Chapter 6:
Stacks

Data Abstraction & Problem Solving with C++
Fifth Edition
by Frank M. Carrano

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

The Abstract Data Type

- Specifications of an abstract data type for a particular problem
  - Can emerge during the design of the problem's solution
  - Examples
    - `readAndCorrect` algorithm
    - `displayBackward` algorithm

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Developing an ADT During the Design of a Solution

- ADT stack operations
  - Create an empty stack
  - Destroy a stack
  - Determine whether a stack is empty
  - Add a new item to the stack
  - Remove the item that was added most recently
  - Retrieve the item that was added most recently

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Developing an ADT During the Design of a Solution

- A stack
  - Last-in, first-out (LIFO) property
    - The last item placed on the stack will be the first item removed
  - Analogy
    - A stack of dishes in a cafeteria

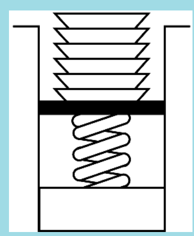

Figure 6-1
Stack of cafeteria dishes

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Refining the Definition of the ADT Stack

- Operation Contract for the ADT Stack

```
isEmpty():boolean {query}
push(in newItem:StackItemType)
        throw StackException
pop() throw StackException
pop(out stackTop:StackItemType)
        throw StackException
getTop(out stackTop:StackItemType) {query}
        throw StackException
```

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Using the ADT Stack in a Solution

- A program can use a stack independently of the stack's implementation
  - `displayBackward` and `readAndCorrect` algorithms can be refined using stack operations
- Use axioms to define an ADT stack formally
  - Example: Specify that the last item inserted is the first item to be removed

    `(aStack.push(newItem)).pop()= aStack`

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces
  - An example of balanced braces

    abc{defg{ijk}{l{mn}}op}qr
  - An example of unbalanced braces

    abc{def}}{ghij{kl}m

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Checking for Balanced Braces

- Requirements for balanced braces
  - Each time you encounter a "}", it matches an already encountered "{"
  - When you reach the end of the string, you have matched each "{"

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Checking for Balanced Braces

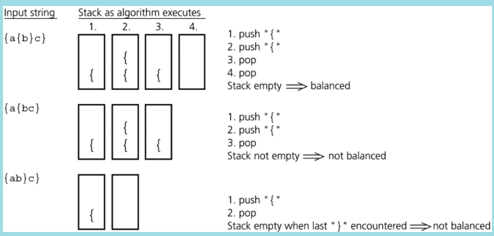

Figure 6-3

Traces of the algorithm that checks for balanced braces

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Recognizing Strings in a Language

- $L = \{w\$w' : w$ is a possibly empty string of characters other than \$, $w' = \text{reverse}(w) \}$
- A solution using a stack
  - Traverse the first half of the string, pushing each character onto a stack
  - Once you reach the \$, for each character in the second half of the string, match a popped character off the stack

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Implementations of the ADT Stack

- The ADT stack can be implemented using
  - An array
  - A linked list
  - The ADT list
- All three implementations use a `StackException` class to handle possible exceptions

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Implementations of the ADT Stack

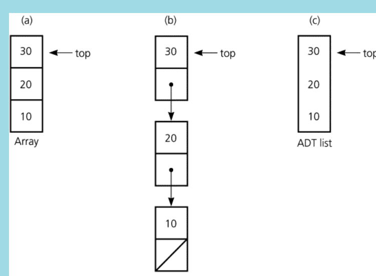

Figure 6-4

Implementations of the ADT stack that use (a) an array; (b) a linked list; (c) an ADT list

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

An Array-Based Implementation of the ADT Stack

- Private data fields
  - An array of `items` of type `StackItemType`
  - The index `top` to the top item
- Compiler-generated destructor and copy constructor

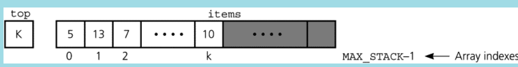

Figure 6-5

An array-based implementation

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

A Pointer-Based Implementation of the ADT Stack

- A pointer-based implementation
  - Enables the stack to grow and shrink dynamically
- `topPtr` is a pointer to the head of a linked list of items
- A copy constructor and destructor must be supplied

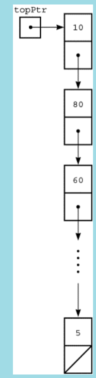

Figure 6-6 A pointer-based implementation

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

An Implementation That Uses the ADT List

- The ADT list can represent the items in a stack
- Let the item in position 1 of the list be the top
  - `push(newItem)`
    - `insert(1, newItem)`
  - `pop()`
    - `remove(1)`
  - `getTop(stackTop)`
    - `retrieve(1, stackTop)`

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

An Implementation That Uses the ADT List

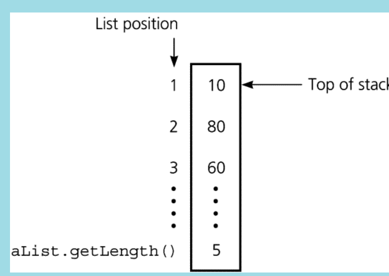

Figure 6-7

An implementation that uses the ADT list

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Comparing Implementations

- Fixed size versus dynamic size
  - A statically allocated array-based implementation
    - Fixed-size stack that can get full
    - Prevents the `push` operation from adding an item to the stack, if the array is full
  - A dynamically allocated array-based implementation or a pointer-based implementation
    - No size restriction on the stack

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Comparing Implementations

- A pointer-based implementation vs. one that uses a pointer-based implementation of the ADT list
  - Pointer-based implementation is more efficient
  - ADT list approach reuses an already implemented class
    - Much simpler to write
    - Saves programming time

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

The STL Class *`stack`*

- Provides a size function
- Has two data type parameters
  - *`T`*, the data type for the stack items
  - *`Container`*, the container class that the STL uses in its implementation of the stack
- Uses the keyword `explicit` in the constructor's declaration to prevent use of the assignment operator to invoke the constructor

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Application: Algebraic Expressions

- When the ADT stack is used to solve a problem, the use of the ADT's operations should not depend on its implementation
- To evaluate an infix expression
  - Convert the infix expression to postfix form
  - Evaluate the postfix expression

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Evaluating Postfix Expressions

- A postfix calculator
  - When an operand is entered, the calculator
    - Pushes it onto a stack
  - When an operator is entered, the calculator
    - Applies it to the top two operands of the stack
    - Pops the operands from the stack
    - Pushes the result of the operation onto the stack

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Evaluating Postfix Expressions

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

Figure 6-8

The action of a postfix calculator when evaluating the expression 2 * (3 + 4)

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Evaluating Postfix Expressions

- To evaluate a postfix expression entered as a string of characters
  - Use the same steps as a postfix calculator
  - Simplifying assumptions
    - The string is a syntactically correct postfix expression
    - No unary operators are present
    - No exponentiation operators are present
    - Operands are single lowercase letters that represent integer values

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Converting Infix Expressions to Equivalent Postfix Expressions

- You can evaluate an infix expression by first converting it into an equivalent postfix expression
- Facts about converting from infix to postfix
  - Operands always stay in the same order with respect to one another
  - An operator will move only “to the right” with respect to the operands
  - All parentheses are removed

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Converting Infix Expressions to Equivalent Postfix Expressions

- Steps as you process the infix expression:
  - Append an operand to the end of an initially empty string `postfixExpr`
  - Push (onto a stack
  - Push an operator onto the stack, if stack is empty; otherwise pop operators and append them to `postfixExpr` as long as they have a precedence >= that of the operator in the infix expression
  - At), pop operators from stack and append them to `postfixExpr` until (is popped

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Converting Infix Expressions to Equivalent Postfix Expressions

ch	Stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators from stack to **postfixExp** until "("
	– (	abcd*+	
	–	abcd*+	
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from stack to **postfixExp**
		abcd*+e/–	

Figure 6-9

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

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Application: A Search Problem

- High Planes Airline Company (HPAir)
  - For each customer request, indicate whether a sequence of HPAir flights exists from the origin city to the destination city
- 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

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Application: A Search Problem

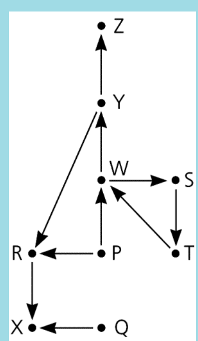

Figure 6-10

Flight map for HPAir

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

A Nonrecursive Solution That Uses a Stack

- The solution performs an exhaustive search
  - Beginning at the origin city, the solution will try every possible sequence of flights until either
    - It finds a sequence that gets to the destination city
    - It determines that no such sequence exists
- Backtracking can be used to recover from choosing a wrong city

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

A Nonrecursive Solution That Uses a Stack

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

Figure 6-13

A trace of the search algorithm, given the flight map in Figure 6-10

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

A Recursive Solution

- Possible outcomes of the recursive search strategy
  - You eventually reach the destination city and can conclude that it is possible to fly from the origin to the destination
  - You reach a city *C* from which there are no departing flights
  - You go around in circles

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

A Recursive Solution

- A refined recursive search strategy

```
+searchR(in originCity:City,
         in 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)
```

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

The Relationship Between Stacks and Recursion

- Typically, stacks are used by compilers to implement recursive methods
  - During execution, each recursive call generates an activation record that is pushed onto a stack
- Stacks can be used to implement a nonrecursive version of a recursive algorithm

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Summary

- ADT stack operations have a last-in, first-out (LIFO) behavior
- Stack applications
  - Algorithms that operate on algebraic expressions
  - Flight maps
- A strong relationship exists between recursion and stacks

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.