Chapter 11

The Stack

G O A L S

After working through this chapter, you will be able to:

Implement and use a stack data structure.

Understand, use, and evaluate expressions in infix and postfix notations.

OVERVIEW

After arrays, the stack is one of the most fundamental data structures in computer science. A stack is a restricted kind of list. A general list represented by an array lets you add and remove elements at the top and bottom of the list, as well as anywhere in between. You can access any element of the list by using array subscripts, and can replace any element using an assignment statement. A stack is like a list in that it contains an ordered sequence of elements. However, the operations that you can perform on a stack are fewer and less powerful than those supported by general lists. With a stack, you can only add and remove elements at one end (called the top of the stack). The only element of a stack you can access is the element at the top, and then you can only do so after you have removed it from the stack.

A close relative of the stack is a data structure called the queue. The word "queue" means a line in which people or things wait for some kind of service or processing. A checkout line at a supermarket is a perfect example of a queue. Queues differ from stacks in only one respect: elements are added to a queue at one end (the back), and are removed from the other end (the front). For example, when you get in line at the supermarket, you add yourself to the line at the end. You have to be at the front of the line before you can pay and leave.

You might be wondering what possible advantage there could be in giving up the power of arrays for the limited capabilities of a stack. The answer is that sometimes a stack provides all the power and flexibility you need. The stack is an abstraction, a pattern that arises again and again in the way programs store and operate on data. The stack proves to be the correct tool for the solution of many different kinds of problems. Programmers use stacks extensively when writing interpreters for computer languages such as Visual Basic itself. In this chapter, you will be learn to use a stack to interpret a very small language of simple arithmetic expressions.

The Stack Data Structure

A stack is list with a difference. A stack is a list from which your program can only access a single element at the top. New elements are pushed onto the top of the stack. At any time, the only element that can be removed—or popped—from the stack is the element at the top. See Figure 11-1.

The rack of trays in a cafeteria line is often designed as a spring-loaded stack. The clean trays are pushed onto the top of the stack. The spring in the bottom of the stack compresses to allow the more trays to be added. People going through the line take a tray from the top of the stack. The first trays pushed onto the stack are the last trays removed from the stack. The last trays pushed on top of the stack are the first trays removed.

If the number of trays exceeds the capacity of the rack, the cafeteria staff has stack overflow. You may have seen a message about the same kind of problem when a program goes awry, or a game program overloads the operating system of the computer. Such error messages reveal a hidden truth of computers: underneath the surface of the computer are stacks of information.

A stack can be represented as an array with a pointer pointing to the top of the stack. Access to the elements of the stack is entirely through this pointer. The pointer is in fact just an array subscript. Random access to the elements of the stack, although possible, is not consistent with the concept of a stack.

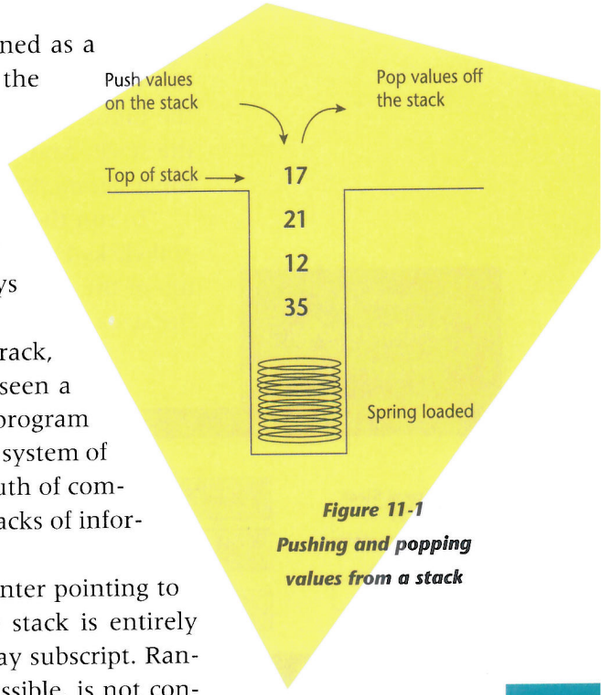


Figure 11-1
Pushing and popping values from a stack

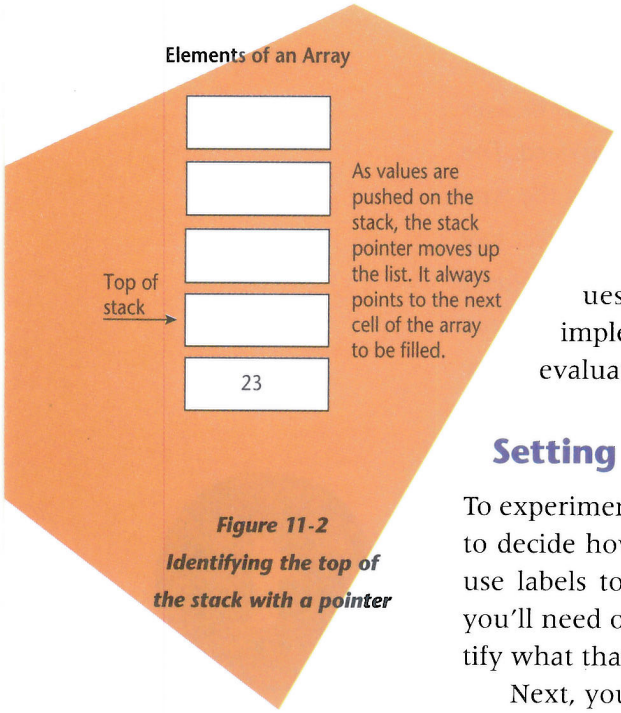


Figure 11-2
Identifying the top of the stack with a pointer

Implementing a stack is easy. You start with an array. Add an integer variable that points to a place in the array called the top of the stack. The top of the stack changes as the number of elements in the stack increases and decreases. See Figure 11-2.

In this section you will write a simple stack program. Values are pushed and popped. This program shows how to implement a stack. Later in the chapter you will use a stack to evaluate expressions.

Setting Up the Form

To experiment with a stack, you need to set up the array, then you need to decide how to display the top entry in the stack. As you know, you use labels to display information calculated during the program. So, you'll need one label to display information, and another label to identify what that information is.

Next, you want users to enter information to be added to the stack. For that purpose, you need a textbox and a label to prompt data entry. After a user has entered data into the textbox, something must happen. What do you want to happen to the user input?

You can add command buttons (or menu commands, for that matter) to perform actions with the user input. Clicking on Push will push the input onto the stack. Clicking on Pop will display the top value in the stack. And, of course, you need an Exit button for the user to close the program. See Figure 11-3.

To run the program, a user enters a whole number in the Data box and clicks Push. The label marked Top of Stack shows the value at the top of the stack. In the figure, the value 41 has just been popped from the stack. The new top of the stack, 33, is displayed.

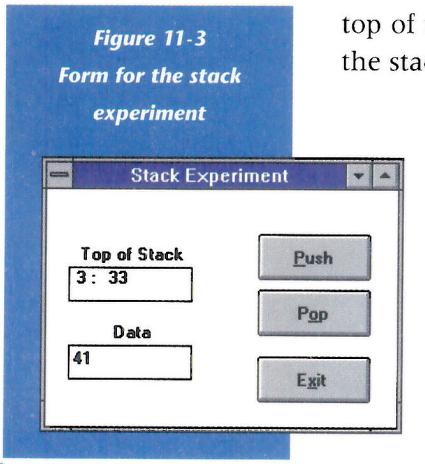


Figure 11-3
Form for the stack experiment

Follow these steps to set up the form shown in Figure 11-3.

1. Start Visual Basic. If Visual Basic is running, choose New Project from the File menu.
2. Change the caption of the file to **Stack Experiment**.
3. Place three command buttons on the form. Change the names and captions as shown.

Caption	*Name*
&Push	cmdPush
P&op	cmdPop
E&xit	cmdExit

4 Place three labels and one text box on the form. Alter the properties of the labels according to the following table.

Name	*Caption*	*Other Properties*
label1	Top of Stack	AutoSize = True
label2	Data	AutoSize = True
lblTOS	*none*	BorderStyle = 1-single
txtData	*none*	

5 Save the form and project files.

Adding Code to the General Declarations

To make the stack available to all the event procedures, you need to declare it in the general declarations section of the form. A stack is composed of an array and a top-of-stack pointer. You declare both.

Enter the following lines in the general declarations section of the form:

```
Option Explicit
Dim Stack(0 To 20) As Integer
Dim TOS As Integer
```

For this implementation, you will use a stack of integers. In a real application, you are just as likely to use a stack of characters or real numbers. The range of subscripts in this example is 0 to 20. The variable *TOS* points to the top of the stack. The value of TOS is initialized to 0.

You will create general functions of the main form that implement the **Push** and **Pop** procedures. This separates the general logic of pushing and popping items from the specific details of how this program obtains values and displays them.

Coding the Pop Function

The **Pop** function is not an event procedure of a control. You put the function's code in the general section of the form. The routine pops an item from the top of the stack and returns that item as the value of the function. A function value can be assigned to a variable, displayed in textbox, or stand in wherever a value is used in an expression.

TOS normally points at the next empty space in the array. To pop a value from the array, subtract 1 from *TOS* to point it at the top value in the stack. When a program tries to pop a value from an empty stack, the error condition is called stack underflow.

In the **Pop** function, the statement **Pop = Stack(TOS)** assigns the stack value to the function name, providing the function with the value it returns to the calling program. See Figure 11-4.

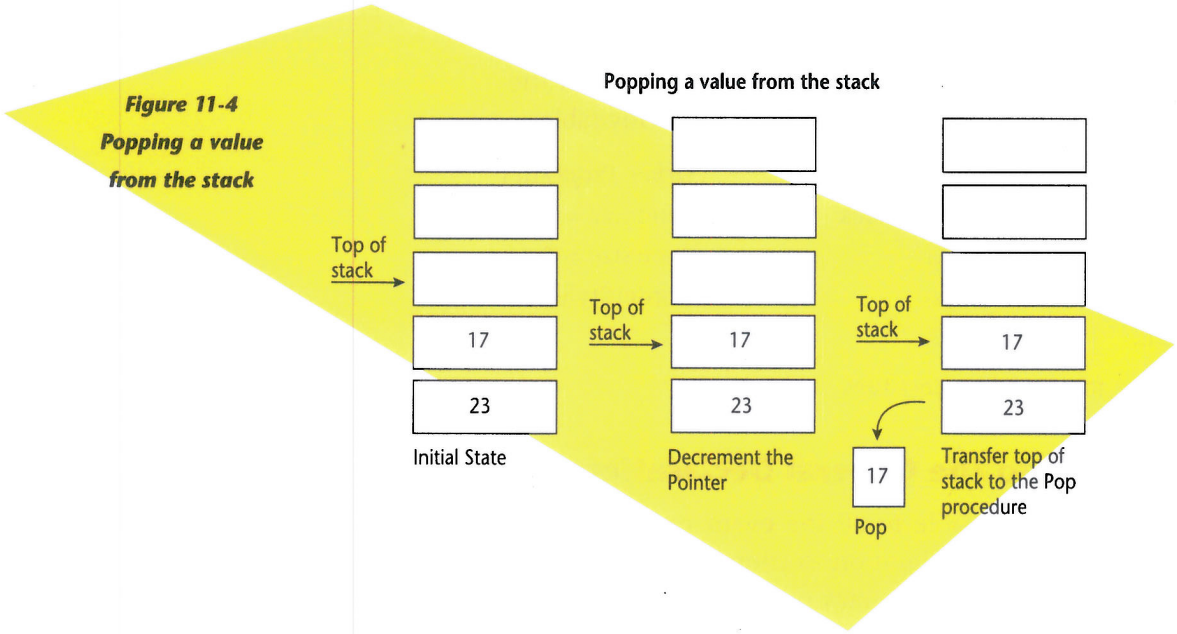


Figure 11-4
Popping a value from the stack

Follow these steps to enter the code for the **Pop** function.

1. Open the Code window for the default form.
2. Select New Procedure from the View menu.
3. Enter the name of the function, **Pop**. Click on the Function option button, then click OK. Visual Basic puts you into the skeleton routine for **Pop**.
4. Enter the following code to implement the **Pop** function:

```
Function Pop () As Integer
   '—Error message for stack underflow condition.
   Const strUnderflow = "Stack underflow"
   '—Legal stack value is tested, if less than or equal
   'to 0, the stack is empty, a value cannot be popped
   If TOS > 0 Then
      '—Decrement the top-of-stack pointer
      TOS = TOS - 1
      '—Transfer data to the function name
      Pop = Stack(TOS)
   Else
      MsgBox strUnderflow
   End If
End Function
```

The **Pop** function is not yet connected to the command button, Pop. The code in cmdPop_Click calls the function **Pop** to collect a value from the stack.

Coding the Push Subroutine

The overall structure of the **Push** subroutine is very similar to that of the **Pop** function. Convention dictates the use of a subroutine for pushing and a function for popping. It need not be so, but it does make sense. **Pop** returns a value from the stack to the calling program. It is the job of a function to return values. See Figure 11-5.

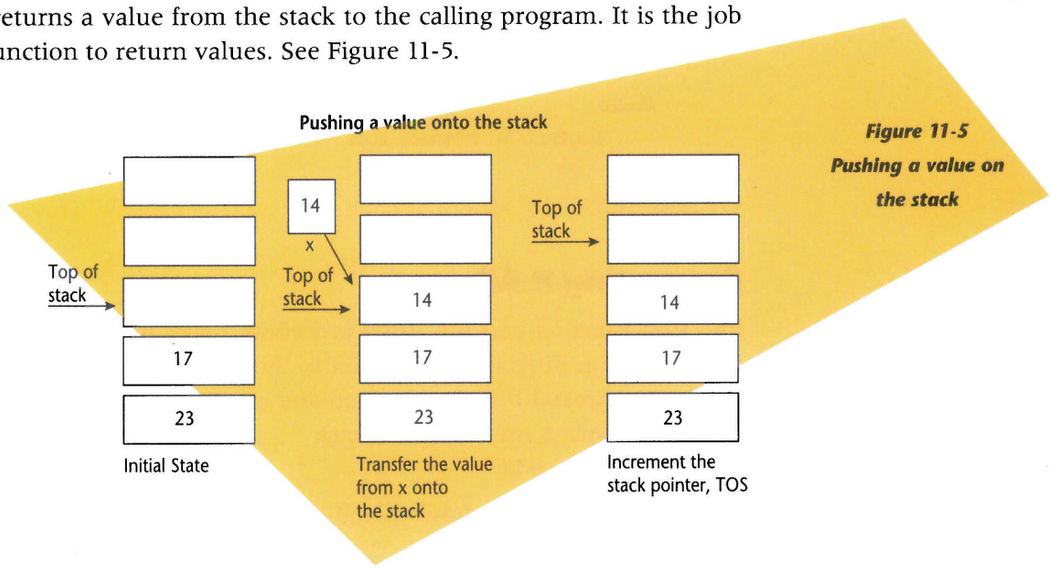


Figure 11-5
Pushing a value on the stack

Push has a single parameter, *x*, the value to be pushed on the stack. It returns no value at all. Using a function for this routine makes little sense.

Push checks for an error condition called stack overflow. This occurs when a program tries to push a value onto a stack that's full. Often this happens when a program using a stack has "run amok"—repeatedly pushing values onto a stack without ever removing those values. This is an error condition that usually crashes the computer.

Follow these steps to enter the **Push** procedure:

1. Open the Code window for the default form.
2. Select New Procedure from the View menu.
3. Enter the name of the procedure, **Push**, and click the Sub option. Visual Basic puts you into the skeleton routine for **Push**.
4. Enter the following code to implement the **Push** function:

```
Sub Push (x As Integer)
   '-Error message for stack overflow condition
   Const strOverflow = "Stack overflow"
```

continued

```
    '—If TOS is less than 20, there is still room
    'in the stack for more values
    If TOS < 20 Then
       '—Use TOS, then add one to point it to the next
       'empty space in the array
       Stack(TOS) = x
       TOS = TOS + 1
    Else
       MsgBox strOverflow
    End If
End Sub
```

Connecting It All

The **Pop** function and the **Push** procedure need drivers to call them from the main program. In addition to those procedures, connected to the two command buttons, cmdPop and cmdPush, the procedure DisplayStack displays the top of the stack.

To enter the code:

1 Open the Code window for cmdPop. Enter these lines:

```
txtData = Pop()     ' read value from stack
DisplayStack     ' show top-of-stack on form
txtData.SetFocus     ' ready for more data
```

2 Open the Code window for cmdPush. Enter these lines:

```
'—Check for problem with value in textbox.
On Error GoTo ErrorHandler
Dim x As Integer
x = Val(txtData)
'—Call Push to put value on stack.
Push (x)
DisplayStack
'—Clear item and set focus to pick up more.
txtData = ""
txtData.SetFocus
Exit Sub
ErrorHandler:
   MsgBox "Problem with the data. Error #: " & Err
   Exit Sub
```

3 Enter the code to display the top of the stack. Remember, *TOS* points to the *next* open space. Either choose New Procedure from the View menu, or enter these lines directly in the general section. Visual Basic automatically gives you a procedure skeleton.

```
Sub DisplayStack ()
   On Error GoTo TOSOutOfRange
   lblTOS = Str$(TOS - 1) & " : " & Str$(Stack(TOS - 1))
   Exit Sub
   TOSOutOfRange:
      MsgBox "Subscript is out of range: empty stack? Error #: " & Err
      Exit Sub
End Sub
```

4 Enter the **End** statement in the procedure for cmdExit.

5 Run the program. Enter a series of values by entering an integer in the Data box and clicking on Push. Pop values by clicking on the Pop button. Test the error checking by pushing too many values and popping too many values. Type a string into the Data box and click on Push.

6 Exit the program. Save the form and project files.

QUESTIONS AND ACTIVITIES

1. Random access to the contents of memory cells means a program may get a value from any area of an array. What characteristic of a stack deliberately ignores random access?
2. You may have bought candy in a PEZ dispenser. How is this candy dispenser like a stack?
3. A southbound train wants to head back north. The cars of the train will remain oriented in the same direction, but the engine needs to be turned around and reattached at the other end. Assuming you can detach the engine from the train and still move the cars of the train with a spare engine, sketch a track that would allow the main engine to be repositioned at the other end of the train.
4. What is stack overflow? Stack underflow? Remove the error handlers in the **Pop** routine and run the Stack Experiment project. Push three values, then pop four. What error message is displayed?
5. Sketch the setup for a stack. Label the stack pointer and show where the next element of the stack will be put.

6. Some implementations of stacks use a stack pointer that actually points to the item at the top of the stack, *not* the next open position of the stack. Rewrite both the **Push** subroutine and the **Pop** function using this convention.
7. In the **Push** subroutine you wrote above, under what condition can a stack overflow not occur? In the **Pop** routine you wrote above, under what condition can a stack underflow not occur?
8. Why is it appropriate to use a function subprogram for popping the stack and a subroutine to push a value on the stack?
9. In the Stack Experiment project, change the Enabled property of the cmdPop to **False**. Add code to change the property to **True** when the stack is not empty. Change the property to **False** when the last value is popped from the stack. Add code to change the Enabled property of cmdPush to **False**, when the stack is full.
10. Add a command button and a multiline textbox to the Stack Experiment project. Change the caption of the button to **&Display**. When the button is pushed, display the contents of the stack horizontally in the textbox. The top of the stack should be to the right.

Section

Infix and Postfix Notation

In this section, you use a stack to evaluate the value of an arithmetic expression. The elements of the stack are constantly displayed. The evaluation of expressions with stacks is an important part of computer science.

You will also see that the way we write arithmetic expressions is not the only possible one, nor is it the easiest for a computer to understand. We use what is known as infix notation, so called because the operators (the signs for addition, multiplication, and so on) appear in between their operands (the subexpressions to which operators are applied). One alternative notation is postfix notation. In this notation, operators always appear after their operands. In postfix notation, no parentheses are ever needed to indicate how subexpressions should be grouped. Infix notation is easier for humans to read, write, and understand, probably because it more closely resembles the languages we speak. However, postfix notation is much easier for computers to evaluate.

Although you won't learn how to do it in this course, it is certainly possible to write a program that converts infix expressions to equivalent

postfix expressions. Not surprisingly, any program that performed this conversion would use a stack.

Infix Notation

Infix notation is the common notation used to express arithmetic operations. In this notation, the operators, +,-,*,/,^, ..., stand between the operands:

a + b
r - t
*s * y ^ 2*

A whole set of rules govern the evaluation of such expressions. The operands are the values being manipulated in the expression, and an operator is the operation performed on the operands. Operators that manipulate two values, such as addition or multiplication, are called binary operators. An operator that works on a single operand is called an unary operator. A minus sign in front of a number, such as -5, is a unary operator.

Postfix Notation

In postfix notation the operator follows the operands:

a b + means: *a + b*
*b b * 4 a * c * -* means: *b*b - 4*a*c*

The use of postfix notation simplifies some important processes. For instance, evaluating expressions in postfix is easier for a computer than evaluating an expression in infix. The rules governing the order of operations for infix expressions are hard for a computer. "Please Excuse My Dear Aunt Sally" is fine for humans, but computers, by their nature, evaluate an expression by scanning its operands and operators sequentially, from one end of the expression to the other.

Postfix comes to the rescue: there is only one rule for the order of operations in postfix expressions: left to right. This rule makes postfix notation ideal for computers.

EVALUATING A POSTFIX EXPRESSION

To evaluate a postfix expression, you use a stack to hold values of subexpressions. Follow these rules:

1. Move from the left to the right.
2. If you encounter an operand, push it on the stack.

3. If you encounter a unary operator:
   - ◎ Pop the top of the stack: this is the operand for the operator.
   - ◎ Apply the operator to the operand.
   - ◎ Push the result on the stack.
4. If you encounter a binary operator:
   - ◎ Pop the top of the stack: this is the second operand.
   - ◎ Pop the top of the stack again: this is the first operand.
   - ◎ Apply the operator to the operands.
   - ◎ Push the result on the stack.

In a properly formed postfix expression, when the entire expression is completely evaluated from left to right, the value on top of the stack is the value of the expression.

AN EXAMPLE

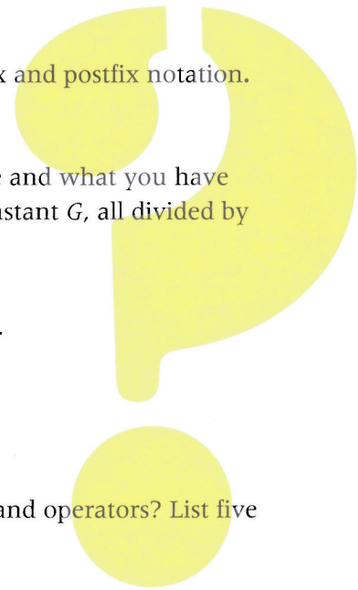
Here is an example of using postfix notation. For the sake of simplicity, use single-digit operands. Look at the step-by-step process for evaluating the expression *5 6 * 4 2 / 3 * -*. Process the characters one at a time from left to right.

1. The first character is an operand, 5, push it on the stack. Top of Stack: 5
2. The next character is an operand, 6, push it on the stack. Top of Stack: 6 5
3. The next character is an operator, *:
   a) Pop the top of the stack into operand2. op2 = 6; Top of Stack: 5
   b) Pop the top of the stack into operand1. op1 = 5 op2 = 6; Top of Stack: empty
   c) Apply the operator; op1 * op2 = 30
   d) Push result onto the stack; Top of Stack: 30
4. The next character is an operand, 4. Push it on the stack. Top of Stack: 4 30
5. The next character is an operand, 2. Push it on the stack. Top of Stack: 2 4 30
6. The next character is an operator, /. Pop twice, apply the operator, and push the result. 4 / 2 = 2, push; Top of Stack: 2 30
7. The next character is an operand, 3. Push it on the stack. Top of Stack: 3 2 30
8. The next character is an operator, *. Pop twice, apply the operator, and push the result. 2 * 3 = 6, push; Top of Stack: 6 30

9. The last character is an operator, -. Pop twice, apply the operator, and push the result. 30 - 6 = 24, push; Top of Stack: 24

The evaluation of the expression is complete. The value of the expression is 24. In the next section, you write a program using these rules to evaluate postfix expressions.

QUESTIONS AND ACTIVITIES



1. Express the following phrases in both infix and postfix notation.
   a) The sum of *x* and *y*
   b) The product of *a* and (*b* plus *c*)
   c) The difference between what you owe and what you have
   d) The product of *mass1*, *mass2*, and a constant *G*, all divided by the distance *r*, squared
   e) The average of *F* and *G*
2. Translate the following into infix notation.
   a) *a b c* * -
   b) *a b c* - *
   c) *a b* * *c d* - /
   d) *a b* * *c* / *d* - *f e* + *
3. What is the difference between operands and operators? List five common operators.
4. What is a unary operator?
5. What is the order of operations for postfix expressions?
6. Evaluate this expression (by hand) using a stack:
   5 6 * 7 - 2 3 + * /

   What error message do you generate?

The Visual Stack

This program evaluates a postfix expression using a stack, and shows what's happening each step of the way. The contents of the stack, the position of the stack pointer, the values of the operands, the value of the result, and the characters being processed are all visible through each step of the program. See Figure 11-6.

The goal of this program is to use a stack to evaluate a postfix expression. In addition, the program

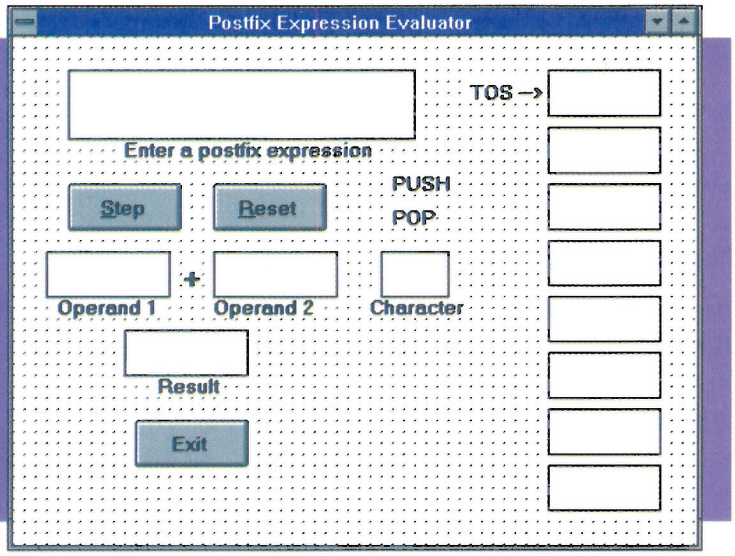


Figure 11-6
Form for the Postfix Expression Evaluator program

- Displays the entire stack
- Shows operands being popped
- Shows operators applied to operands
- Shows results pushed onto the stack

Starting Out

Think through the form shown in Figure 11-7. A user enters a postfix expression in the textbox at the top of the form. The labels at the right side of the form are used to display the contents of the stack and the position of the top of stack pointer, *TOS*. Two command buttons have been provided for users to step through the processing of the postfix expression.

The label boxes below the command buttons display the first and second operands, and below those there is a label that displays the result of the operation. The operation itself is displayed in a label between the operands which is updated to show the actual operation being performed.

Alongside the command buttons, the labels POP and PUSH are visible when the stack is accessed. A label marked Character shows the character of the input string that is being processed. Clicking on an Exit button at the bottom of the form ends the program.

Figure 11-7 shows the form you are going to build during the processing of a postfix expression. This is the same expression used in the last section.

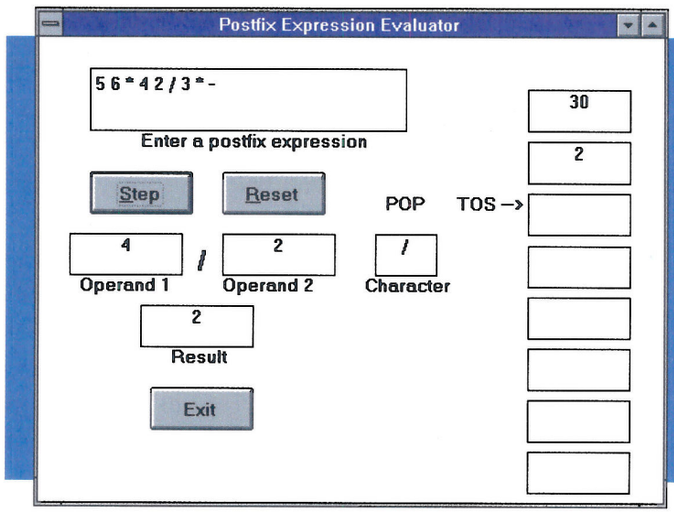


Figure 11-7
Postfix form with values displayed

As shown in the figure, 5*6 has been calculated and the result pushed on the stack. The quotient of 4 and 2 has been pushed on the stack. You can see the values still displayed in the operand and result textboxes.

Designing the Form: Stage 1

Using Figure 11-6 as a guide, set up a form for the Postfix Expression Evaluation project. Set the FontSize property of every label and box to 9.75. In this first stage, you'll build the part of the form involving the command buttons, the textbox used to enter the postfix expression, and a couple of labels.

1. Start Visual Basic. If Visual Basic is running, choose New Project from the File menu.
2. Rename the default form to **frmStackDemo**. Change the caption of the form to **Postfix Expression Evaluator**.
3. Create a textbox to enter the postfix expression. Delete the text. Change the name to **txtPostfix**.
4. Create a label below the box. Change the caption to **Enter a postfix expression**.
5. Create two command buttons. Change their captions to **&Step** and **&Reset**. Change their names to **cmdStep** and **cmdReset**.
6. Next to the command button, create two labels. These labels are made visible and invisible depending upon the operation, push or pop. Change the captions to **PUSH** and **POP**, and the names to **lblPush** and **lblPop**.

Designing the Form: Stage 2

In this stage, you set up the labels used to show the operands, the current operation, and the stack itself. The stack is always visible on the form. The contents of the stack are displayed in labels all bearing the same name. Labels with the same name form a control array.

To complete the second stage:

1. Assemble the part of the form that shows the operands and the operator between them. Create a label. Set the Alignment property to Center. Change the name to **lblOp1**. Delete the caption.
2. Select the label you just created. Copy it with Ctrl+C. Paste it with Ctrl+V. Respond No to creating a control array. Creating a label in this way preserves all the properties. Name the box **lblOp2**.
3. Between lblOp1 and lblOp2, insert a label with the name **lblOperator**. Delete the caption.
4. Again select and copy lblOp1. Paste it with Ctrl+V. Respond No to the control array. Change the name of the label to **lblChar**.
5. Place labels below the operand and character boxes. Change the captions to **Operand 1**, **Operand 2**, and **Character**.
6. Select and copy lblOp1. Paste it, respond No to the control array, and change the name to **lblResult**.
7. Label the result with a label with caption **Result**.
8. Create a new label for the top of the stack along the right side of the form. Change the name to **lblStack**. Change the BorderStyle to Single, Alignment to Center, and delete the caption.
9. Select the label just created. Copy and paste. Respond Yes to creating a control array. A control array is like a menu control array. Each of the labels has the same name. Each is accessed as if it is a member of an array. Each has a subscript appended to the name to distinguish it from the other elements of the control array. All elements of the control array share the same event procedures.
10. Paste (Ctrl+V) again and again until there are eight labels equally spaced along the right side of the form.
11. Create a TOS label as shown. Change the name of the label to **lblTOS**. Set the Alignment to Right Justify.
12. Save the form and project files.

Entering the Declarations

Declarations include the stack, the top-of-stack pointer, a pointer to the character being processed in the input string, and a variable, set in Form_Load used to determine the position of the moving *TOS* label.

Enter these lines in the general declarations section of the form:

```
Option Explicit
'—The stack
Dim Stack(0 To 7) As Integer
'—The top-of-stack pointer
Dim TOS As Integer
'—Pointer into the input string
Dim P1 As Integer
'—Number of twips to move and position the TOS label
Dim TOSTwips As Integer
```

Most postfix expressions don't require a stack with a lot of elements. This stack has a capacity of eight values. The values of the stack, the stack pointer, *TOS*, *p1*, and *TOSTwips* are available to every event procedure in the form.

Entering the Code

Now, enter the code:

1. Enter **End** in the procedure for cmdExit.
2. Enter the following lines in cmdReset_Click:

```
'—Declare local variable as loop index
Dim x As Integer
'—Reposition the top-of-stack label to the top
'of the stack, TOSTwips many twips from the top of the form
'_TOSTwips is initialized in the Form_Load procedure.
lblTOS.Top = TOSTwips
'—Clear the 8 visible stack entries
For x = 0 To 7
   lblStack(x) = ""
Next x
'—Clear several textboxes
lblChar = ""
lblOp1 = ""
lblOp2 = ""
lblResult = ""
txtPostfix = ""
TOS = 0
P1 = 1
```

The screen stack, comprised of the lblStack labels, is something new. Each of the labels is given the same name. Whenever you give controls the same name, Visual Basic creates a control array. This allows you to refer to each box with their common name and a number designating a particular box.

3 Enter the following lines in the Form_Load procedure:

```
lblpop.Visible = False
lblpush.Visible = False
'—Set to first character in input string
P1 = 1
'—Initialize top-of-stack variable
TOS = 0
'—Reposition the top-of-stack label to the top
'of the stack, TOSTwips twips from the top of the form
TOSTwips = ScaleHeight / 9
lblTOS.Top = TOSTwips
'—The loop that follows positions the 8 labels showing
'the contents of the stack evenly down the right edge of the form
Dim x As Integer
For x = 0 To 7
   lblStack(x).Top = (x + 1) * TOSTwips
Next x
```

4 Enter the **Pop** function in the general section of the form. Copy the top and bottom lines:

```
Function Pop () As Integer
   '—If the stack is not empty, decrement pointer,
   'copy value from array, and return as the function's value
   If TOS > 0 Then
      TOS = TOS - 1
      Pop = Stack(TOS)
   Else
      MsgBox "Stack underflow"
   End If
End Function
```

5 Enter the **Push** procedure in the general section of the form. Copy the top and bottom lines:

```
Sub Push (R As Integer)
   '—If the stack is not full, put value, R
   'into the top of the stack, and increment
   'the stack pointer
```

```
    If TOS < 20 Then
       Stack(TOS) = R
       TOS = TOS + 1
    Else
       MsgBox "Stack overflow"
    End If
End Sub
```

6 Enter the following code for cmdStep. This code automates the processing of the postfix expression. Start with local variable declarations. Declare a string for the postfix expression, a single-character string for the character being processed, and integer variables for the operands and the result of the operations.

```
'—String used to hold entered postfix expression
Dim Postfix As String     ' input string
'—Declare variables for the operands and the result
Dim Op1 As Integer, Op2 As Integer, Result As Integer
'—Declare variable for characters taken from input string
Dim Ch As String * 1
```

7 Enter the section of code that pulls a character out of the input string. *P1* points to the next character to be processed. It is checked against the length of the input string. If there are more characters left to process, the procedure continues.
The **Mid$***(string, starting char, number of chars)* function creates a new string that is a copy of a segment of an existing string. The source string, the starting point, and the number of characters to copy are passed as parameters.

```
'—Pick up expression from textbox
Postfix = txtPostfix
'—p1 is a pointer into the input string. It picks out
'the character to work on.
If P1 <= Len(Postfix) Then
   Ch = Mid$(Postfix, P1, 1)
   '—Assign the character to a label for display
   lblChar = Ch
EndIf
```

8 The direction the program takes from here depends on the value of the character being processed. If it is a digit "0" through "9", it is an operand and it is pushed on the stack. If it is an operator, (+ - * /), pop two values from the stack, store them in the variables *op1* and *op2*, apply the operator, and push the result.

The **Select Case** statement controls which branch of the program executes.

Reposition the *TOS* pointer by resetting its Top property. Its initial position is recorded in twips in the Top property of the control.

The **Select Case** statement branches based on a single value, an aggregate of values separated by commas, or a range of values. To specify a range of values, use the keyword **To** between the low end of the range and the high end of the range: "0" to "9" is the range of characters, 0 through 9.

Enter these lines.

```
Select Case Ch
   '—If the character is a digit, push on the stack
   Case "0" To "9"
      lblpush.Visible = True
      lblpop.Visible = False
      '—Display character in on-screen stack
      lblStack(TOS) = Ch
      '—Push the character on the internal stack
      Push (Val(Ch))
      '—Move the top-of-stack pointer TOSTwips twips down the form
      lblTOS.Top = lblTOS.Top + TOSTwips
      '—Advance to next character
      P1 = P1 + 1
      '—Skip spaces
   Case " "
      P1 = P1 + 1
      '—Operators, Pop, Pop, apply operator, Push result
   Case "+", "-", "*", "/", "\"
      lblpop.Visible = True     ' pop label on
      lblpush.Visible = False   ' push label off
      lblTOS.Top = lblTOS.Top - TOSTwips     ' move TOS label
      Op2 = Pop()     ' second operand
      lblStack(TOS) = ""     ' clear stack value
      lblOp2 = Op2     ' fill op2 box
      lblTOS.Top = lblTOS.Top - TOSTwips     ' move TOS label
      Op1 = Pop()     ' first operand
      lblStack(TOS) = ""     ' clear stack value
      lblOp1 = Op1     ' fill op1 box
      '—Apply operator
      Select Case Ch
         Case "+"
```

```
                Result = Op1 + Op2
                lbloperator.Caption = "+"
            Case "-"
                Result = Op1 - Op2
                lbloperator.Caption = "-"
            Case "*"
                Result = Op1 * Op2
                lbloperator.Caption = "*"
            Case "/", "\"
                Result = Op1 \ Op2
                lbloperator.Caption = "/"
        End Select
        '—Display result and push onto stack
        lblResult = Result
        lblStack(TOS) = Result
        Push (Result)
        '—Move TOS label
        lblTOS.Top = lblTOS.Top + TOSTwips
        '—Move pointer to next character
        P1 = P1 + 1
    End Select
    End Sub
```

9 Save the form and project files.

10 Run the program. Enter the following postfix expressions and step through the evaluation of each. Use only single digit numbers!

a) 5 6 7 8 9 + - + *
b) 5 6 + 7 - 8 + 9 *
c) 3 3 * 4 1 * 5 * -
d) 2 3 1 * 4 2 / 5 * - 3 2 + +

QUESTIONS AND ACTIVITIES

1. How deep a stack is needed to evaluate the following expression?
   0 3 - 3 3 * 4 2 * 5 * - + 2 2 * /

2. The labels in the Postfix Evaluation program form a control array. The values from a postfix expression are not only pushed onto a stack, they are also displayed in the control array of labels used to display the stack contents on the form. Could the program have been written substituting the control array of labels on the form for the stack of values used by **Push** and **Pop**?

3. Use the Help system to look up the entry for the keyword **Static**. How does **Static** work? How is it unlike **Dim**?
4. Assume the string *A$* has the value "George Washington". Write a program segment that would display the following in a textbox named txtStars:
   G*e*o*r*g*e* *W*a*s*h*i*n*g*t*o*n*

   Use the **Mid$()** function in a **For-Next** loop to separate individual letters of *A$*.
5. How would you specify a range of values from top to bottom in a **Select Case** statement? How would you specify the range of characters from "q" through "s"?

Summary

A stack is a list of items for which access is restricted to the top item of the list. You can create a stack by using an array and a variable that acts as a pointer to the top element of the stack. Stacks are accessed by two subprograms: The **Pop** function pops a value from the top of the stack, and the **Push** subroutine puts a value onto the top of the stack. Each procedure leaves the stack pointer pointing to the next open space of the stack.

Infix notation puts the operator between the operands (*a*+*b*). Postfix notation puts the operator after the operands (*a b* +). Postfix notation is particularly well suited for computers because its rule for order of operations is to process expressions in order left to right.

You can make labels, as well as several other screen controls, into control arrays by giving each the same name. The labels are accessed with the common name and a numerical index.

The Form_Load procedure is used for initializing variables and taking care of other housekeeping chores. This procedure executes whenever a form is loaded into memory.

The **Select Case** statement can use a range of values to choose a branch using the following syntax:

Case *low item* **To** *high item*

1. Rewriting the Stack Project

Rewrite the stack project to display the entire stack in a list box. Use Visual Basic Help to find out how to use the **RemoveItem** method to remove lines from the list box.

2. The Unary Minus Modification

Using the tilde, ~, as the unary minus symbol, rewrite the Postfix Evaluator program to process this operation. When the tilde is encountered, the top of the stack should be negated. The unary minus works like this: 5~ 2 + , would add negative 5 to 2.

3. The Stripped Down Evaluator

Write a program that enters a postfix expression and displays either the value of the expression, or an error message. Leave out all the screen display stuff in the original program.

4. Reversing Characters with a Stack

Write a program to enter a name using an InputBox. With the **Mid$** function, peel the individual characters off the name, and push the characters onto a character stack. When the last character has been processed, pop and display the characters in a textbox. The characters should be in reverse order.

5. Multiple Digit Values

Rewrite the Postfix Evaluator program to allow the user to use multiple digit integers separated by spaces. Your program should allow the user to enter the Postfix expression in a textbox, process that expression using a stack, and display the resulting value of the expression.

```
Sub Subtract (X
As Fraction, Y As
Fraction, Rslt As
Fraction)

Dim Temp As
Fraction

Negate Y,
Temp
Add A, Rslt
End Sub
```