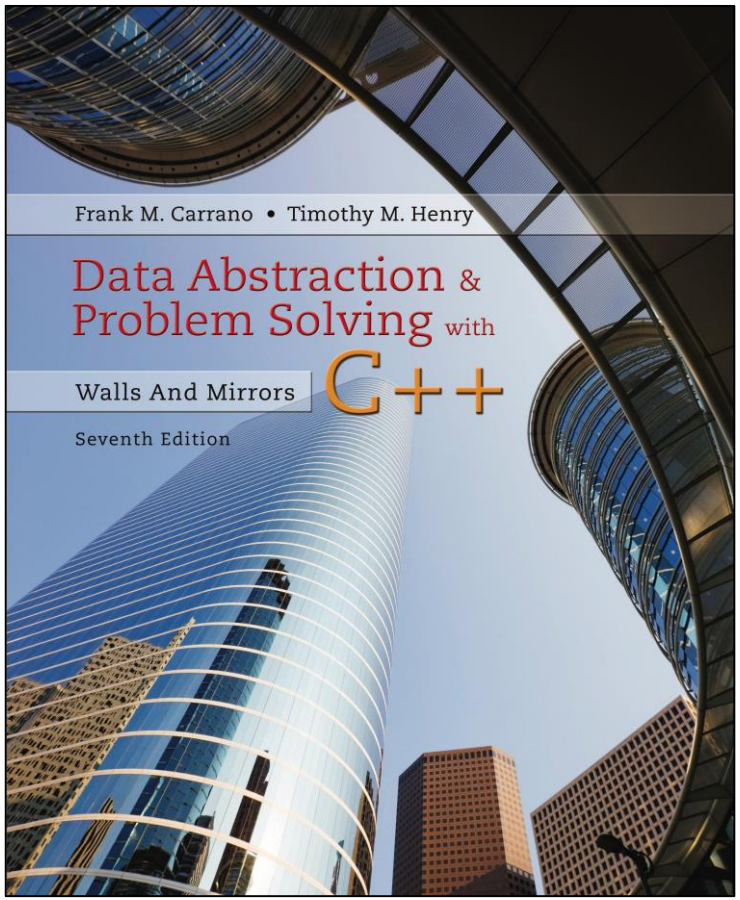
Data Abstraction & Problem Solving with C++: Walls and Mirrors

Seventh Edition



Chapter 6

Stacks

The Abstract Data Type Stack

- Operations on a stack
  - Last-in,
  - First-out behavior.
- Applications demonstrated
  - Evaluating algebraic expressions
  - Searching for a path between two points

Developing an ADT During the Design of a Solution (1 of 4)

- Consider typing a line of text on a keyboard
  - Use of backspace key to make corrections
  - You type `abcc←ddde←←←eg←fg`
  - Corrected input will be `abcdefg`
- Must decide how to store the input line.

Developing an ADT During the Design of a Solution (2 of 4)

```
// Read the line, correcting mistakes along the way
while (not end of line)
{
    Read a new character ch
    if (ch is not a '←')
        Add ch to the ADT
    else
        Remove from the ADT (and discard) the item that was added most recently
}
```

- Initial draft of solution.
- Two required operations
  - Add new item to ADT
  - Remove item added most recently

Developing an ADT During the Design of a Solution (3 of 4)

```
// Read the line, correcting mistakes along the way
while (not end of line)
{
    Read a new character ch
    if (ch is not a '←')
        Add ch to the ADT
    else if (the ADT is not empty)
        Remove from the ADT and discard the item that was added most recently
    else
        Ignore the '←'
}
```

- Read and correct algorithm.
- Third operation required
  - See whether ADT is empty

Developing an ADT During the Design of a Solution (4 of 4)

```
// Display the line in reverse order
while (the ADT is not empty)
{
    Get a copy of the item that was added to the ADT most recently and assign it to ch
    Display ch
    Remove from the ADT and discard the item that was added most recently
}
```

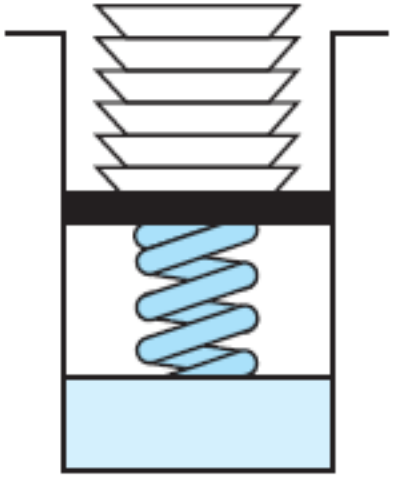
- Write-backward algorithm
- Fourth operation required
  - Get item that was added to ADT most recently.

Specifications for the ADT Stack (1 of 6)

- See whether stack is empty.
- Add new item to the stack.
- Remove from and discard stack item that was added most recently.
- Get copy of item that was added to stack most recently.

Specifications for the ADT Stack (2 of 6)

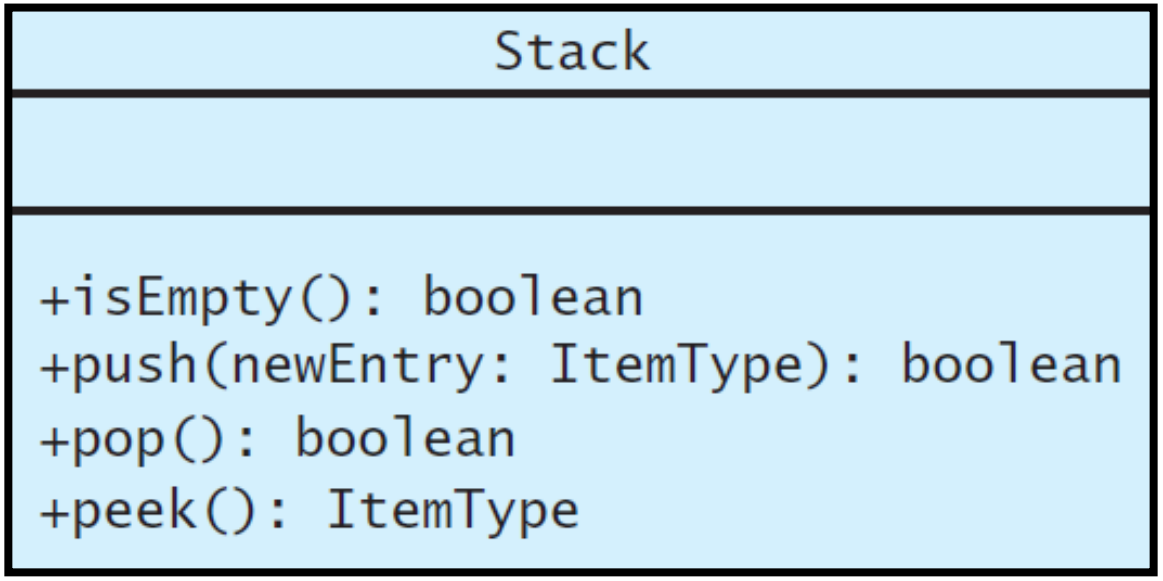
Figure 6-1 A stack of cafeteria plates



LIFO: The last item inserted onto a stack is the first item out

Specifications for the ADT Stack (3 of 6)

Figure 6-2 UML diagram for the class **Stack**



Specifications for the ADT Stack (4 of 6)

Listing 6-1 A C++ interface for stacks

```
1  /** @file StackInterface.h */
2  #ifndef STACK_INTERFACE_
3  #define STACK_INTERFACE_
4
5  template<class ItemType>
6  class StackInterface
7  {
8  public:
9     /** Sees whether this stack is empty.
10     @return  True if the stack is empty, or false if not. */
11    virtual bool isEmpty() const = 0;
12
13    /** Adds a new entry to the top of this stack.
14     @post  If the operation was successful, newEntry is at the top of the stack.
15     @param newEntry  The object to be added as a new entry.
16     @return  True if the addition is successful or false if not. */
17    virtual bool push(const ItemType& newEntry) = 0;
18
```

Specifications for the ADT Stack (5 of 6)

Listing 6-1 [Continued]

```
18
19      /** Removes the top of this stack.
20       @post  If the operation was successful, the top of the stack
21          has been removed.
22       @return  True if the removal is successful or false if not. */
23      virtual bool pop() = 0;
24
25      /** Returns a copy of the top of this stack.
26       @pre  The stack is not empty.
27       @post  A copy of the top of the stack has been returned, and
28          the stack is unchanged.
29       @return  A copy of the top of the stack. */
30      virtual ItemType peek() const = 0;
31
32      /** Destroys this stack and frees its assigned memory. */
33      virtual ~StackInterface() { }
34  }; // end StackInterface
35  #endif
```

Specifications for the ADT Stack (6 of 6)

- Axioms for ADT stack

```
(Stack()).isEmpty() = true
(Stack()).pop() = false
(Stack()).peek() = error
(aStack.push(item)).isEmpty() = false
(aStack.push(item)).peek() = item
(aStack.push(item)).pop() = true
(aStack.push(item)).pop() ⇒ aStack
```

Checking for Balanced Braces (1 of 5)

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

Checking for Balanced Braces (2 of 5)

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 (3 of 5)

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 '}')
      {
```

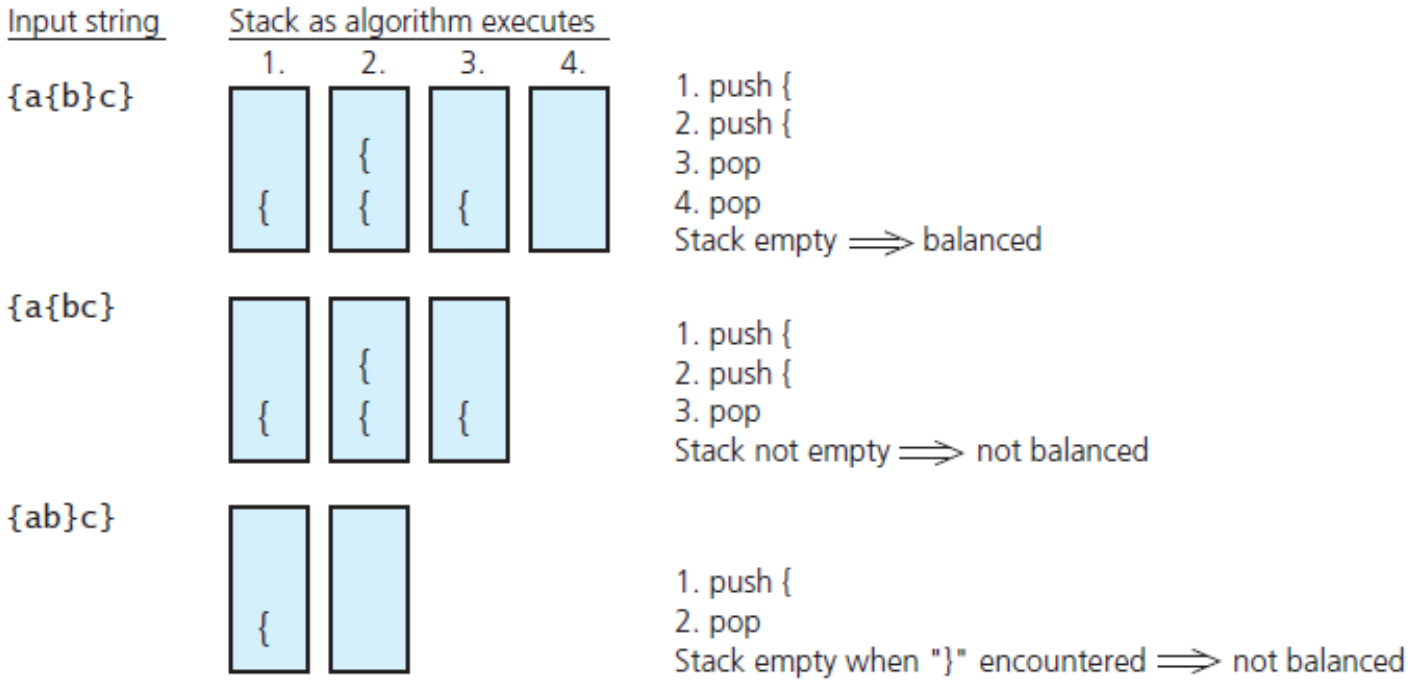
Checking for Balanced Braces (4 of 5)

Detailed pseudocode solution.

```
            aStack.push('{')
        // 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 (5 of 5)

Figure 6-3 Traces of algorithm that checks for balanced braces



Defining Languages

- A language a set of strings of symbols from a finite alphabet.
- Consider the C++ language

$$C{+}{+}Programs = \{\text{string } s : s \text{ is a syntactically correct C++ program}\}$$

- The set of algebraic expressions forms a language

$$AlgebraicExpressions = \{\text{string } s : s \text{ is an algebraic expression}\}$$

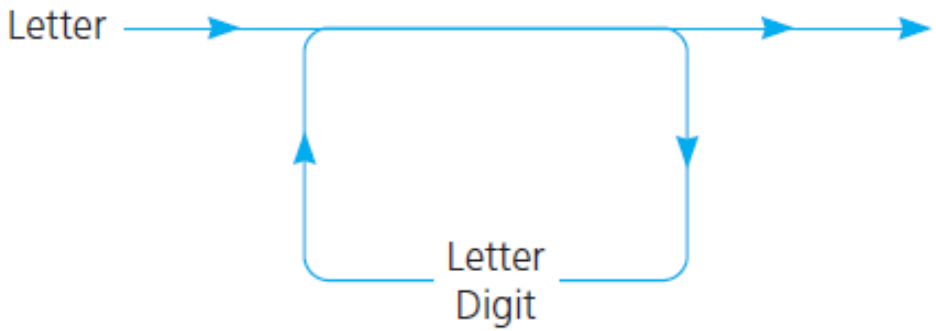
- A grammar states the rules of a language.

The Basics of Grammars (1 of 5)

- A grammar uses several special symbols
  - x | y means x or y.
  - x y (and sometimes $x \cdot y$) means x followed by y.
  - < word > means any instance of word, where word is a symbol that must be defined elsewhere in the grammar.

The Basics of Grammars (2 of 5)

Figure 5-1 A syntax diagram for C++ identifiers



- Grammar

  <identifier> = *<letter>* | *<identifier>* *<letter>* | *<identifier>* *<digit>*

  <letter> = a | b | … | z | A | B | …| Z | _ | $

  <digit> = 0 | 1 | … | 9

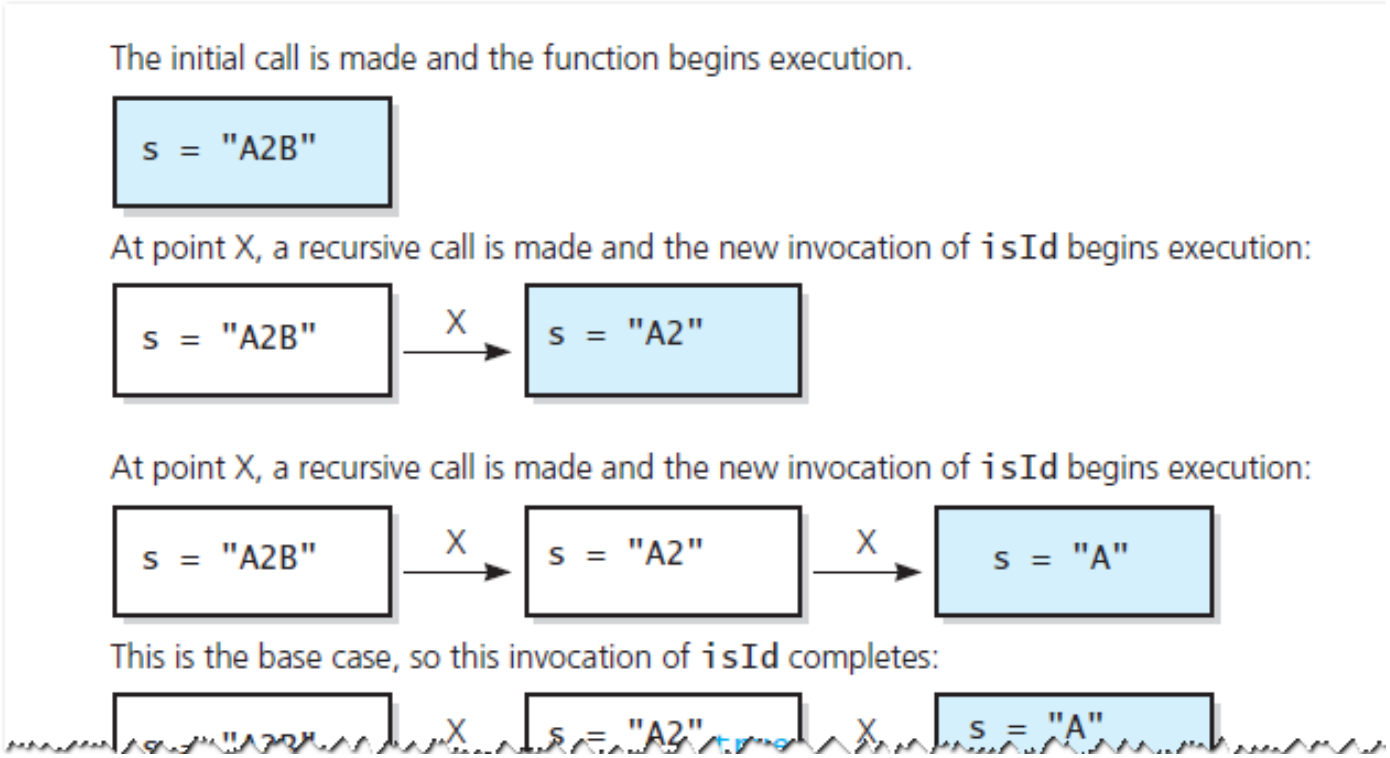
The Basics of Grammars (3 of 5)

Pseudocode for a recursive valued function that determines whether a string is in the language C++Identifiers

```
// Returns true if s is a legal C++ identifier; otherwise returns false.
isId(s: string): boolean
{
   if (s is of length 1)                          // Base case
      if (s is a letter)
         return true
      else
         return false
   else if (the last character of s is a letter or a digit)
      return isId(s minus its last character)  // Point X
   else
      return false
}
```

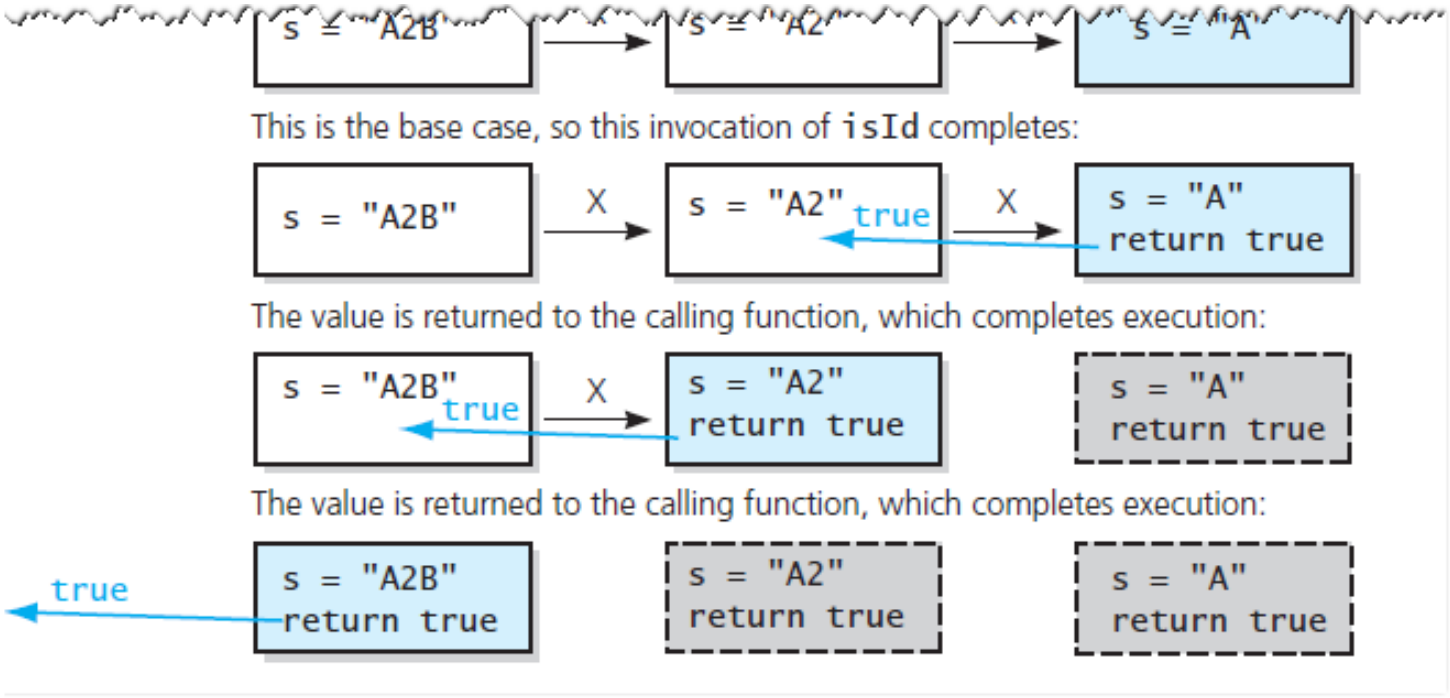
The Basics of Grammars (4 of 5)

Figure 5-2 Trace of **isId("A2B")**



The Basics of Grammars (5 of 5)

Figure 5-2 [continued]



Two Simple Languages (1 of 2)

- Palindromes

$$Palindromes = \{\text{string } s : s \text{ reads the same left to right as right to left}\}$$

- Recursive definition of palindrome
  - The first and last characters of *s* are the same
  - *s* minus its first and last characters is a palindrome
- Grammar for the language of palindromes

$$\langle pal \rangle = \text{empty string} \mid \langle ch \rangle \mid \text{a} \langle pal \rangle \text{ a} \mid \text{b} \langle pal \rangle \text{ b} \mid \ldots \mid \text{Z} \langle pal \rangle \text{ Z}$$
$$\langle ch \rangle = \text{a} \mid \text{b} \mid \ldots \mid \text{z} \mid \text{A} \mid \text{B} \mid \ldots \mid \text{Z}$$

Two Simple Languages (2 of 2)

- Strings of the form A^nB^n

$$AnBn = \{\text{string } s : s \text{ is of the form } A^nB^n \text{ for some } n \geq 0\}$$

- Grammar for the language A^nB^n is

$$\langle legal\_word \rangle = \text{empty string} \mid A \langle legal\_word \rangle B$$

Algebraic Expressions

- Compiler must recognize and evaluate algebraic expressions

```
y = x + z*(w/k + z*(7*6));
```

  - Determine if legal expression
  - If legal, evaluate expression

Kinds of Algebraic Expressions (1 of 2)

- Infix expressions
  - Every binary operator appears between its operands
- This convention necessitates …
  - Associativity rules
  - Precedence rules
  - Use of parentheses

$$a + b * c$$

$$(a + b) * c$$

Kinds of Algebraic Expressions (2 of 2)

- Prefix expressions
  - Operator appears before its operands

    a + *b* equivalent to +*ab*
- Postfix expressions
  - Operator appears after its operands

    a + *b* equivalent to *ab*+
- Example:
  - Infix: a + b * c + (d * e + f) * g
  - Prefix: + + a * b c * + * d e f g
  - Postfix: a b c * + d e * f + g * +

// move operator right to find infix
// move operator left to find infix

Prefix Expressions (1 of 11)

- Grammar that defines language of all prefix expressions

$$\begin{aligned} &\langle prefix \rangle = \langle identifier \rangle \mid \langle operator \rangle \langle prefix \rangle \langle prefix \rangle \\ &\langle operator \rangle = + \mid - \mid * \mid / \\ &\langle identifier \rangle = a \mid b \mid \ldots \mid z \end{aligned}$$

- Recursive algorithm that recognizes whether string is a prefix expression
  - Check if first character is an operator
  - Remainder of string consists of two consecutive prefix expressions (recursion --> smaller problem)

Prefix Expressions (2 of 11)

endPre determines the end of a prefix expression

```
// Finds the end of a prefix expression, if one exists.
// Precondition: The substring of strExp from the index first through the end of
// the string contains no blank characters.
// Postcondition: Returns the index of the last character in the prefix expression that
// begins at index first of strExp, or –1 if no such prefix expression exists.
endPre(strExp: string, first: integer): integer
{
   last = strExp.length() - 1
   if (first < 0 or first > last)
      return -1

   ch = character at position first of strExp
   if (ch is an identifier)
      return first                    // Index of last character in simple prefix expression
   else if (ch is an operator)
   {
```

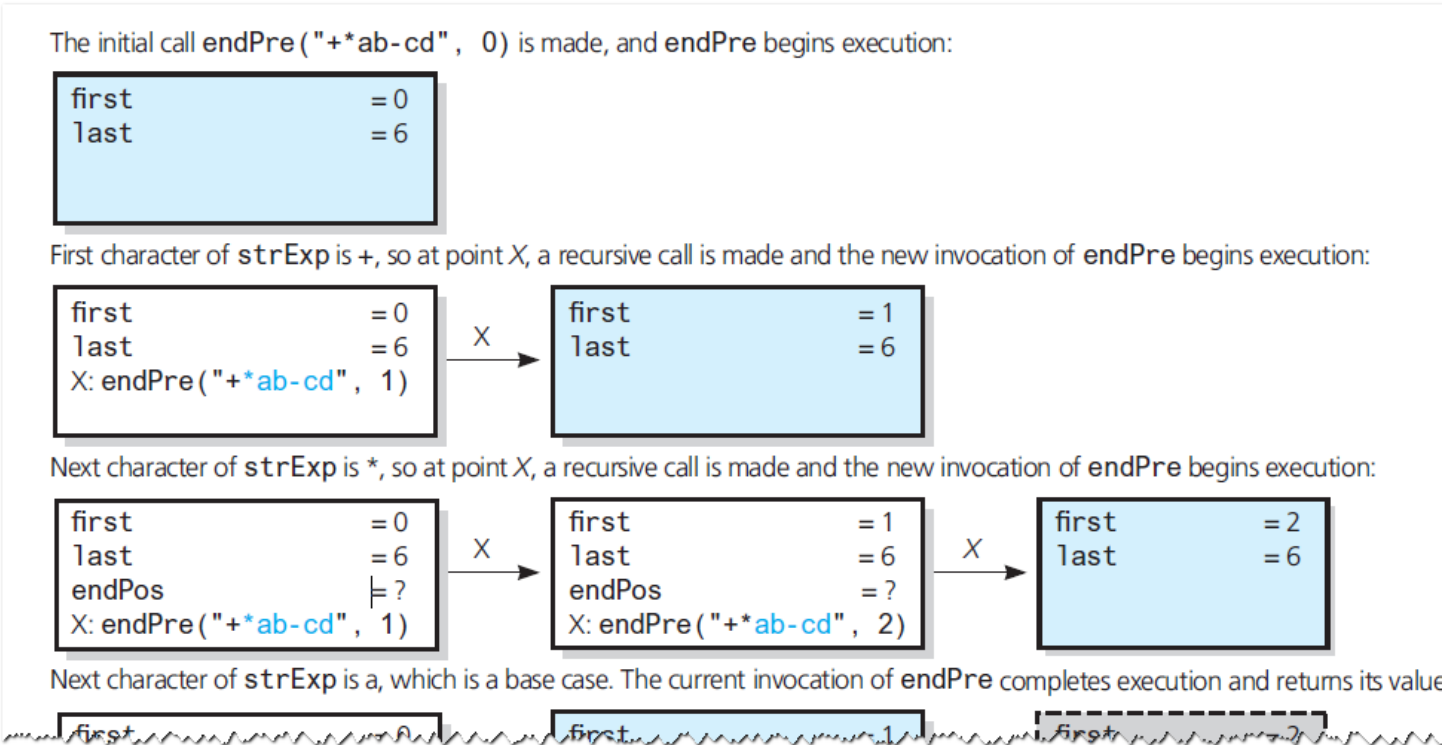
Prefix Expressions (3 of 11)

endPre determines the end of a prefix expression

```
        return first          // Index of last character in simple prefix expression
    else if (ch is an operator)
    {
        // Find the end of the first prefix expression
        endPos = endPre(strExp, first + 1)     // Point X
        // If the end of the first prefix expression was found, find the end of the second
        // prefix expression
        if (endPos > -1)
            return endPre(strExp, endPos + 1) // Point Y
        else
            return -1
    }
    else
        return -1
}
```

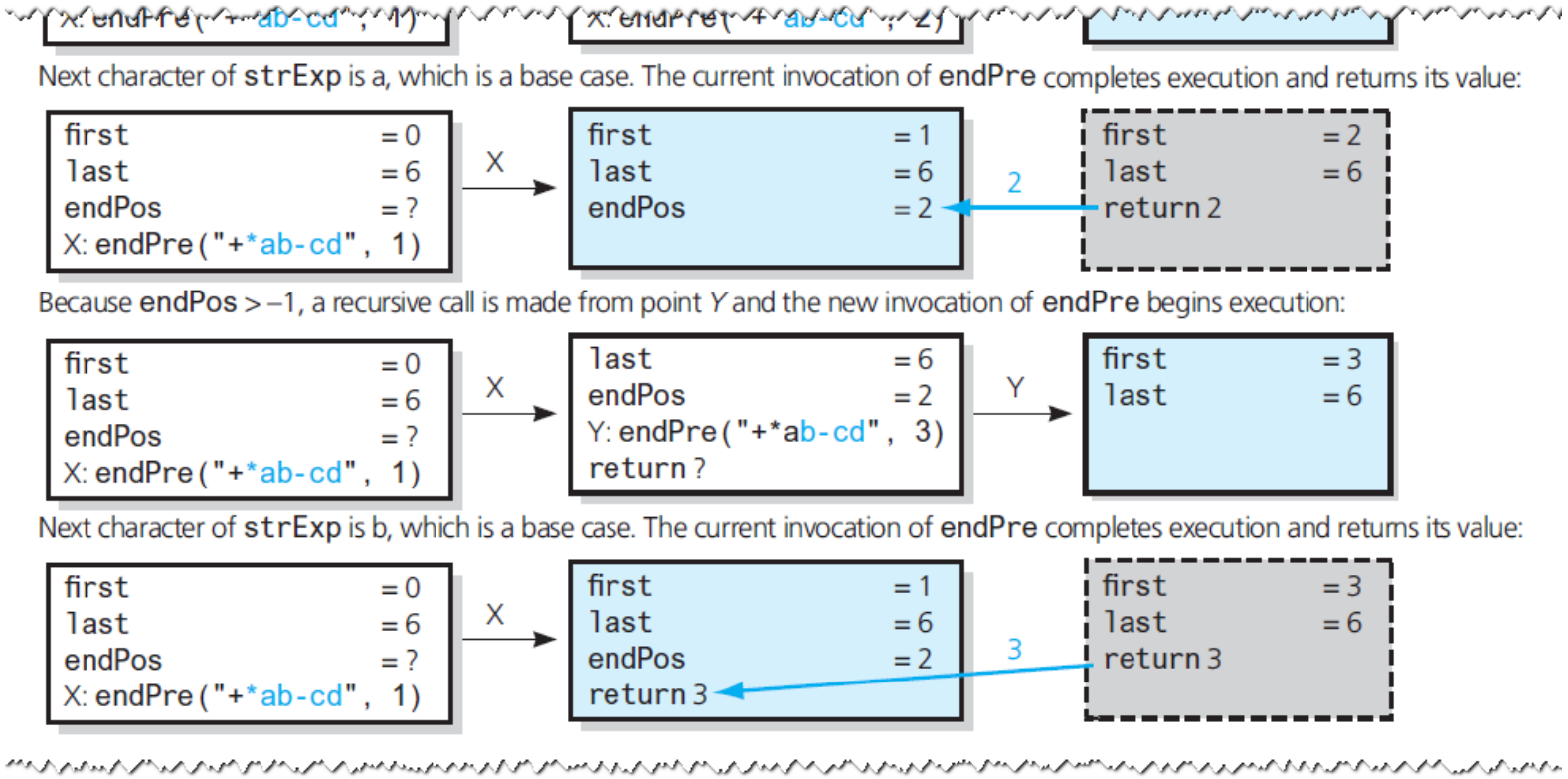
Prefix Expressions (4 of 11)

Figure 5-3 Trace of **endPre ("+*ab–cd", 0)**



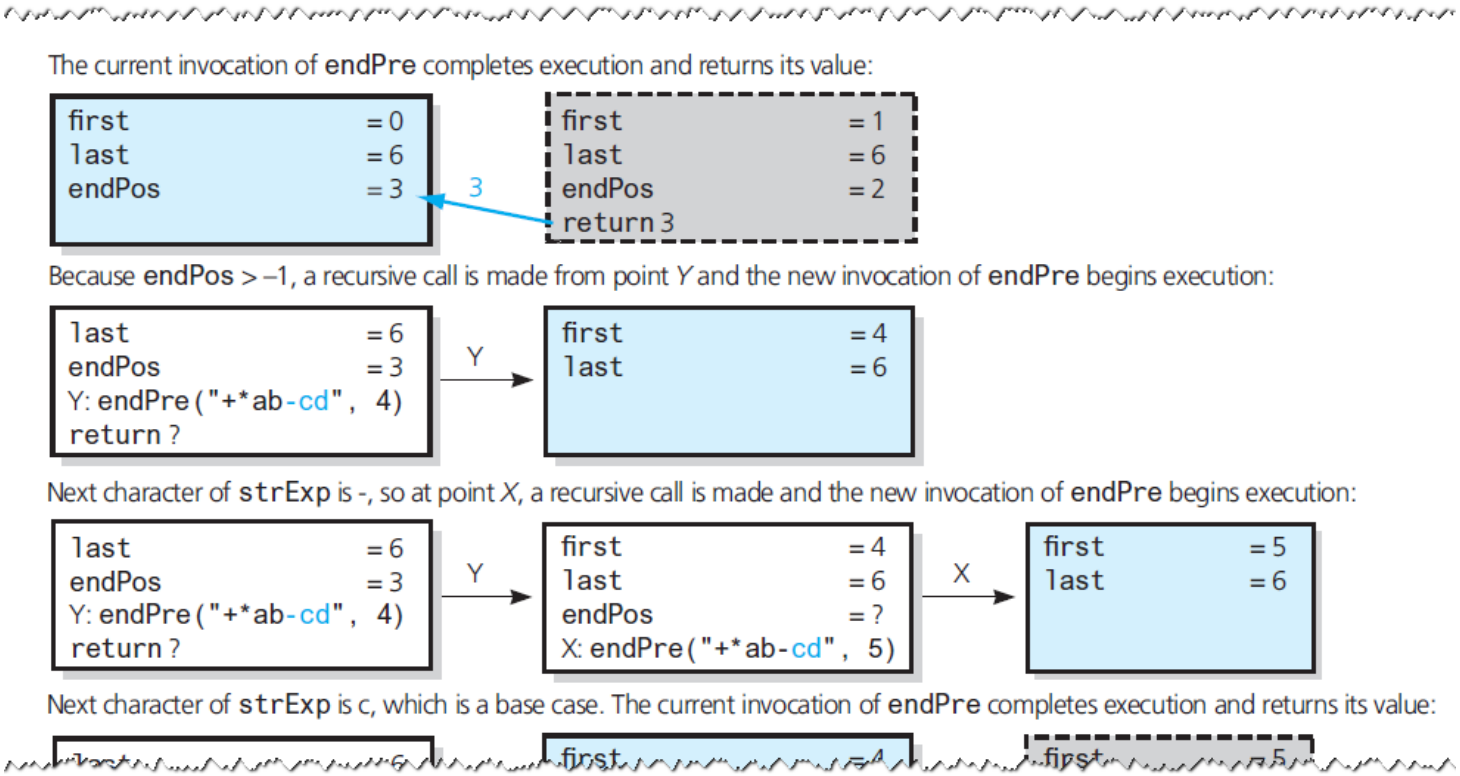
Prefix Expressions (5 of 11)

Figure 5-3 [Continued]



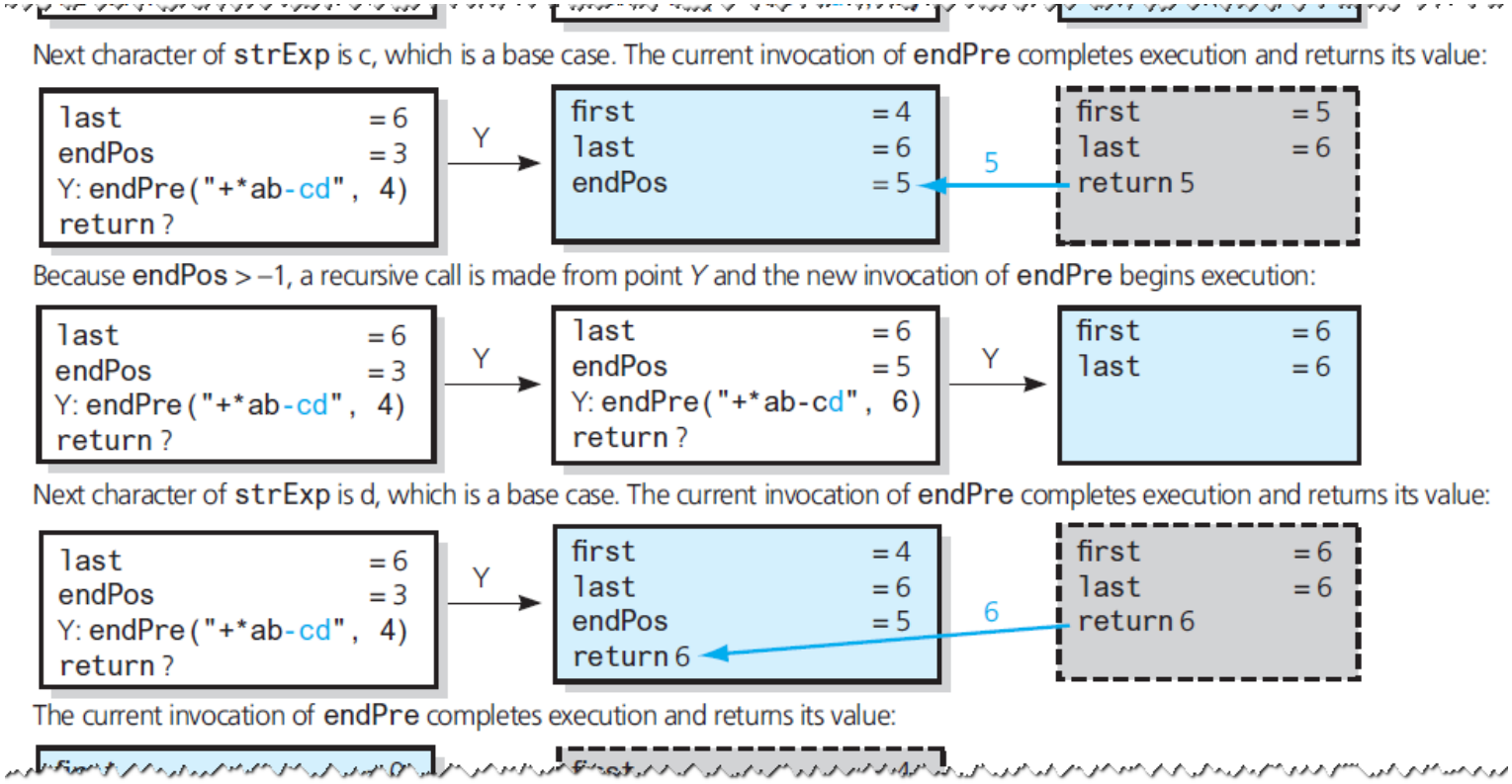
Prefix Expressions (6 of 11)

Figure 5-3 [Continued]



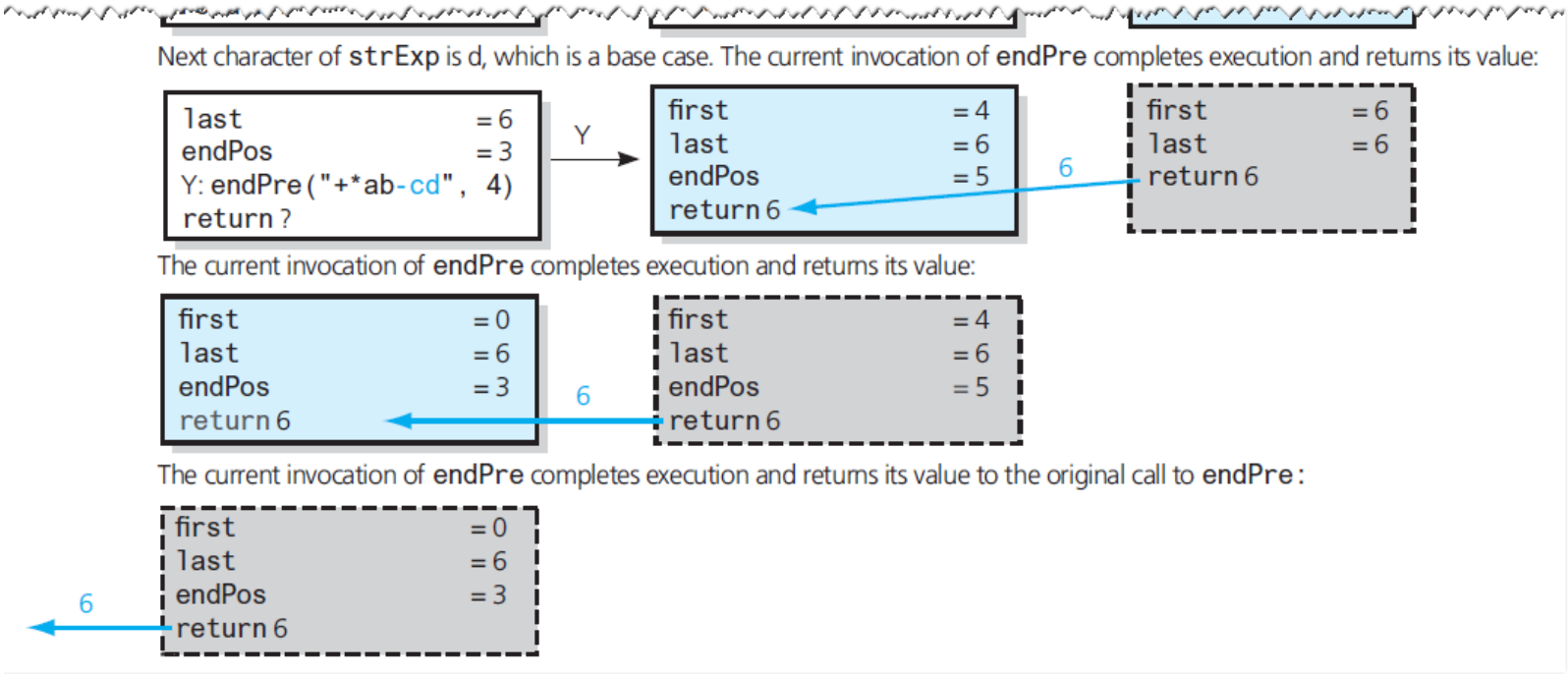
Prefix Expressions (7 of 11)

Figure 5-3 [Continued]



Prefix Expressions (8 of 11)

Figure 5-3 [Continued]



Prefix Expressions (9 of 11)

A recognition algorithm for prefix expressions

```
// Sees whether an expression is a prefix expression.
// Precondition: strExp contains a string with no blank characters.
// Postcondition: Returns true if the expression is in prefix form; otherwise returns false.
isPrefix(strExp: string): boolean
{
   lastChar = endPre(strExp, 0)
   return (lastChar >= 0) and (lastChar == strExp.length() - 1)
}
```

Prefix Expressions (10 of 11)

An algorithm to evaluate a prefix expression

```
// Returns the value of a given prefix expression.
// Precondition: strExp is a string containing a valid prefix expression with no blanks.
evaluatePrefix(strExp: string): float
{
   strLength = the length of strExp
   if (strLength == 1)
      return value of the identifier // Base case—single identifier
   else
   {
      op = strExp[0]                 // strExp begins with an operator
      // Find the end of the first prefix expression—will be the first operand
      endFirst = endPre(strExp, 1)
      // Recursively evaluate this first prefix expression
```

Prefix Expressions (11 of 11)

An algorithm to evaluate a prefix expression

```
        endFirst = endPre(strExp, 1)

        // Recursively evaluate this first prefix expression
        operand1 = evaluatePrefix(strExp[1..endFirst]);

        // Recursively evaluate the second prefix expression—will be the second operand
        endSecond = strLength - endFirst + 1
        operand2 = evaluatePrefix(strExp[endFirst + 1..endSecond])

        // Evaluate the prefix expression
        return operand1 op operand2
    }
}
```

Postfix Expressions (1 of 2)

Grammar that defines the language of all postfix expressions

$$\langle postfix \rangle = \langle identifier \rangle \mid \langle postfix \rangle \langle postfix \rangle \langle operator \rangle$$
$$\langle operator \rangle = + \mid - \mid * \mid /$$
$$\langle identifier \rangle = a \mid b \mid \ldots \mid z$$

An algorithm that converts a prefix expression to postfix form

```
if (exp is a single letter)
   return exp
else
   return postfix(prefix1) • postfix(prefix2) • <operator>
```

Postfix Expressions (2 of 2)

Recursive algorithm that converts a prefix expression to postfix form

```
// Converts a prefix expression to postfix form.
// Precondition: The string preExp is a valid prefix expression with no blanks.
// Postcondition: Returns the equivalent postfix expression.
convertPreToPost(preExp: string): string
{
   preLength = the length of preExp
   ch = first character in preExp
   postExp = an empty string

   if (ch is a lowercase letter)
      // Base case—single identifier
      postExp = postExp • ch          // Append to end of postExp
   else // ch is an operator
   {
      // pre has the form <operator> <prefix1> <prefix2>
      endFirst = endPre(preExp, 1)    // Find the end of prefix1

      // Recursively convert prefix1 into postfix form
      postExp = postExp • convert(preExp[1..endFirst])

      // Recursively convert prefix2 into postfix form
      postExp = postExp • convert(preExp[endFirst + 1..preLength - 1))

      postExp = postExp • ch          // Append the operator to the end of postExp
   }
   return postExp
}
```

Fully Parenthesized Expressions

- Grammar for language of fully parenthesized algebraic expressions

$$\begin{aligned} &\langle infix \rangle = \langle identifier \rangle \mid (\langle infix \rangle \langle operator \rangle \langle infix \rangle) \\ &\langle operator \rangle = + \mid - \mid * \mid / \\ &\langle identifier \rangle = a \mid b \mid \ldots \mid z \end{aligned}$$

- Most programming languages support definition of algebraic expressions
  - Includes both precedence rules for operators and rules of association

Using Stacks with Algebraic Expressions

- Strategy
  - Develop algorithm to evaluate postfix
  - Develop algorithm to transform infix to postfix
- These give us capability to evaluate infix expressions
  - This strategy easier than **directly** evaluating infix expression

Evaluating Postfix Expressions (1 of 3)

- Infix expression $2 * (3 + 4)$
- Equivalent postfix 2 3 4 + *
  - Operator in postfix applies to two operands immediately preceding
- Assumptions for our algorithm
  - Given string is correct postfix
  - No unary, no exponentiation operators
  - Operands are single lowercase letters, integers

Evaluating Postfix Expressions (2 of 3)

Figure 6-4 The effect of a postfix calculator on a stack when evaluating the expression $2 * (3 + 4)$

Key entered	Calculator action		Stack (bottom to top):
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = peek	(4)	2 3 4
	pop		2 3
	operand1 = peek	(3)	2 3
	pop		2
	result = operand1 + operand2	(7)	
	push result		2 7
*	operand2 = peek	(7)	2 7
	pop		2
	operand1 = peek	(2)	2
	pop		
	result = operand1 * operand2	(14)	
	push result		14

Evaluating Postfix Expressions (3 of 3)

A pseudocode algorithm that evaluates postfix expressions

```
for (each character ch in the string)
{
    if (ch is an operand)
        Push the value of the operand ch onto the stack
    else // ch is an operator named op
    {
        // Evaluate and push the result
        operand2 = top of stack
        Pop the stack

        operand1 = top of stack
        Pop the stack

        result = operand1 op operand2
        Push result onto the stack
    }
}
```

integer -> push
operator -> pop last 2
and do op and push result

Infix to Postfix (1 of 6)

- Important facts
  - Operands always stay in same order with respect to one another.
  - Operator will move only “to the right” with respect to the operands;
    - If in the infix expression the operand *x* precedes the operator *op*,
    - Also true that in the postfix expression the operand *x* precedes the operator *op*.
  - All parentheses are removed.

Infix to Postfix (2 of 6)

First draft of algorithm to convert infix to postfix

```
Initialize postfixExp to the empty string
for (each character ch in the infix expression)
{
    switch (ch)
    {
        case ch is an operand:
            Append ch to the end of postfixExp
            break
        case ch is an operator:
            Save ch until you know where to place it
            break
        case ch is a '(' or a ')':
            Discard ch
            break
    }
}
```

Infix to Postfix (3 of 6)

- Determining where to place operators in postfix expression
  - Parentheses
  - Operator precedence
  - Left-to-right association
- Note difficulty
  - Infix expression not always fully parenthesized
  - Precedence and left-to-right association also affect results

Infix to Postfix (4 of 6)

Figure 6-5 A trace of the algorithm that converts the infix expression **a - (b + c * d) / e** to postfix form

ch	operatorStack (top to bottom)	postfixExp	
a		a	
–	–	a	append operands, push operators and (until
(	(–	a	ch is). If ch is) pop operators and append from stack
b	(–	a b	
+	+(–	a b	
c	+(–	a b c	
*	*+(–	a b c	
d	*+(–	a b c d	
)	+(–	a b c d *	Move operators from stack to
	(–	a b c d * +	postfixExp until "("
	–	a b c d * +	
/	/–	a b c d * +	
e	/–	a b c d * + e	
	–	a b c d * + e /	Copy operators from
		a b c d * + e / –	stack to postfixExp

Infix to Postfix (5 of 6)

Pseudocode algorithm that converts infix to postfix

```
for (each character ch in the infix expression)
{
   switch (ch)
   {
      case operand:        // Append operand to end of postfix expression—step 1
         postfixExp = postfixExp • ch
         break
      case '(':            // Save '(' on stack—step 2
         operatorStack.push(ch)
         break
      case operator:       // Process stack operators of greater precedence—step 3
         while (!operatorStack.isEmpty() and operatorStack.peek() is not a '(' and
                precedence(ch) <= precedence(operatorStack.peek()))
         {
            Append operatorStack.peek() to the end of postfixExp
            operatorStack.pop()
         }
         operatorStack.push(ch) // Save the operator
         break
      case ')':                 // Pop stack until matching '('—step 4
```

Infix to Postfix (6 of 6)

Pseudocode algorithm that converts infix to postfix

```
            break
         case ')':                    // Pop stack until matching '('—step 4
            while (operatorStack.peek() is not a '(')
            {
               Append operatorStack.peek() to the end of postfixExp
               operatorStack.pop()
            }
            operatorStack.pop()       // Remove the open parenthesis
            break
      }
   }
   // Append to postfixExp the operators remaining in the stack—step 5
   while (!operatorStack.isEmpty())
   {
      Append operatorStack.peek() to the end of postfixExp
      operatorStack.pop()
   }
```

Backtracking

continue here

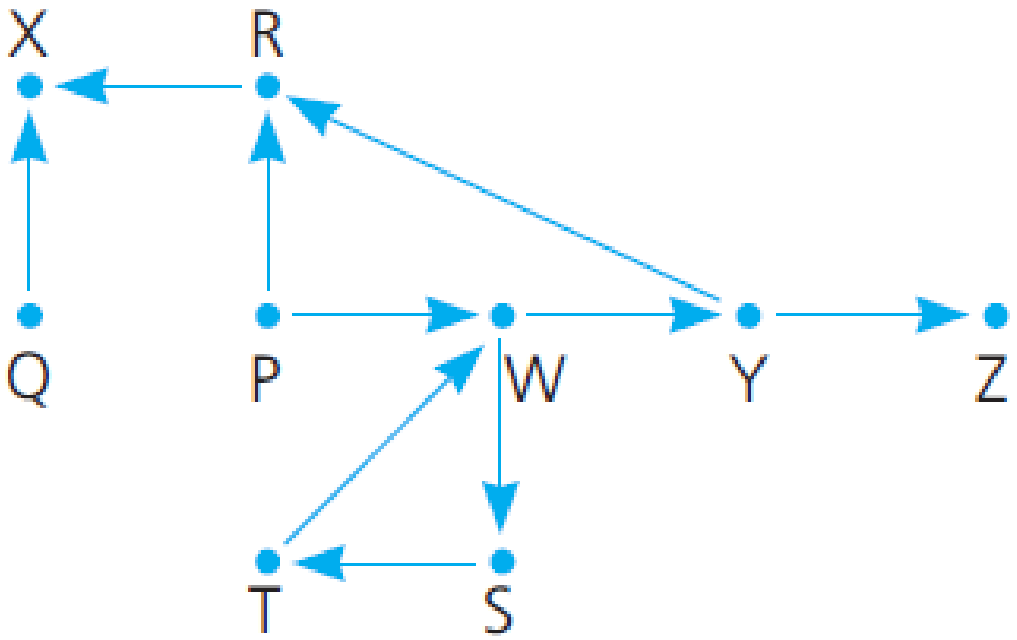
- Strategy for guessing at a solution and …
  - Backing up when an impasse is reached
  - Retracing steps in reverse order
  - Trying a new sequence of steps
- Combine recursion and backtracking to solve problems

Searching for an Airline Route (1 of 11)

- Must find a path from some point of origin to some destination point
- Program to process customer requests to fly
  - From some origin city
  - To some destination city
- Use three input text files
  - names of cities served
  - Pairs of city names, flight origins and destinations
  - Pairs of names, request origins, destinations

Searching for an Airline Route (2 of 11)

Figure 5-4 Flight map for HPAir



- The flight map for HPAir is a graph
  - Adjacent vertices are two vertices that are joined by an edge
  - A directed path is a sequence of directed edges

Searching for an Airline Route (3 of 11)

A recursive search strategy

```
To fly from the origin to the destination
{
   Select a city C adjacent to the origin
   Fly from the origin to city C
   if (C is the destination city)
      Terminate— the destination is reached
   else
      Fly from city C to the destination
}
```

Searching for an Airline Route (4 of 11)

- Possible outcomes of exhaustive search strategy
  1. Reach destination city, decide possible to fly from origin to destination
  2. Reach a city, *C* from which no departing flights
  3. You go around in circles
- Use backtracking to recover from a wrong choice (2 or 3)

Searching for an Airline Route (5 of 11)

Refinement of the recursive search algorithm

```
// Discovers whether a sequence of flights from originCity to destinationCity exists.
searchR(originCity: City, destinationCity: City): boolean
{
   Mark originCity as visited
   if (originCity is destinationCity)
      Terminate—the destination is reached
   else
      for (each unvisited city C adjacent to originCity)
         searchR(C, destinationCity)
}
```

Searching for an Airline Route (7 of 11)

ADT flight map operations

```
// Reads flight information into the flight map.
+readFlightMap(cityFileName: string, flightFileName: string): void

// Displays flight information.
+displayFlightMap(): void

// Displays the names of all cities that HPAir serves.
+displayAllCities(): void

// Displays all cities that are adjacent to a given city.
+displayAdjacentCities(aCity: City): void

// Marks a city as visited.
+markVisited(aCity: City): void

// Clears marks on all cities.
```

Searching for an Airline Route (8 of 11)

ADT flight map operations

```
+markVisited(aCity: City): void

// Clears marks on all cities.
+unvisitAll(): void

// Sees whether a city was visited.
+isVisited(aCity: City): boolean

// Inserts a city adjacent to another city in a flight map.
+insertAdjacent(aCity: City, adjCity: City): void

// Returns the next unvisited city, if any, that is adjacent to a given city.
// Returns a sentinel value if no unvisited adjacent city was found.
+getNextCity(fromCity: City): City

// Tests whether a sequence of flights exists between two cities.
+isPath(originCity: City, destinationCity: City): boolean
```

Searching for an Airline Route (9 of 11)

C++ implementation of **searchR**

```
/** Tests whether a sequence of flights exists between two cities.
 @pre  originCity and destinationCity both exist in the flight map.
 @post  Cities visited during the search are marked as visited
     in the flight map.
 @param originCity  The origin city.
 @param destinationCity  The destination city.
 @return  True if a sequence of flights exists from originCity
    to destinationCity; otherwise returns false. */
bool Map::isPath(City originCity, City destinationCity)
{
   // Mark the current city as visited
   markVisited(originCity);

   bool foundDestination = (originCity == destinationCity);
   if (!foundDestination)
   {
      // Try a flight to each unvisited city
```

Searching for an Airline Route (10 of 11)

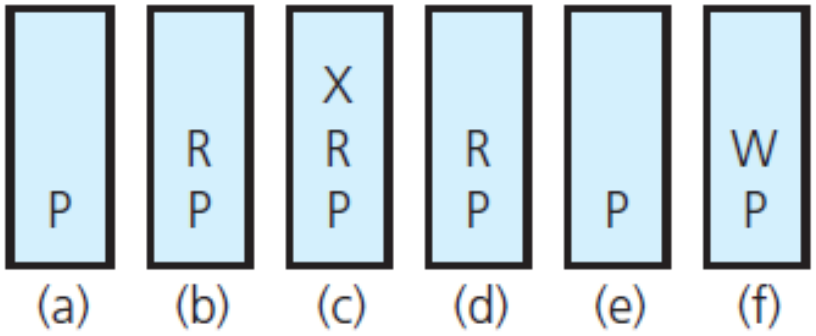
C++ implementation of **searchR**

```
{
      // Try a flight to each unvisited city
      City nextCity = getNextCity(originCity);
      while (!foundDestination && (nextCity != NO_CITY))
      {
         foundDestination = isPath(nextCity, destinationCity);
         if (!foundDestination)
            nextCity = getNextCity(originCity);
      }  // end while
   }  // end if

   return foundDestination;
}  // end isPath
```

Using Stack to Search a Flight Map (4 of 12)

- Strategy requires information about order in which it visits cities

Figure 6-7 The stack of cities as you travel from P to W

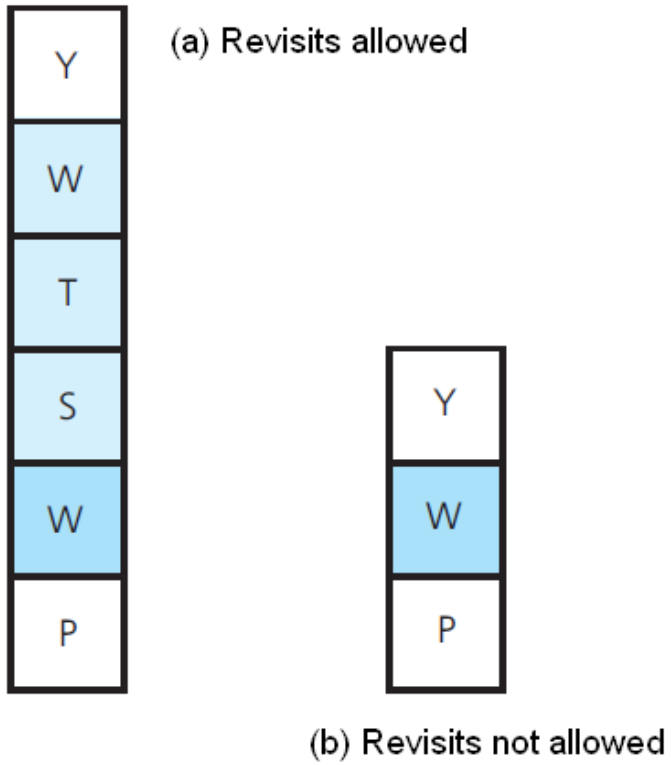


Using Stack to Search a Flight Map (5 of 12)

- Stack will contain directed path from
  - Origin city at bottom to …
  - Current visited city at top
- When to backtrack
  - No flights out of current city
  - Top of stack city already somewhere in the stack

Using Stack to Search a Flight Map (6 of 12)

Figure 6-8 The effect of revisits on the stack of cities



Using Stack to Search a Flight Map (7 of 12)

Final draft of algorithm.

```
// Searches for a sequence of flights from originCity to destinationCity
searchS(originCity: City, destinationCity: City): boolean
{
   cityStack = a new empty stack
   Clear marks on all cities

   cityStack.push(originCity) // Push origin onto the stack
   Mark the origin as visited

   while (!cityStack.isEmpty() and destinationCity is not at the top of the stack)
   {
      // Loop invariant: The stack contains a directed path from the origin city at
      // the bottom of the stack to the city at the top of the stack
      if (no flights exist from the city on the top of the stack to unvisited cities)
         cityStack.pop()          // Backtrack
      else
```

Using Stack to Search a Flight Map (8 of 12)

Final draft of algorithm.

```
            cityStack.pop()          // Backtrack
        else
        {
            Select an unvisited destination city C for a flight from the city on the top of the stack
            cityStack.push(C)
            Mark C as visited
        }
    }
    if (cityStack.isEmpty())
        return false // No path exists
    else
        return true  // Path exists
}
```

Using Stack to Search a Flight Map (9 of 12)

Figure 6-9 A trace of the search algorithm for the given flight map

Action	Reason	Contents of stack (bottom to top)
Push P	Initialize	P
Push R	Next unvisited adjacent city	P R
Push X	Next unvisited adjacent city	P R X
Pop X	No unvisited adjacent city	P R
Pop R	No unvisited adjacent city	P
Push W	Next unvisited adjacent city	P W
Push S	Next unvisited adjacent city	P W S
Push T	Next unvisited adjacent city	P W S T
Pop T	No unvisited adjacent city	P W S
Pop S	No unvisited adjacent city	P W
Push Y	Next unvisited adjacent city	P W Y
Push Z	Next unvisited adjacent city	P W Y Z

Using Stack to Search a Flight Map (10 of 12)

C++ implementation of **searchS**

```
bool Map::isPath(City originCity, City destinationCity)
{
   Stack cityStack;

   unvisitAll(); // Clear marks on all cities

   // Push origin city onto cityStack and mark it as visited
   cityStack.push(originCity);
   markVisited(originCity);

   City topCity = cityStack.peek();
   while (!cityStack.isEmpty() && (topCity != destinationCity))
   {
      // The stack contains a directed path from the origin city
      // at the bottom of the stack to the city at the top of the stack

      // Find an unvisited city adjacent to the city on the top of the stack
      City nextCity = getNextCity(topCity);

      if (nextCity == NO_CITY)
```

Using Stack to Search a Flight Map (11 of 12)

C++ implementation of **searchS**

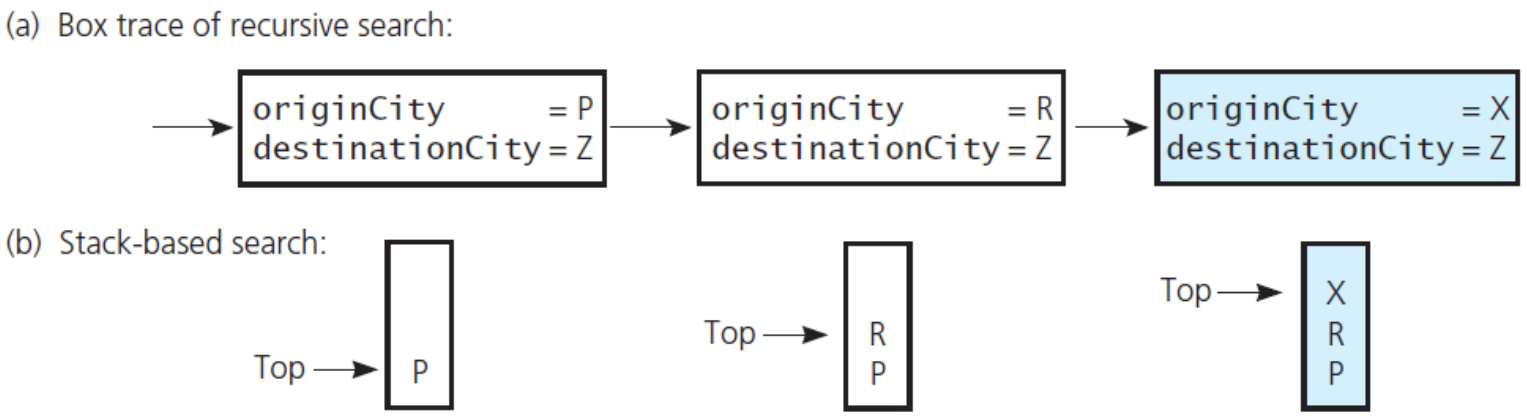
```
        City nextCity = getNextCity(topCity);

        if (nextCity == NO_CITY)
           cityStack.pop(); // No city found; backtrack
        else                // Visit city
        {
           cityStack.push(nextCity);
           markVisited(nextCity);
        }  // end if

        if (!cityStack.isEmpty())
           topCity = cityStack.peek();
     }  // end while

     return !cityStack.isEmpty();
  }  // end isPath
```

Relationship Between Stacks and Recursion (1 of 3)

- Key aspects of common strategy
  - Visiting a new city
  - Backtracking
  - Termination

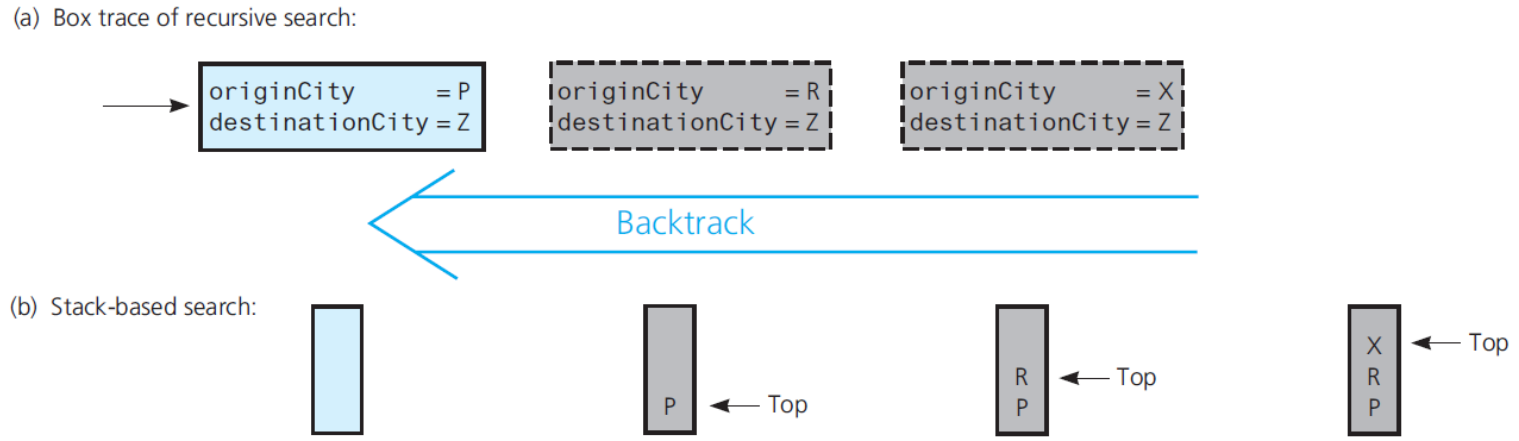
Relationship Between Stacks and Recursion (2 of 3)

Figure 6-11 Visiting city P, then R, then X: (a) box trace versus (b) stack



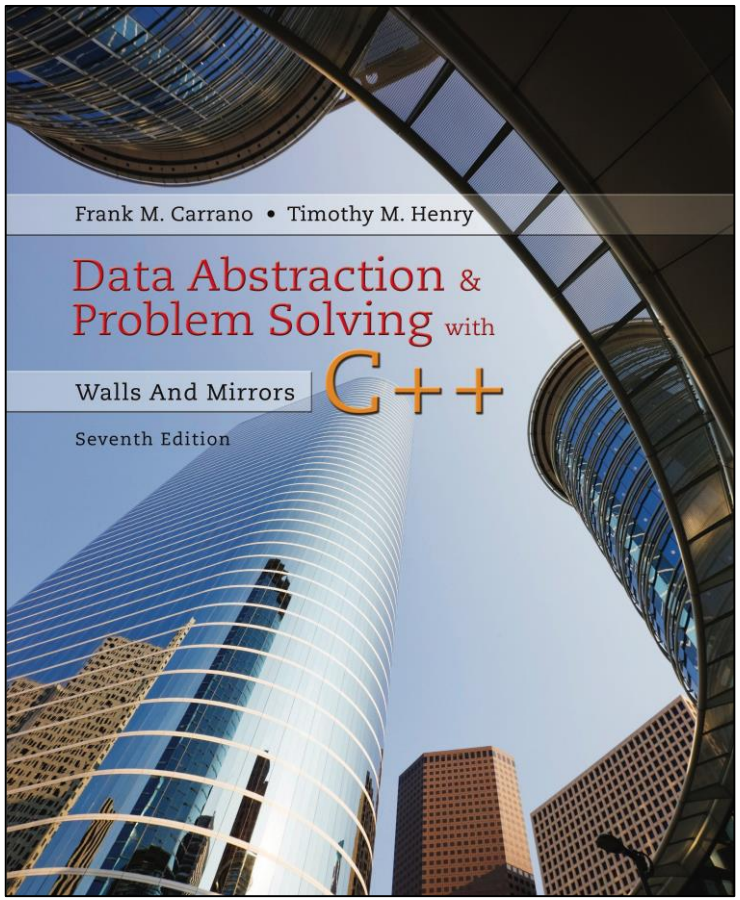
Relationship Between Stacks and Recursion (3 of 3)

Figure 6-12 Backtracking from city X to R to P: (a) box trace versus (b) stack



Data Abstraction & Problem Solving with C++: Walls and Mirrors

Seventh Edition

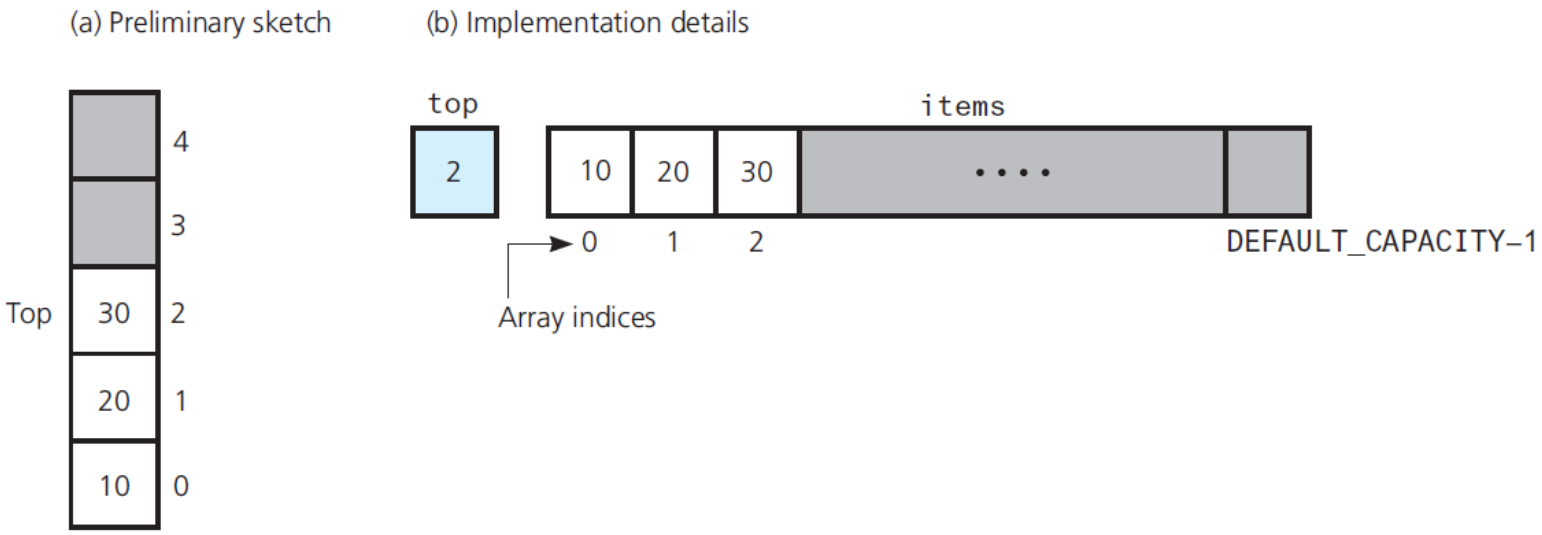


Chapter 7

Implementations of the ADT Stack

An Array-Based Implementation (1 of 7)

Figure 7-1 Using an array to store a stack's entries



An Array-Based Implementation (2 of 7)

Listing 7-1 The header file for an array-based stack

```
1  /** ADT stack: Array-based implementation.
2   @file ArrayStack.h */
3
4  #ifndef ARRAY_STACK_
5  #define ARRAY_STACK_
6
7  #include "StackInterface.h"
8
9  template<class ItemType>
10 class ArrayStack : public StackInterface<ItemType>
11 {
12 private:
13    static const int DEFAULT_CAPACITY = maximum-size-of-stack;
14    ItemType items[DEFAULT_CAPACITY]; // Array of stack items
15    int      top;                     // Index to top of stack
```

An Array-Based Implementation (3 of 7)

Listing 7-1 [Continued]

```
16  public:
17      ArrayStack();                          // Default constructor
18      bool isEmpty() const;
19      bool push(const ItemType& newEntry);
20      bool pop();
21      ItemType peek() const;
22  }; // end ArrayStack
23
24  #include "ArrayStack.cpp"
25  #endif
```

An Array-Based Implementation (4 of 7)

Listing 7-2 The implementation file for an array-based stack

```
1  /** @file ArrayStack.cpp */
2
3  #include <cassert>              // For assert
4  #include "ArrayStack.h"         // Header file
5
6  template<class ItemType>
7  ArrayStack<ItemType>::ArrayStack() : top(-1)
8  {
9  }  // end default constructor
10
11 // Copy constructor and destructor are supplied by the compiler
12
13 template<class ItemType>
14 bool ArrayStack<ItemType>::isEmpty() const
15 {
16    return top < 0;
17 }  // end isEmpty
18
19 template<class ItemType>
20 bool ArrayStack<ItemType>::push(const ItemType& newEntry)
21 {
```

An Array-Based Implementation (5 of 7)

Listing 7-2 [Continued]

```
21  {
22     bool result = false;
23     if (top < DEFAULT_CAPACITY - 1) // Does stack have room for newEntry?
24     {
25        top++;
26        items[top] = newEntry;
27        result = true;
28     }  // end if
29
30     return result;
31  }  // end push
32  template<class ItemType>
33  bool ArrayStack<ItemType>::pop()
34  {
35     bool result = false;
36     if (!isEmpty())
37     {
38        top--;
39        result = true;
40     }  // end if
```

An Array-Based Implementation (6 of 7)

Listing 7-2 [Continued]

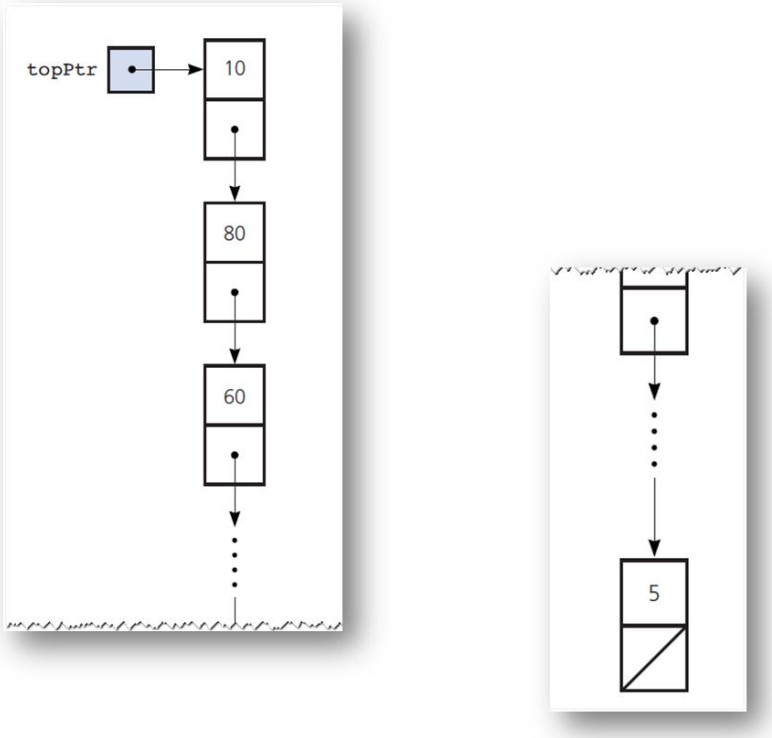
```
40      } // end if
41
42      return result;
43   } // end pop
44
45   template<class ItemType>
46   ItemType ArrayStack<ItemType>::peek() const
47   {
48      assert (!isEmpty()); // Enforce precondition during debugging
49
50      // Stack is not empty; return top
51      return items[top];
52   } // end peek
53   // end of implementation file
```

An Array-Based Implementation (7 of 7)

- Protecting the ADT's walls
  - Implement stack as a class
  - Declaring **items** and top as **private**
- Note
  - **push** receives **newEntry** as constant reference argument
  - **push** uses **newEntry** as an alias … no copy made

A Link-Based Implementation (1 of 9)

Figure 7-2 A link-based implementation of a stack



A Link-Based Implementation (2 of 9)

Listing 7-3 The header file for the class **LinkedStack**

```
1  /** ADT stack: Link-based implementation.
2   @file LinkedStack.h */
3
4  #ifndef LINKED_STACK_
5  #define LINKED_STACK_
6
7  #include "StackInterface.h"
8  #include "Node.h"
9
10 template<class ItemType>
11 class LinkedStack : public StackInterface<ItemType>
12 {
13 private:
14    Node<ItemType>* topPtr; // Pointer to first node in the chain;
15                            // this node contains the stack's top
16
```

A Link-Based Implementation (3 of 9)

Listing 7-3 [Continued]

```
15                                  // this node contains the stack's top
16
17  public:
18  // Constructors and destructor:
19     LinkedStack();                                        // Default constructor
20     LinkedStack(const LinkedStack<ItemType>& aStack); // Copy constructor
21     virtual ~LinkedStack();                               // Destructor
22
23  // Stack operations:
24     bool isEmpty() const;
25     bool push(const ItemType& newItem);
26     bool pop();
27     ItemType peek() const;
28  }; // end LinkedStack
29
30  #include "LinkedStack.cpp"
31  #endif
```

A Link-Based Implementation (4 of 9)

Listing 7-4 The implementation file for the class LinkedStack

```
1  /** @file LinkedStack.cpp */
2  #include <cassert>              // For assert
3  #include "LinkedStack.h"        // Header file
4
5  template<class ItemType>
6  LinkedStack<ItemType>::LinkedStack() : topPtr(nullptr)
7  {
8  } // end default constructor
9
10 template<class ItemType>
11 LinkedStack<ItemType>::LinkedStack(const LinkedStack<ItemType>& aStack)
12 {
13    // Point to nodes in original chain
14    Node<ItemType>* origChainPtr = aStack.topPtr;
```

A Link-Based Implementation (5 of 9)

Listing 7-4 [Continued]

```
15      if (origChainPtr == nullptr)
16         topPtr = nullptr;            // Original stack is empty
17      else
18      {
19         // Copy first node
20         topPtr = new Node<ItemType>();
21         topPtr->setItem(origChainPtr->getItem());
22
23         // Point to first node in new chain
24         Node<ItemType>* newChainPtr = topPtr;
25
26         // Advance original-chain pointer
27         origChainPtr = origChainPtr->getNext();
28
29         // Copy remaining nodes
30         while (origChainPtr != nullptr)
31         {
32            // Get next item from original chain
33            ItemType nextItem = origChainPtr->getItem();
34
```

A Link-Based Implementation (6 of 9)

Listing 7-4 [Continued]

```
34
35            // Create a new node containing the next item
36            Node<ItemType>* newNodePtr = new Node<ItemType>(nextItem);
37
38            // Link new node to end of new chain
39            newChainPtr->setNext(newNodePtr);
40
41            // Advance pointer to new last node
42            newChainPtr = newChainPtr->getNext();
43
44            // Advance original-chain pointer
45            origChainPtr = origChainPtr->getNext();
46         }  // end while
47         newChainPtr->setNext(nullptr); // Flag end of chain
48      }  // end if
49   }  // end copy constructor
50
51   template<class ItemType>
```

A Link-Based Implementation (7 of 9)

Listing 7-4 [Continued]

```
50
51  template<class ItemType>
52  LinkedStack<ItemType>::~LinkedStack()
53  {
54     // Pop until stack is empty
55     while (!isEmpty())
56        pop();
57  }  // end destructor
58
59  template<class ItemType>
60  bool LinkedStack<ItemType>::push(const ItemType& newItem)
61  {
62     Node<ItemType>* newNodePtr = new Node<ItemType>(newItem, topPtr);
63     topPtr = newNodePtr;
64     newNodePtr = nullptr;
65     return true;
66  }  // end push
67
```

A Link-Based Implementation (8 of 9)

Listing 7-4 [Continued]

```
68  template<class ItemType>
69  bool LinkedStack<ItemType>::pop()
70  {
71     bool result = false;
72     if (!isEmpty())
73     {
74        // Stack is not empty; delete top
75        Node<ItemType>* nodeToDeletePtr = topPtr;
76        topPtr = topPtr->getNext();
77
78        // Return deleted node to system
79        nodeToDeletePtr->setNext(nullptr);
80        delete nodeToDeletePtr;
81        nodeToDeletePtr = nullptr;
82
83        result = true;
84     }  // end if
85
```

A Link-Based Implementation (9 of 9)

Listing 7-4 [Continued]

```
86      return result;
87   }  // end pop
88
89   template<class ItemType>
90   ItemType LinkedStack<ItemType>::peek() const
91   {
92      assert(!isEmpty()); // Enforce precondition during debugging
93
94      // Stack is not empty; return top
95      return topPtr->getItem();
96   }  // end peek
97
98   template<class ItemType>
99   bool LinkedStack<ItemType>::isEmpty() const
100  {
101     return topPtr == nullptr;
102  }  // end isEmpty
103  // end of implementation file
```

Implementations That Use Exceptions (1 of 3)

- Method **peek** does not expect client to look at top of an empty stack
  - **assert** statement merely issues error message, and halts execution
- Consider having **peek** throw an exception

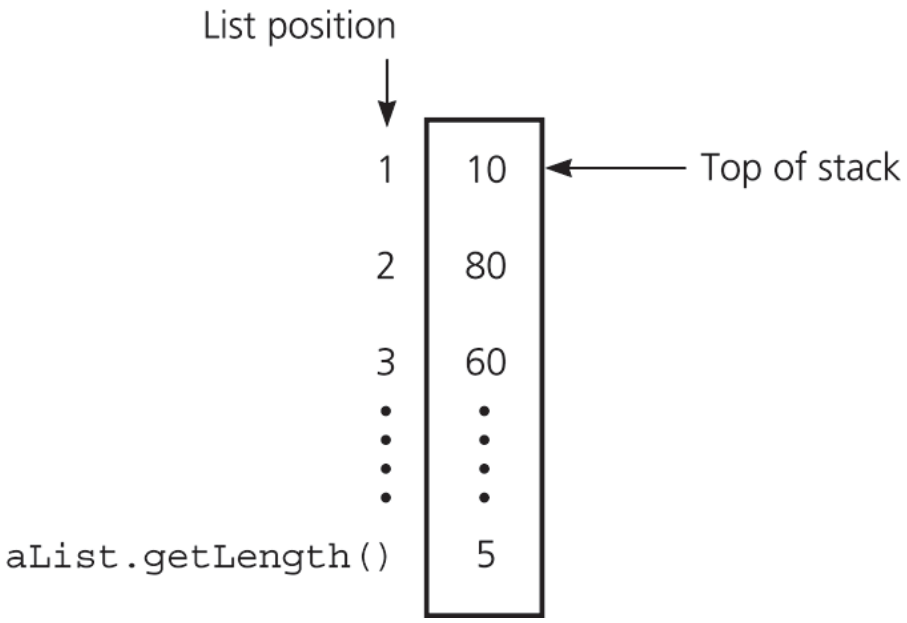
An Implementation That Uses ADT List

- The ADT list can be used to represent items in a stack
- If the item in position 1 is the top

`push(newItem)`
  - `insert(1, newItem)`

`pop()`
  - `remove(1)`

`peek()`
  - `getEntry(1)`



Comparing Implementations

- Fixed size versus dynamic size
  - An array-based implementation
    - Prevents the `push` operation from adding an item to the stack if the stack's size limit has been reached
  - A pointer-based implementation
    - Does not put a limit on the size of the stack
- An implementation that uses a linked list versus one that uses a pointer-based implementation of the ADT list
  - ADT list approach reuses an already implemented class
    - Much simpler to write
    - Saves time

Copyright