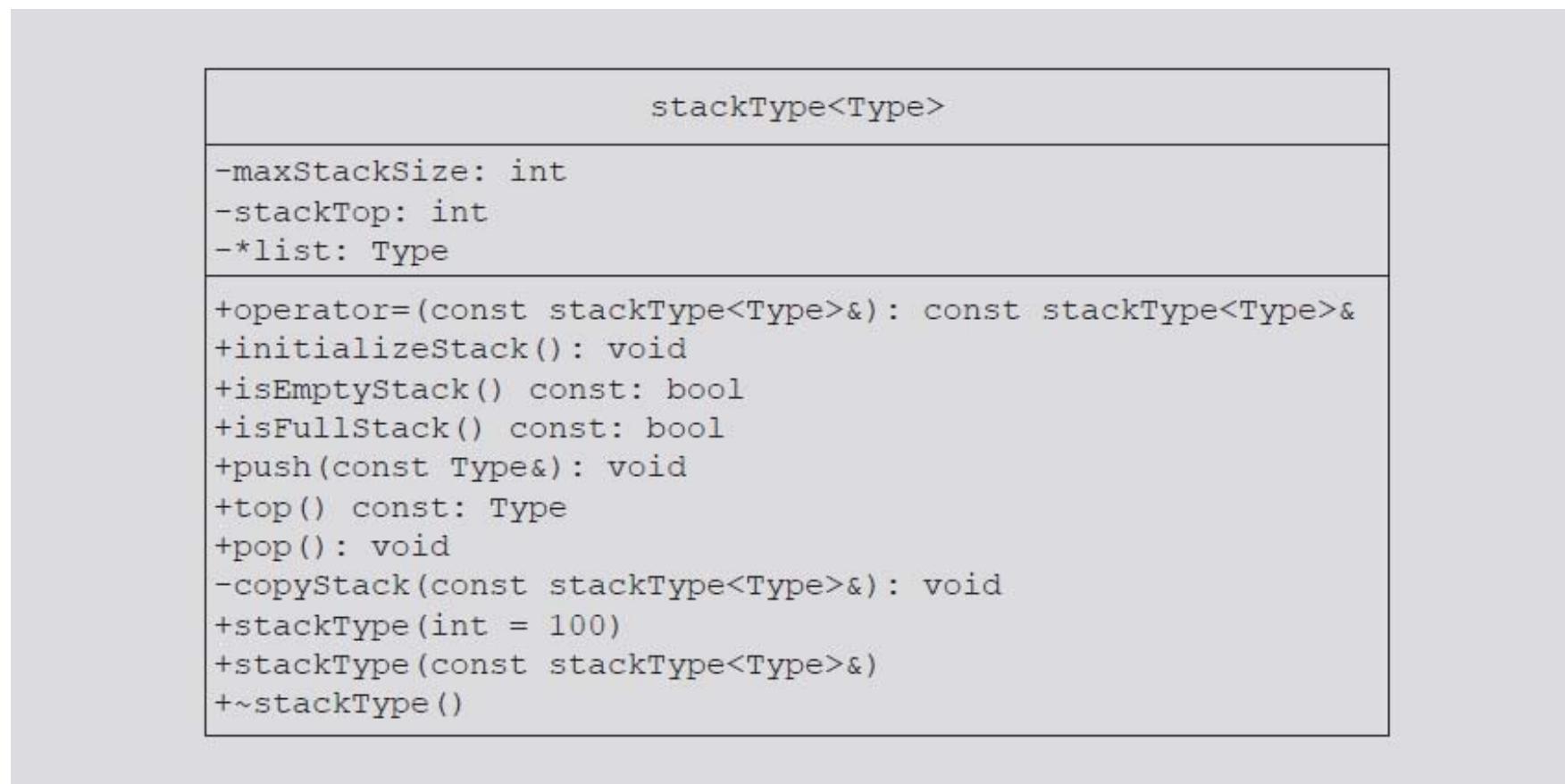
# Implementation of Stacks as Arrays

- First stack element
  - Put in first array slot
- Second stack element
  - Put in second array slot, and so on
- Top of stack
  - Index of last element added to stack
- Stack element accessed only through the top
  - Problem: array is a random access data structure
  - Solution: use another variable (`stackTop`)
    - Keeps track of the top position of the array

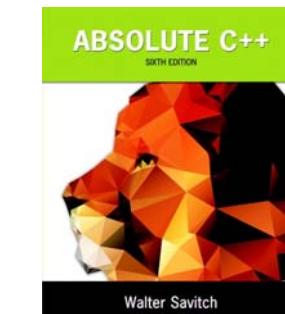


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Implementation of Stacks as Arrays (cont'd.)

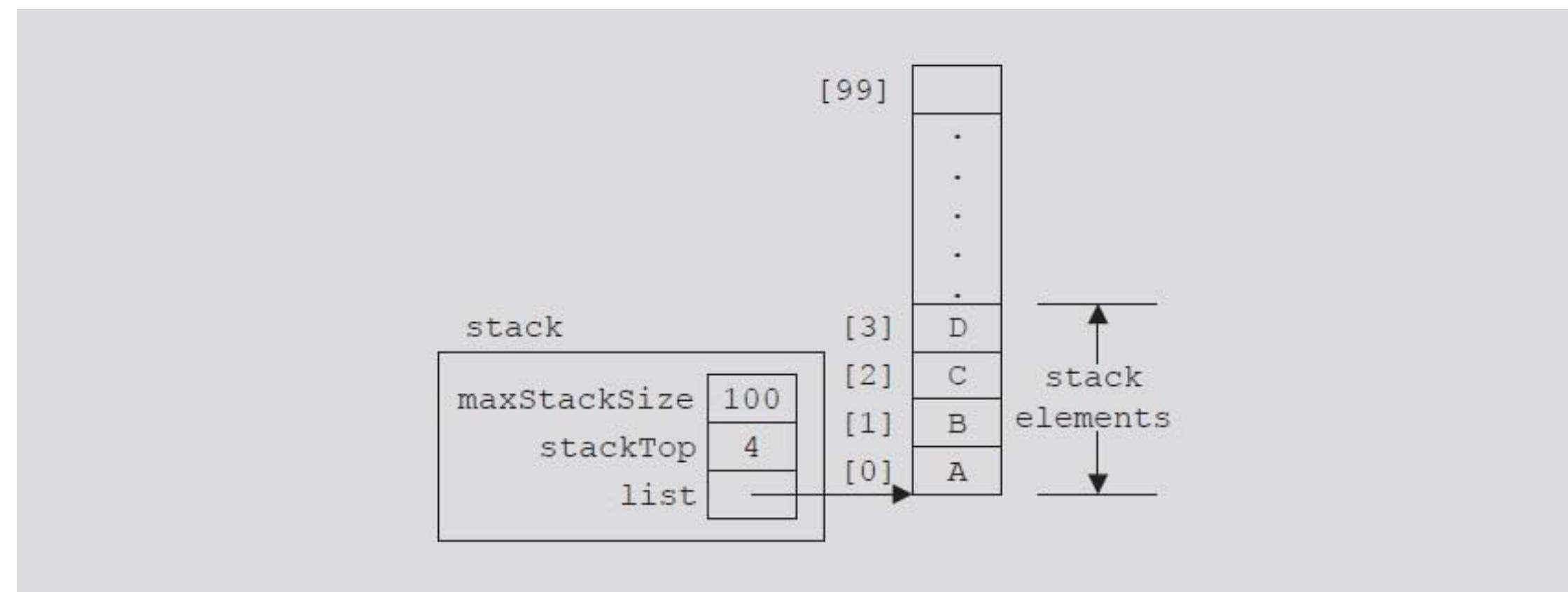


**FIGURE 7-5** UML class diagram of the class `stackType`

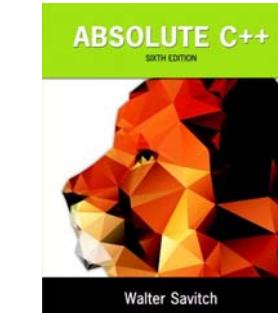


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Implementation of Stacks as Arrays (cont'd.)



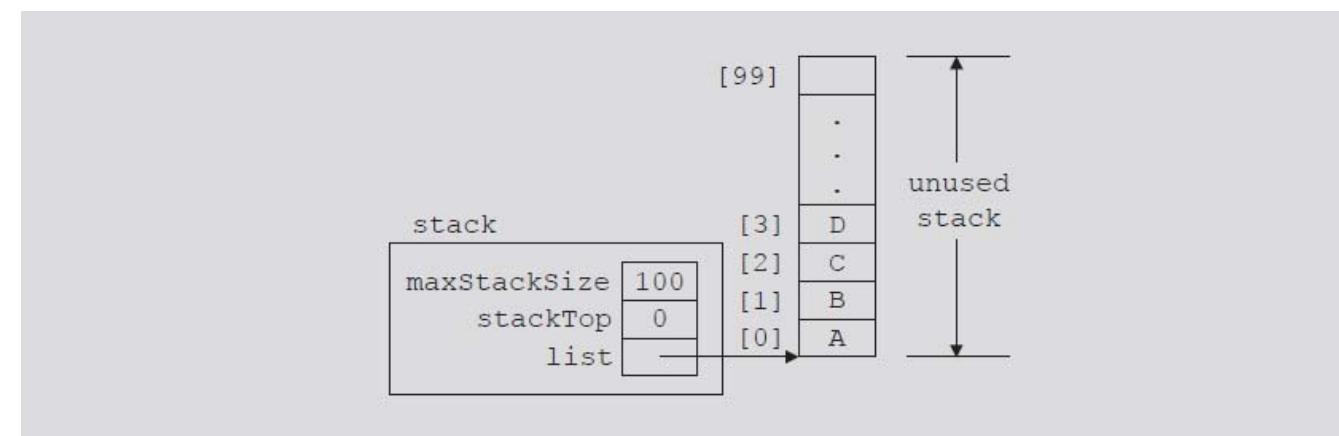
**FIGURE 7-6** Example of a stack



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

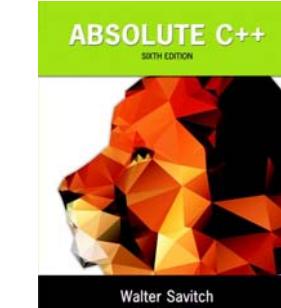
# Initialize Stack

- Value of stackTop if stack empty
  - Set stackTop to zero to initialize the stack
- Definition of function initializeStack



**FIGURE 7-7** Empty stack

```
template <class Type>
void stackType<Type>::initializeStack()
{
    stackTop = 0;
}//end initializeStack
```

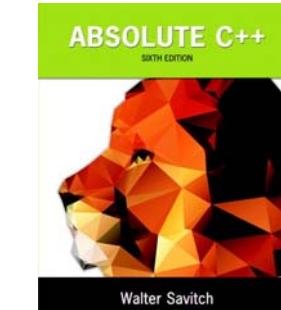


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Empty Stack

- Value of stackTop indicates if stack empty
  - If stackTop = zero: stack empty
  - Otherwise: stack not empty
- Definition of function isEmptyStack

```
template <class Type>
bool stackType<Type>::isEmptyStack() const
{
    return(stackTop == 0);
}//end isEmptyStack
```

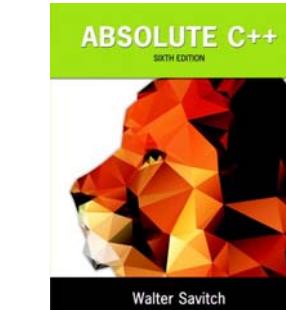


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Full Stack

- Stack full
  - If stackTop is equal to maxStackSize
- Definition of function `isFullStack`

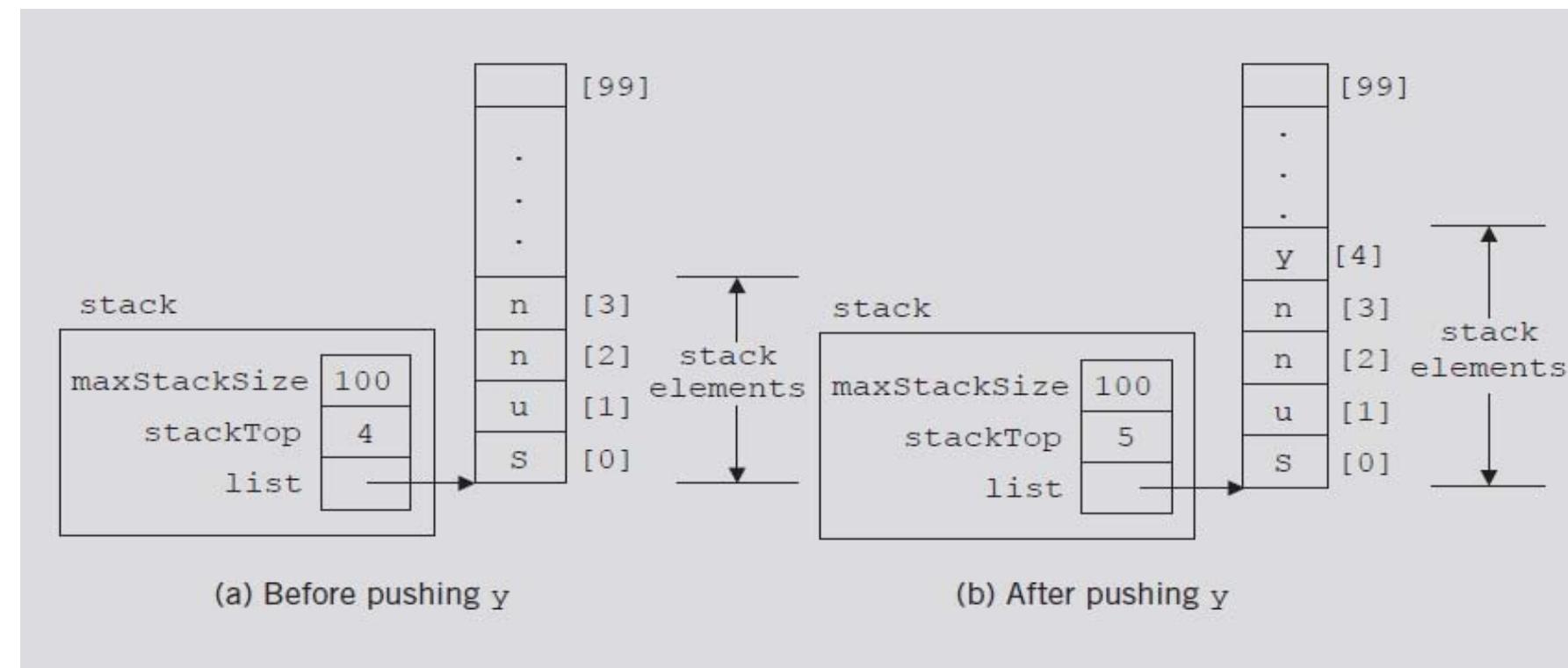
```
template <class Type>
bool stackType<Type>::isFullStack() const
{
    return(stackTop == maxStackSize);
} //end isFullStack
```



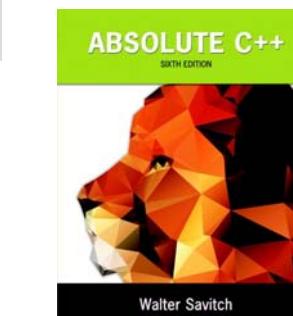
Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Push

- Two-step process
  - Store newItem in array component indicated by stackTop
  - Increment stackTop



**FIGURE 7-8** Stack before and after the push operation

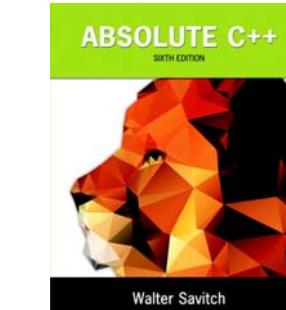


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Push (cont'd.)

- Definition of push operation

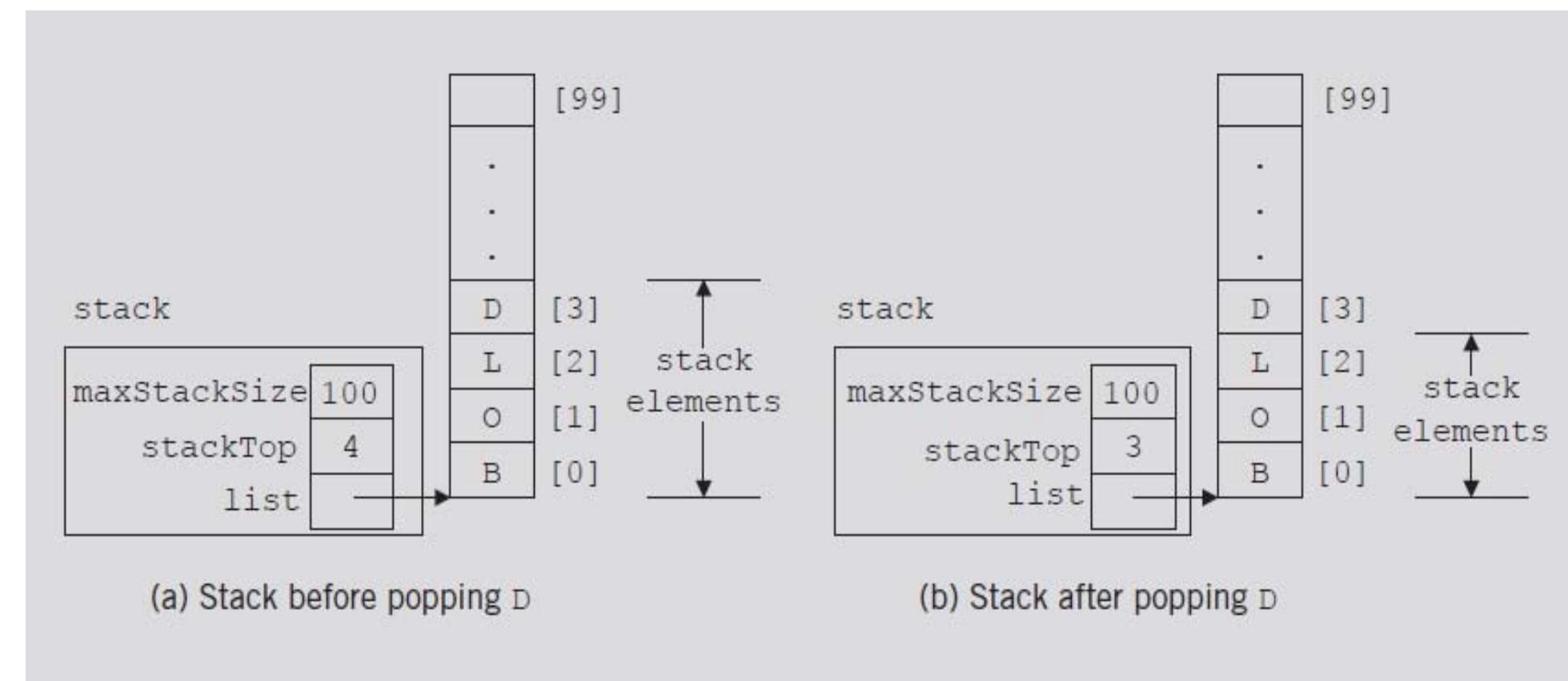
```
template <class Type>
void stackType<Type>::push (const Type& newItem)
{
    if (!isFullStack ())
    {
        list[stackTop] = newItem; //add newItem at the top
        stackTop++; //increment stackTop
    }
    else
        cout << "Cannot add to a full stack." << endl;
}//end push
```



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Pop

- Remove (pop) element from stack
  - Decrement stackTop by one



**FIGURE 7-9** Stack before and after the pop operation

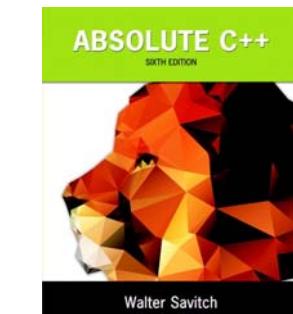


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Pop (cont'd.)

- Definition of pop operation
- Underflow
  - Removing an item from an empty stack
    - Check within pop operation (see below)
    - Check before calling function pop

```
template <class Type>
void stackType<Type>::pop()
{
    if (!isEmptyStack())
        stackTop--; //decrement stackTop
    else
        cout << "Cannot remove from an empty stack." << endl;
}//end pop
```



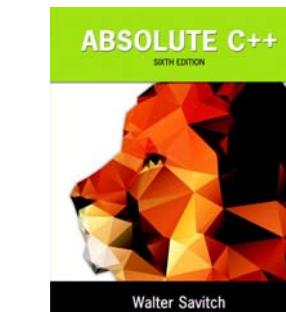
Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Copy Stack

- Definition of function `copyStack`

```
template <class Type>
void stackType<Type>::copyStack(const stackType<Type> & otherStack)
{
    delete [] list;
    maxStackSize = otherStack.maxStackSize;
    stackTop = otherStack.stackTop;
    list = new Type[maxStackSize];

    //copy otherStack into this stack
    for (int j = 0; j < stackTop; j++)
        list[j] = otherStack.list[j];
} //end copyStack
```



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

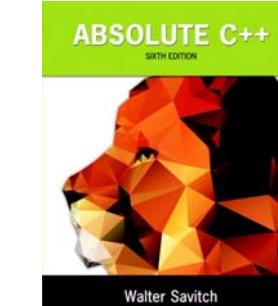
# Constructor and Destructor

```
template <class Type>
stackType<Type>::stackType(int stackSize)
{
    if (stackSize <= 0)
    {
        cout << "Size of the array to hold the stack must "
            << "be positive." << endl;
        cout << "Creating an array of size 100." << endl;

        maxStackSize = 100;
    }
    else
        maxStackSize = stackSize;      //set the stack size to
                                        //the value specified by
                                        //the parameter stackSize

    stackTop = 0;                  //set stackTop to 0
    list = new Type[maxStackSize]; //create the array to
                                    //hold the stack elements
} //end constructor

template <class Type>
stackType<Type>::~stackType() //destructor
{
    delete [] list; //deallocate the memory occupied
                    //by the array
} //end destructor
```



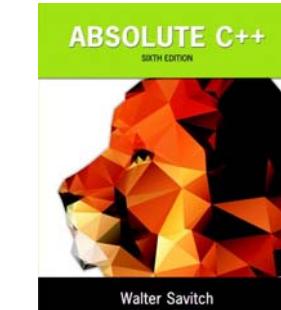
Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Copy Constructor

- Definition of the copy constructor

```
template <class Type>
stackType<Type>::stackType (const stackType<Type>& otherStack)
{
    list = NULL;

    copyStack(otherStack);
}//end copy constructor
```



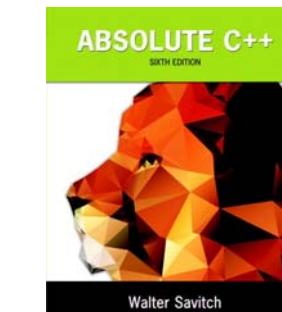
Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Overloading the Assignment Operator (=)

- Classes with pointer member variables
  - Assignment operator must be explicitly overloaded
- Function definition to overload assignment

```
template <class Type>
const stackType<Type>& stackType<Type>::operator=
    (const stackType<Type>& otherStack)
{
    if (this != &otherStack) //avoid self-copy
        copyStack(otherStack);

    return *this;
} //end operator=
```



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Stack Header File

- myStack.h
  - Header file name containing class stackType definition

```
//Header file: myStack.h

#ifndef H_StackType
#define H_StackType

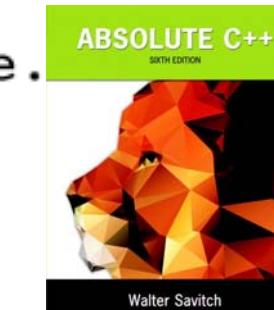
#include <iostream>
#include <cassert>

#include "stackADT.h"

using namespace std;

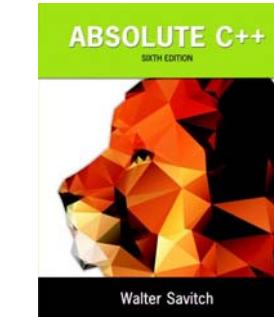
//Place the definition of the class template stackType, as given
//previously in this chapter, here.

//Place the definitions of the member functions as discussed here.
#endif
```



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

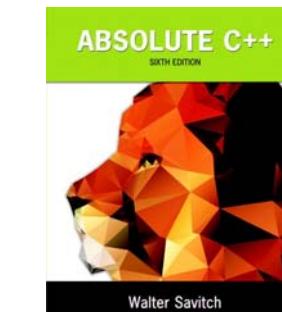
# Implementation of Stacks as Linked List



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Linked Implementation of Stacks

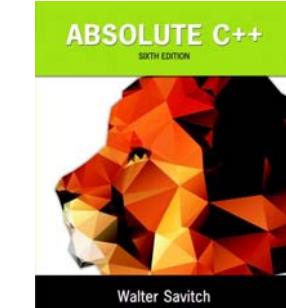
- Disadvantage of array (linear) stack representation
  - Fixed number of elements can be pushed onto stack
- Solution
  - Use pointer variables to dynamically allocate, deallocate memory
  - Use linked list to dynamically organize data
- Value of `stackTop`: linear representation
  - Indicates number of elements in the stack
    - Gives index of the array
  - Value of `stackTop - 1`
    - Points to top item in the stack



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Implementing a Stack: Linked List

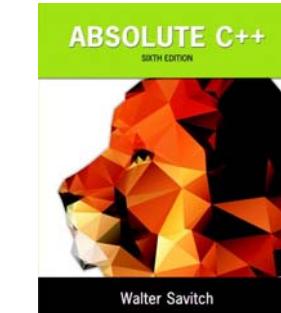
- Advantages:
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantages
  - the common case is the slowest of all the implementations
  - can grow to an infinite size
- Basic implementation
  - list is initially empty
  - *push()* method adds a new item to the head of the list
  - *pop()* method removes the head of the list



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Linked Implementation of Stacks (cont'd.)

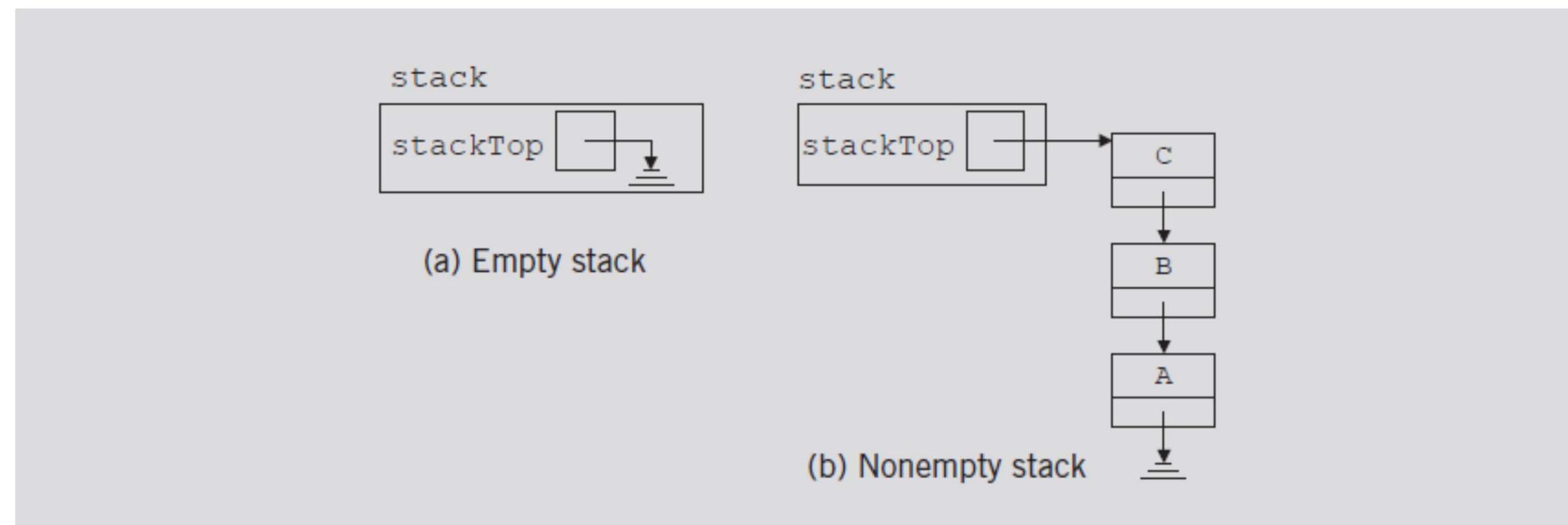
- Value of `stackTop`: linked representation
  - Locates top element in the stack
    - Gives address (memory location) of the top element of the stack



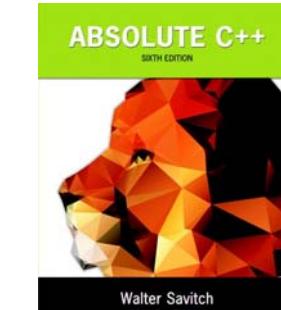
Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Linked Implementation of Stacks (cont'd.)

- Example 7-2
  - Stack: object of type `linkedStackType`



**FIGURE 7-10** Empty and nonempty linked stacks

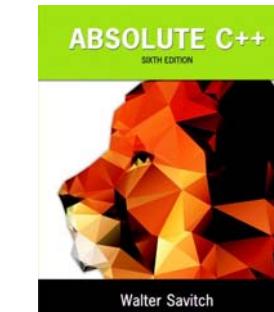


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Default Constructor

- When stack object declared
  - Initializes stack to an empty state
  - Sets `stackTop` to `NULL`
- Definition of the default constructor

```
template <class Type>
linkedStackType<Type>::linkedStackType()
{
    stackTop = NULL;
}
```

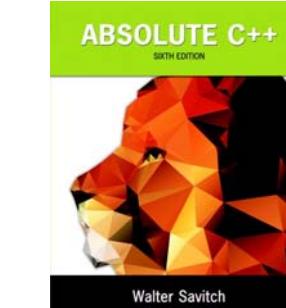


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Empty Stack and Full Stack

- Stack empty if stackTop is NULL
- Stack never full
  - Element memory allocated/deallocated dynamically

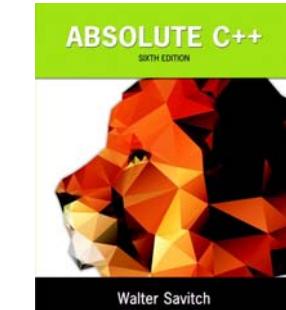
```
template <class Type>
bool linkedStackType<Type>::isEmptyStack() const
{
    return(stackTop == NULL);
} //end isEmptyStack
```
  - ```
template <class Type>
bool linkedStackType<Type>::isFullStack() const
{
    return false;
} //end isFullStack
```



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Initialize Stack

- Reinitializes stack to an empty state
- Because stack might contain elements and you are using a linked implementation of a stack
  - Must deallocate memory occupied by the stack elements, set stackTop to NULL
- Definition of the initializeStack function

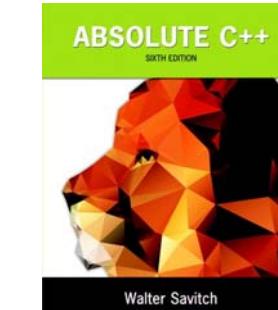


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Initialize Stack (cont'd.)

```
template <class Type>
void linkedStackType<Type>:: initializeStack()
{
    nodeType<Type> *temp; //pointer to delete the node

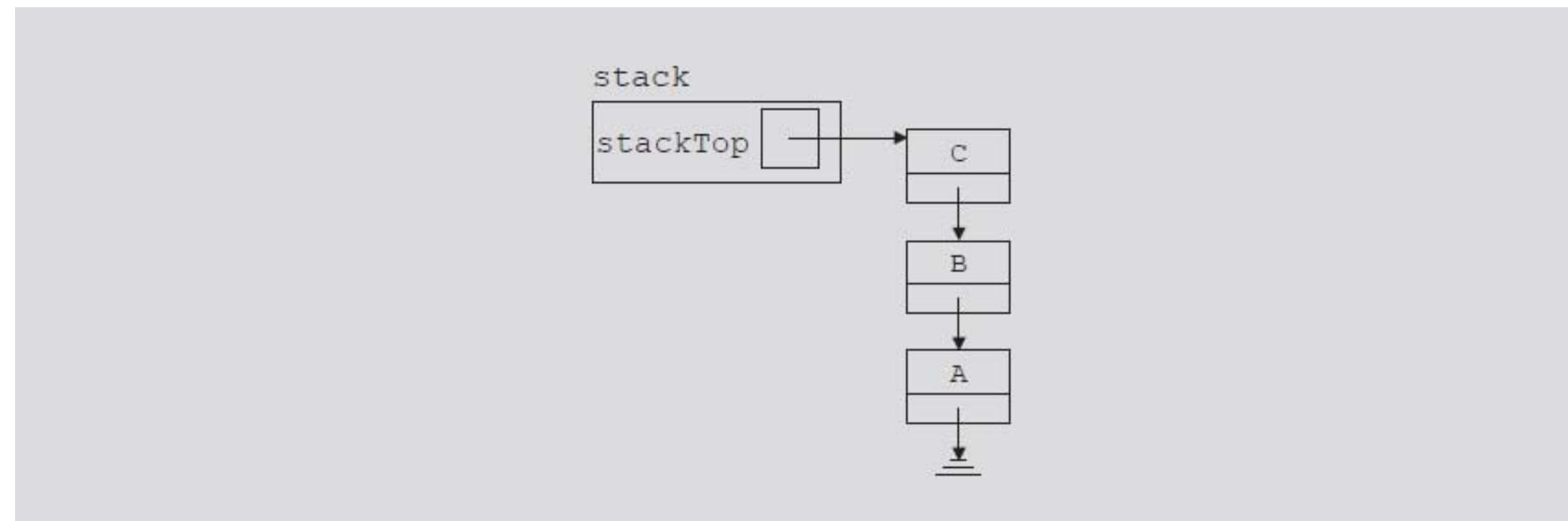
    while (stackTop != NULL) //while there are elements in
                            //the stack
    {
        temp = stackTop; //set temp to point to the
                           //current node
        stackTop = stackTop->link; //advance stackTop to the
                                   //next node
        delete temp; //deallocate memory occupied by temp
    }
} //end initializeStack
```



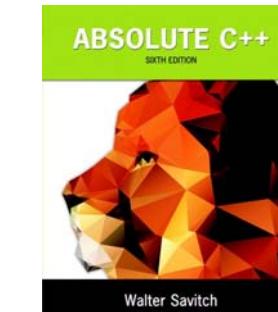
Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Push

- newElement added at the beginning of the linked list pointed to by stackTop
- Value of pointer stackTop updated

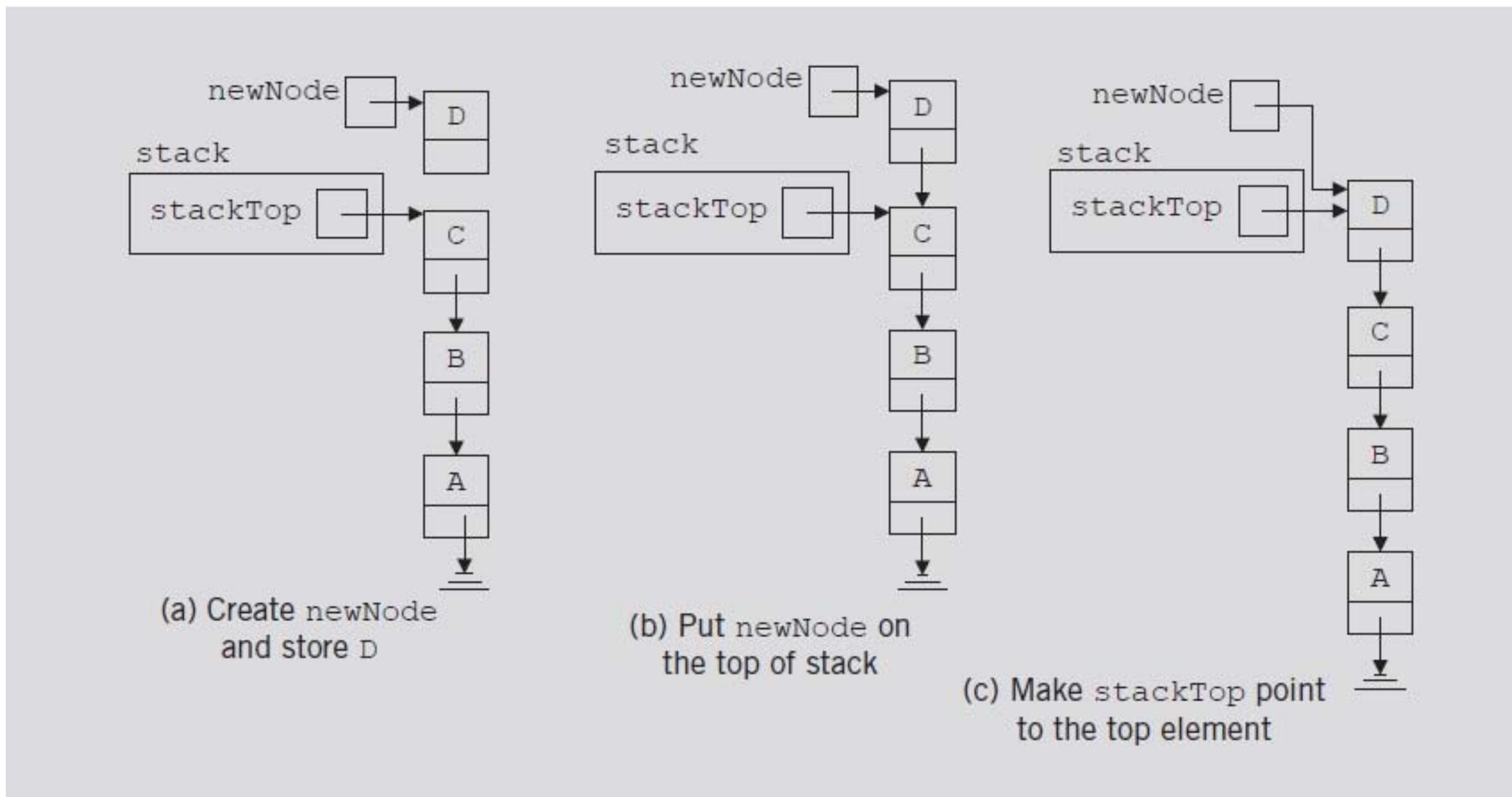


**FIGURE 7-11** Stack before the push operation

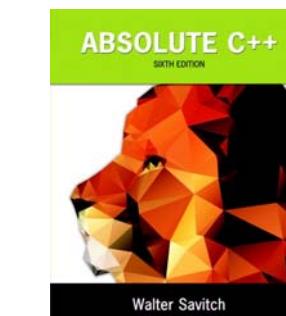


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Push (cont'd.)



**FIGURE 7-12** Push operation



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

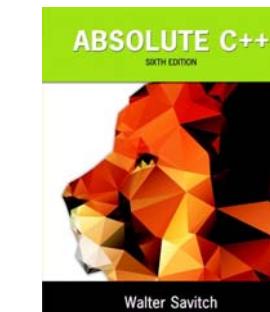
# Push (cont'd.)

- Definition of the push function

```
template <class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
    nodeType<Type> *newNode; //pointer to create the new node

    newNode = new nodeType<Type>; //create the node

    newNode->info = newElement; //store newElement in the node
    newNode->link = stackTop; //insert newNode before stackTop
    stackTop = newNode;        //set stackTop to point to the
                               //top node
} //end push
```

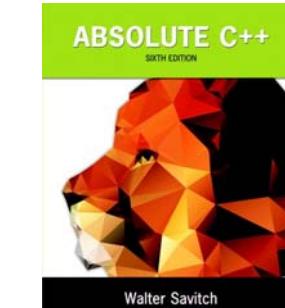


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Return the Top Element

- Returns information of the node to which stackTop pointing
- Definition of the top function

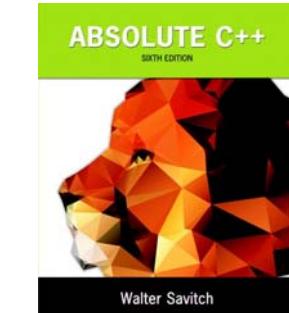
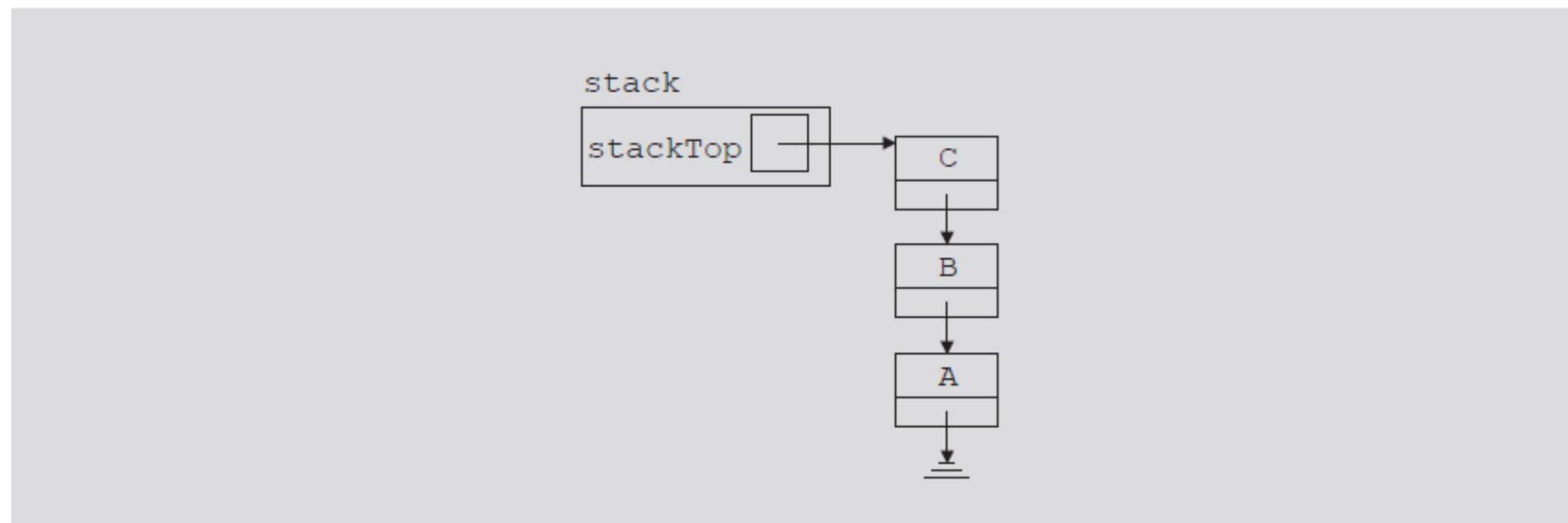
```
template <class Type>
Type linkedStackType<Type>::top() const
{
    assert(stackTop != NULL); //if stack is empty,
                           //terminate the program
    return stackTop->info;   //return the top element
}//end top
```



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

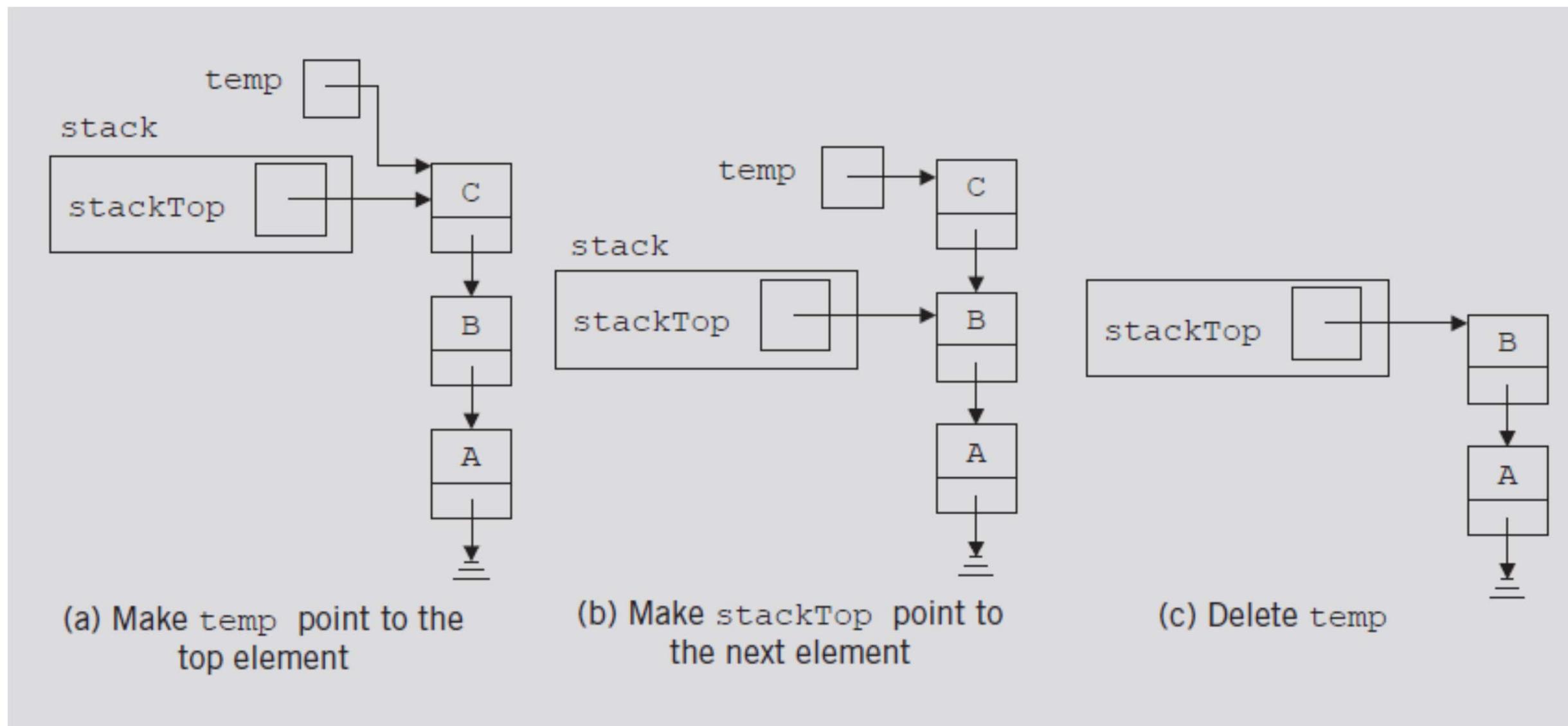
# Pop

- Removes top element of the stack
  - Node pointed to by `stackTop` removed
  - Value of pointer `stackTop` updated

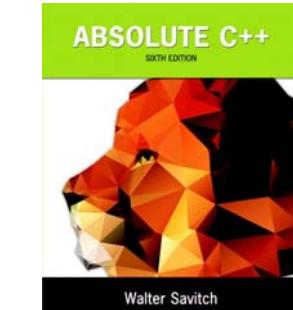


Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Pop (cont'd.)



**FIGURE 7-14** Pop operation



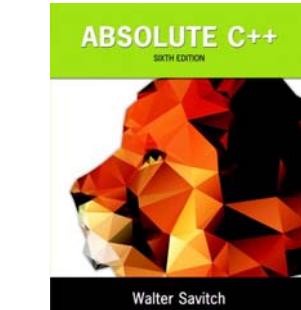
Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Pop (cont'd.)

- Definition of the pop function

```
template <class Type>
void linkedStackType<Type>::pop()
{
    nodeType<Type> *temp; //pointer to deallocate memory

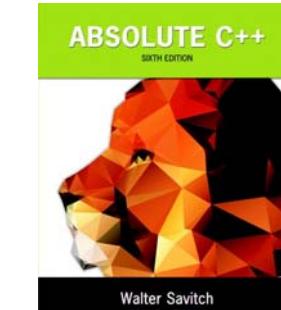
    if (stackTop != NULL)
    {
        temp = stackTop; //set temp to point to the top node
        stackTop = stackTop->link; //advance stackTop to the
                                    //next node
        delete temp; //delete the top node
    }
    else
        cout << "Cannot remove from an empty stack." << endl;
}//end pop
```



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Copy Stack

- Makes an identical copy of a stack
- Definition similar to the definition of `copyList` for linked lists
- Definition of the `copyStack` function



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

```

template <class Type>
void linkedStackType<Type>::copyStack
    (const linkedStackType<Type>& otherStack)
{
    nodeType<Type> *newNode, *current, *last;

    if (stackTop != NULL) //if stack is nonempty, make it empty
        initializeStack();

    if (otherStack.stackTop == NULL)
        stackTop = NULL;
    else
    {
        current = otherStack.stackTop; //set current to point
                                         //to the stack to be copied

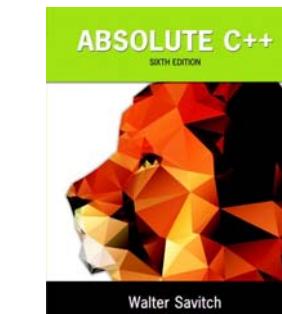
        //copy the stackTop element of the stack
        stackTop = new nodeType<Type>; //create the node

        stackTop->info = current->info; //copy the info
        stackTop->link = NULL; //set the link field to NULL
        last = stackTop; //set last to point to the node
        current = current->link; //set current to point to the
                               //next node

        //copy the remaining stack
        while (current != NULL)
        {
            newNode = new nodeType<Type>;

            newNode->info = current->info;
            newNode->link = NULL;
            last->link = newNode;
            last = newNode;
            current = current->link;
        } //end while
    } //end else
} //end copyStack

```



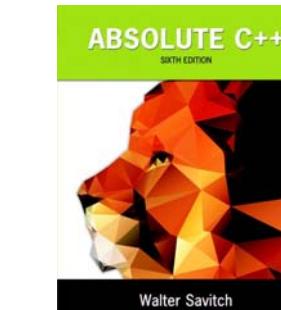
Modified Slides from:  
 Absolute C++ 6th Edition  
 Walter Savitch  
 Copyright © 2016 Pearson, Inc.  
 All rights reserved.

# Constructors and Destructors

- Definition of the functions to implement the copy constructor and the destructor

```
//copy constructor
template <class Type>
linkedStackType<Type>::linkedStackType(
    const linkedStackType<Type>& otherStack)
{
    stackTop = NULL;
    copyStack(otherStack);
}//end copy constructor

//destructor
template <class Type>
linkedStackType<Type>::~linkedStackType()
{
    initializeStack();
}//end destructor
```



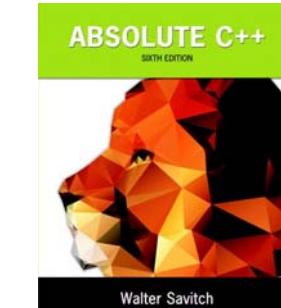
Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Overloading the Assignment Operator (=)

- Definition of the functions to overload the assignment operator

```
template <class Type>
const linkedStackType<Type>& linkedStackType<Type>::operator=
    (const linkedStackType<Type>& otherStack)
{
    if (this != &otherStack) //avoid self-copy
        copyStack(otherStack);

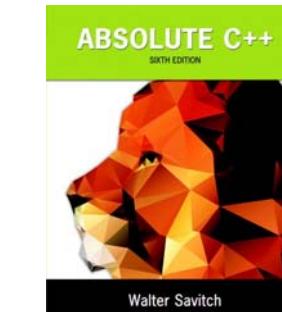
    return *this;
}//end operator=
```



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Stack Applications

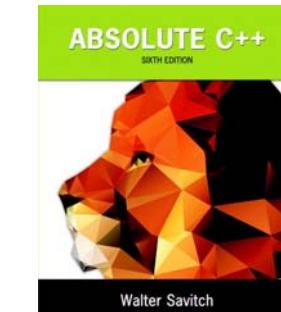
- Stacks are a very common data structure
  - compilers
    - parsing data between delimiters (brackets)
  - operating systems
    - program stack
  - virtual machines
    - manipulating numbers
      - pop 2 numbers off stack, do work (such as add)
      - push result back on stack and repeat
  - artificial intelligence
    - finding a path



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Application of Stacks: Postfix Expressions Calculator

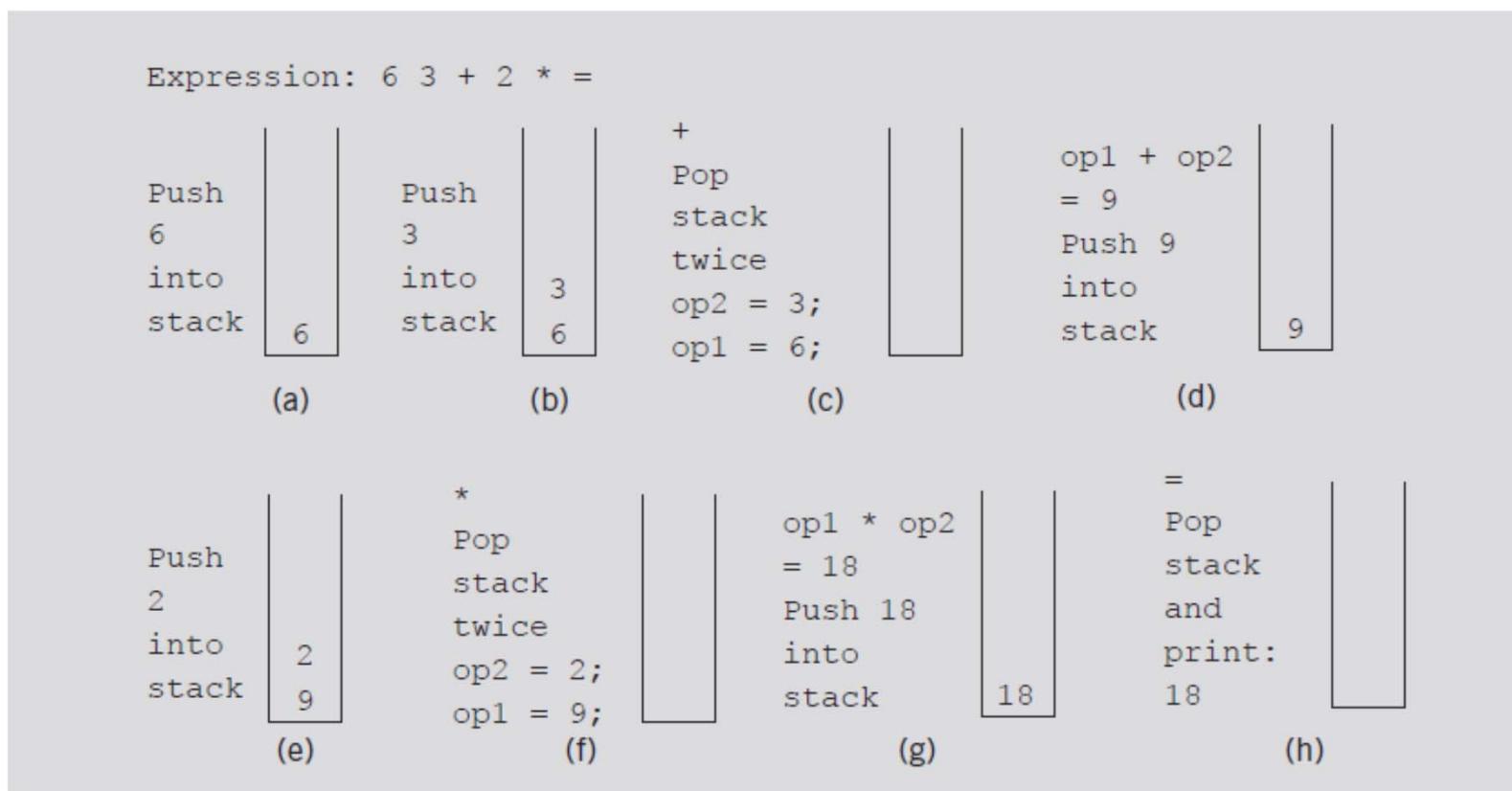
- Arithmetic notations
  - Infix notation: operator between operands
  - Prefix (Polish) notation: operator precedes operands
  - Reverse Polish notation: operator follows operands
- Stack use in compilers
  - Translate infix expressions into some form of postfix notation
  - Translate postfix expression into machine code



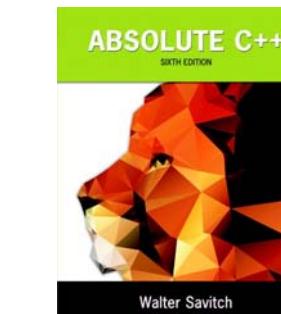
Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Application of Stacks: Postfix Expressions Calculator (cont'd.)

- Postfix expression:  $6\ 3\ +\ 2\ *\ =$



**FIGURE 7-15** Evaluating the postfix expression:  $6\ 3\ +\ 2\ *\ =$



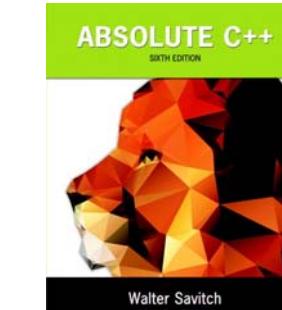
Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Application of Stacks: Postfix Expressions

## Calculator (cont'd.)

- Main algorithm pseudocode
  - Broken into four functions for simplicity
    - Function evaluateExpression
    - Function evaluateOpr
    - Function discardExp
    - Function printResult

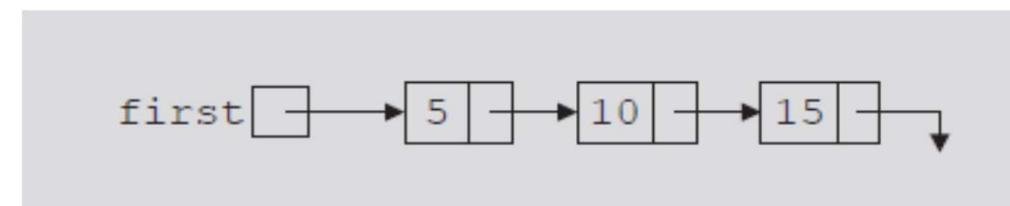
```
Read the first character
while not the end of input data
{
    a. initialize the stack
    b. process the expression
    c. output result
    d. get the next expression
}
```



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

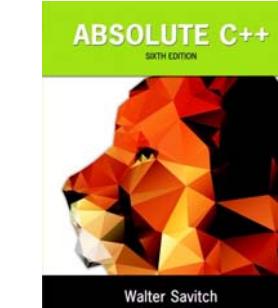
# Algorithm to Print a Linked List Backward

- Stack
  - Print a linked list backward
- Use linked implementation of stack

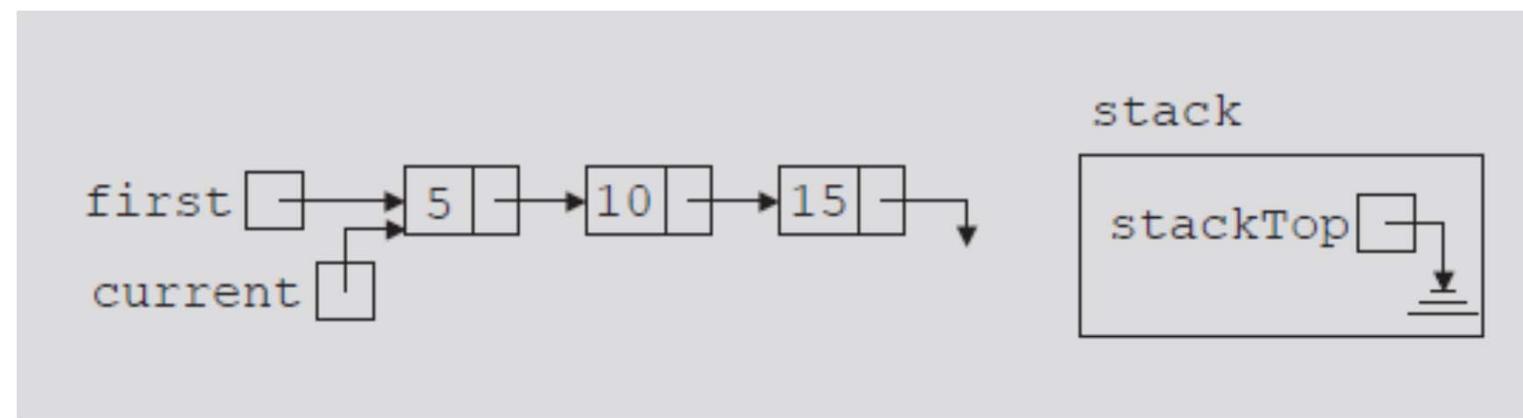


**FIGURE 7-16** Linked list

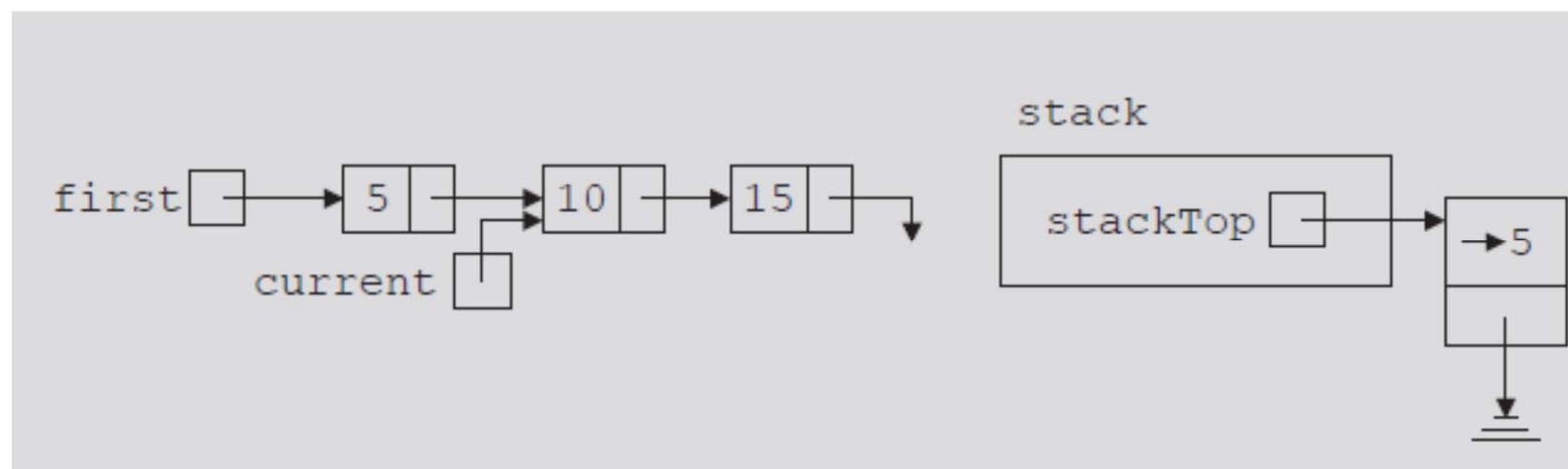
```
current = first;           //Line 1
while (current != NULL)    //Line 2
{
    stack.push(current);   //Line 3
    current = current->link; //Line 4
}                           //Line 5
                           //Line 6
```



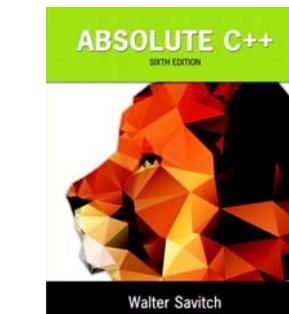
Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.



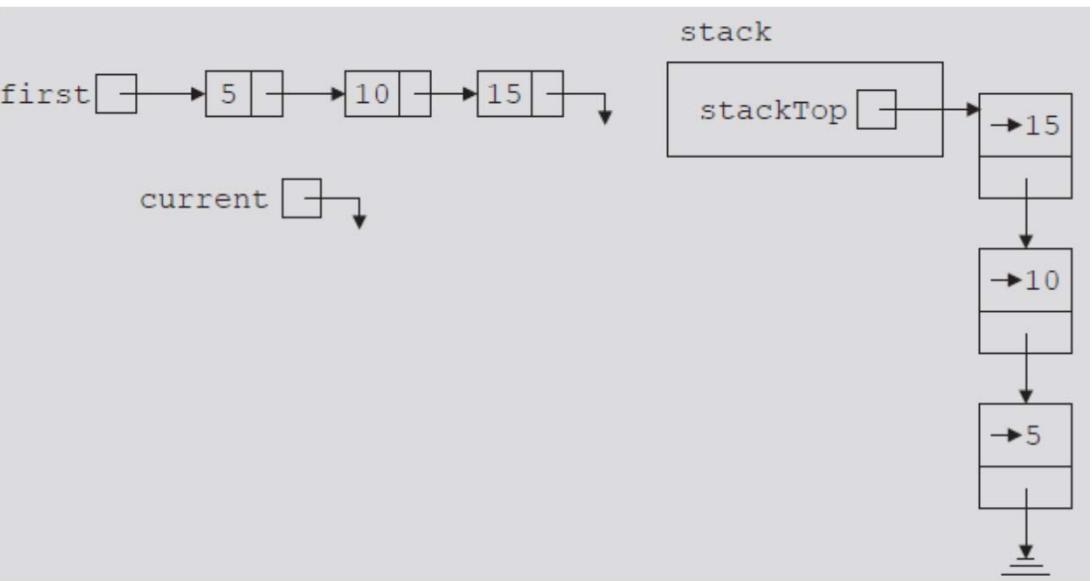
**FIGURE 7-17** List after the statement  
`current = first;` executes



**FIGURE 7-18** List and stack after the statements  
`stack.push(current);` and `current = current->link;` execute



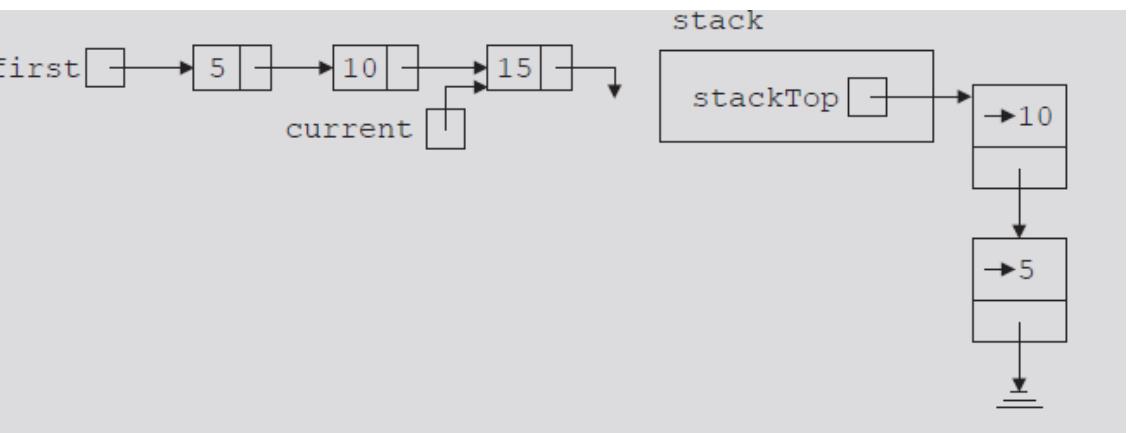
Modified Slides from:  
 Absolute C++ 6th Edition  
 Walter Savitch  
 Copyright © 2016 Pearson, Inc.  
 All rights reserved.



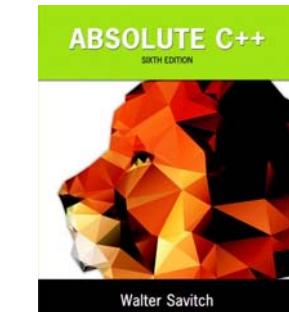
**FIGURE 7-19** List and stack after the while statement executes

```

while (!stack.isEmptyStack())           //Line 7
{
    current = stack.top();             //Line 8
    stack.pop();                      //Line 9
    cout << current->info << " ";   //Line 10
}
                                         //Line 11
                                         //Line 12
    
```



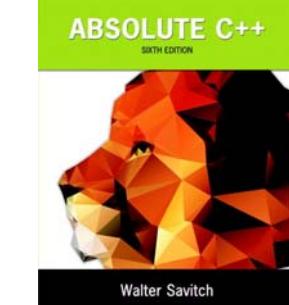
**FIGURE 7-20** List and stack after the statements `current = stack.top();` and `stack.pop();` execute



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Checking for Balanced Braces

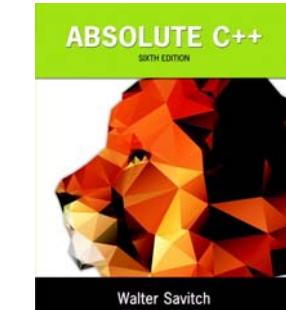
- Example of curly braces in C++ language
  - Balanced abc{defg{ijk}{l{mn}}op}qr
  - Not balanced abc{def}}{ghij{k}lm
- Requirements for balanced braces
  - For each }, must match an already encountered {
  - At end of string, must have matched each {



# Checking for Balanced Braces

- Initial draft of a solution.

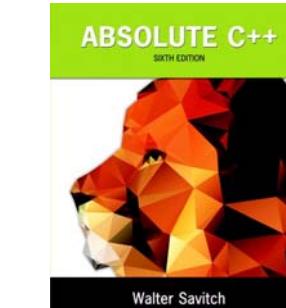
```
for (each character in the string)
{
    if (the character is a '{')
        aStack.push('{')
    else if (the character is a '}')
        aStack.pop()
}
```



# Checking for Balanced Braces

- Detailed pseudocode solution.

```
// Checks the string aString to verify that braces match.  
// Returns true if aString contains matching braces, false otherwise.  
checkBraces(aString: string): boolean  
{  
    aStack = a new empty stack  
    balancedSoFar = true  
    i = 0           // Tracks character position in string  
  
    while (balancedSoFar and i < length of aString)  
    {  
        ch = character at position i in aString  
        i++  
  
        // Push an open brace  
        if (ch is a '{')  
            aStack.push('{')  
  
        // Close brace  
        else if (ch is a '}')  
    }
```

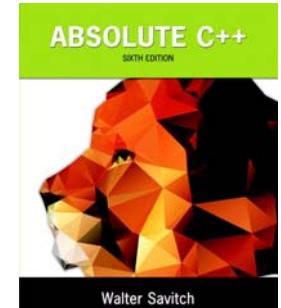


# Checking for Balanced Braces

- Detailed pseudocode solution.

```
// Close brace
else if (ch is a '}')
{
    if (!aStack.isEmpty())
        aStack.pop() // Pop a matching open brace
    else           // No matching open brace
        balancedSoFar = false
}
// Ignore all characters other than braces
}

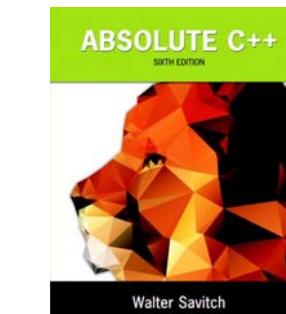
if (balancedSoFar and aStack.isEmpty())
    aString has balanced braces
else
    aString does not have balanced braces
}
```



# Checking for Balanced Braces

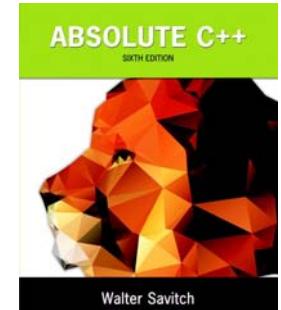
- Traces of algorithm that checks for balanced braces

| <u>Input string</u> | <u>Stack as algorithm executes</u> |    |    |    |                                                                                                                                                  |
|---------------------|------------------------------------|----|----|----|--------------------------------------------------------------------------------------------------------------------------------------------------|
|                     | 1.                                 | 2. | 3. | 4. |                                                                                                                                                  |
| {a{b}c}             |                                    |    |    |    | <ol style="list-style-type: none"><li>push {</li><li>push {</li><li>pop</li><li>pop</li></ol> <p>stack empty -&gt; balanced</p>                  |
| {a{bc}}             |                                    |    |    |    | <ol style="list-style-type: none"><li>push {</li><li>push {</li><li>pop</li></ol> <p>Stack not empty <math>\Rightarrow</math> not balanced</p>   |
| {ab}c}              |                                    |    |    |    | <ol style="list-style-type: none"><li>push {</li><li>pop</li></ol> <p>Stack empty when "}" encountered <math>\Rightarrow</math> not balanced</p> |



# Summary

- Stack
  - Last In First Out (LIFO) data structure
  - Implemented as array or linked list
  - Arrays: limited number of elements
  - Linked lists: allow dynamic element addition



Modified Slides from:  
Absolute C++ 6th Edition  
Walter Savitch  
Copyright © 2016 Pearson, Inc.  
All rights reserved.

# Preliminary Discussion

## Data Structures – a *Cheatsheet*

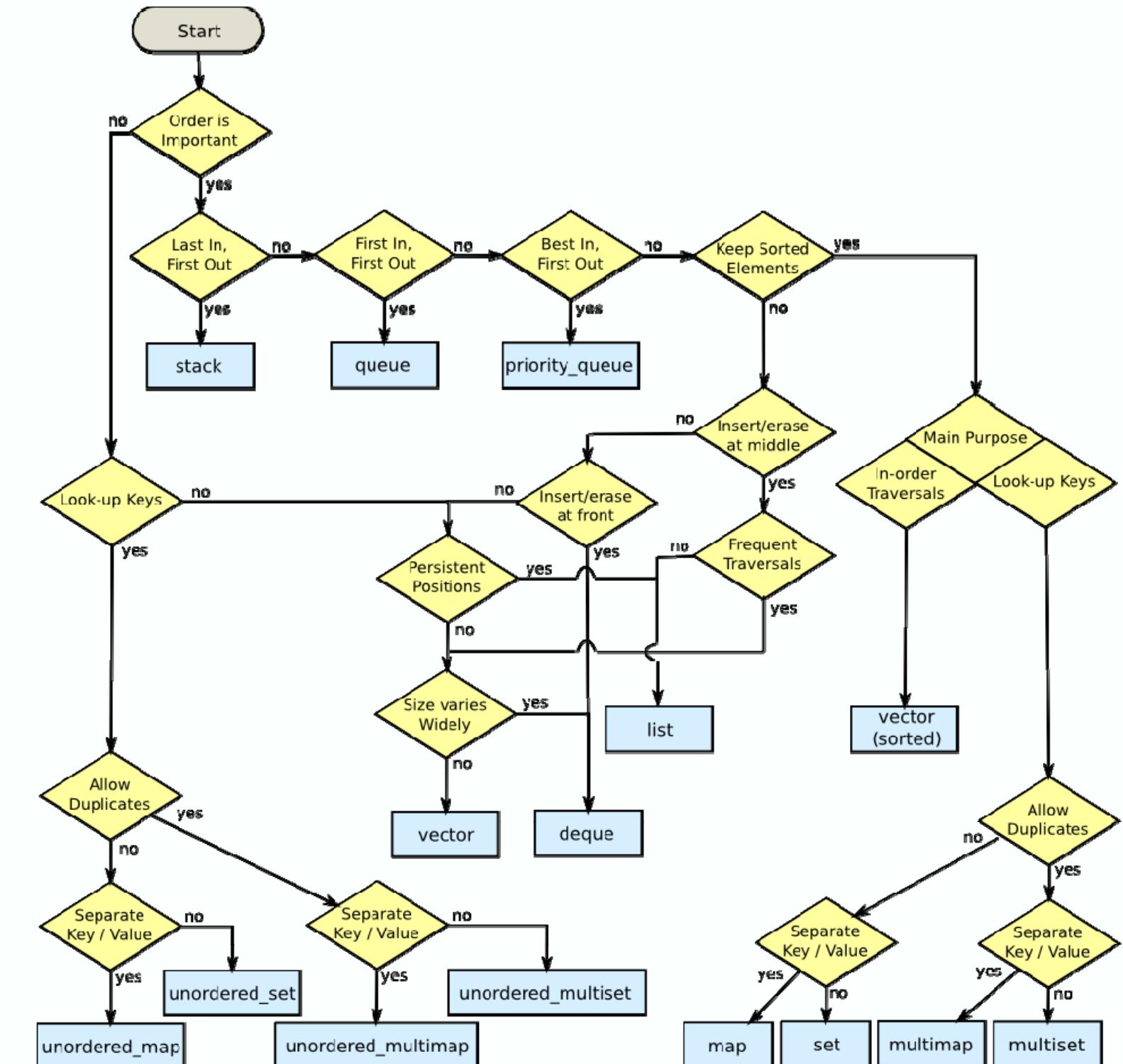
Literally just scratched the surface:

- Array
- Stack
- Queue

Not even Priority Queue yet ...

- List
  - Usually Doubly-Linked List
  - But a Forward-List also exists

Many more ...



CS-202

Time for Questions !