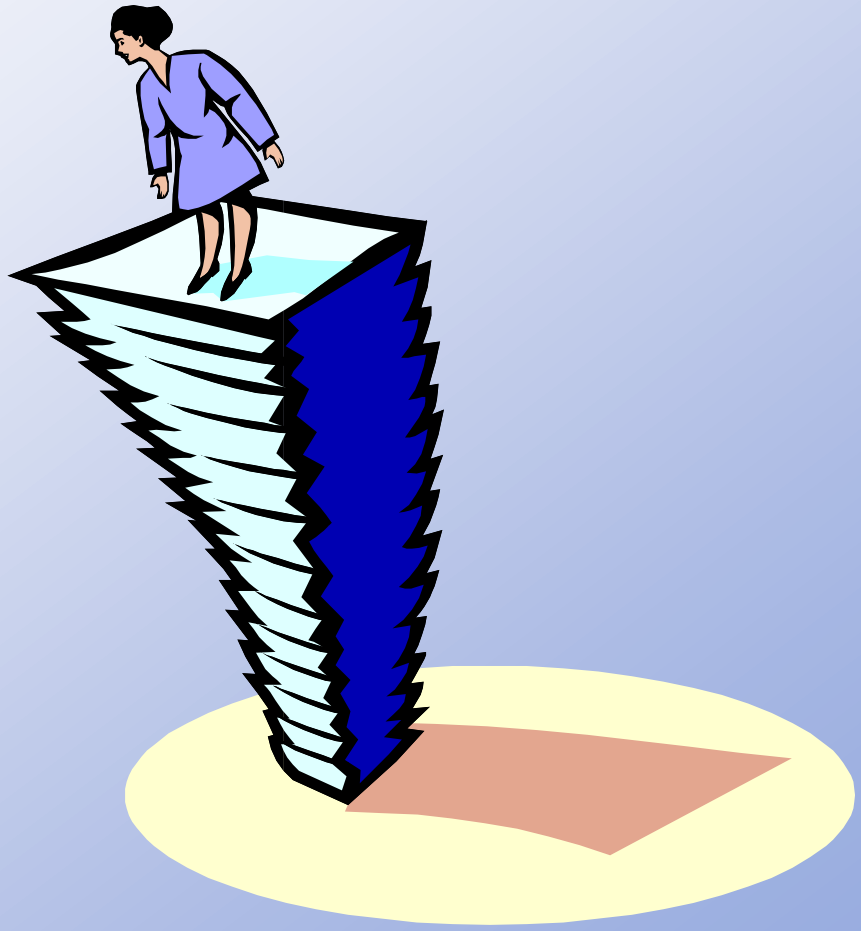
Stacks

Chapter 7



Chapter Contents

Chapter Objectives

- Study a stack as an ADT
- Build a static-array-based implementation of stacks
- Build a dynamic-array-based implementation of stacks
- Build a linked-implementation of stacks
- Show how a run-time stack is used to store information during function calls
- (Optional) Study postfix notation and see how stacks are used to convert expressions from infix to postfix and how to evaluate postfix expressions

Introduction to Stacks

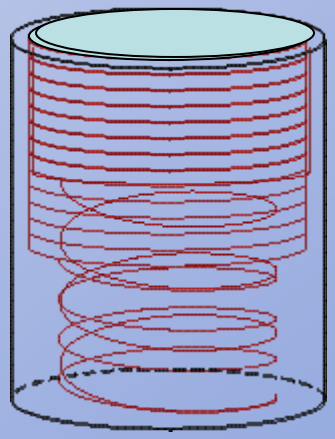
- Consider a card game with a discard pile
  - Discards always placed on the top of the pile
  - Players may retrieve a card only from the top

What other examples can you think of that are modeled by a stack?

- We seek a way to represent and manipulate this in a computer program
- This is a stack

Introduction to Stacks

- A stack is a last-in-first-out (LIFO) data structure
- Adding an item
  - Referred to as pushing it onto the stack
- Removing an item
  - Referred to as popping it from the stack

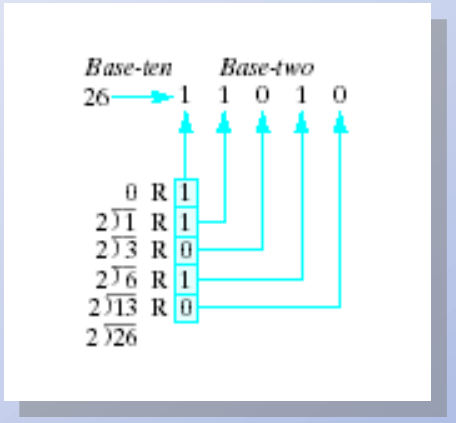


A Stack

- Definition:
  - An ordered collection of data items
  - Can be accessed at only one end (the top)
- Operations:
  - construct a stack (usually empty)
  - check if it is empty
  - Push: add an element to the top
  - Top: retrieve the top element
  - Pop: remove the top element

Example Program

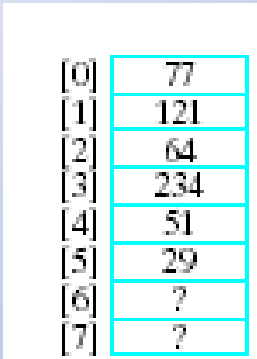
- Consider a program to do base conversion of a number (ten to two)



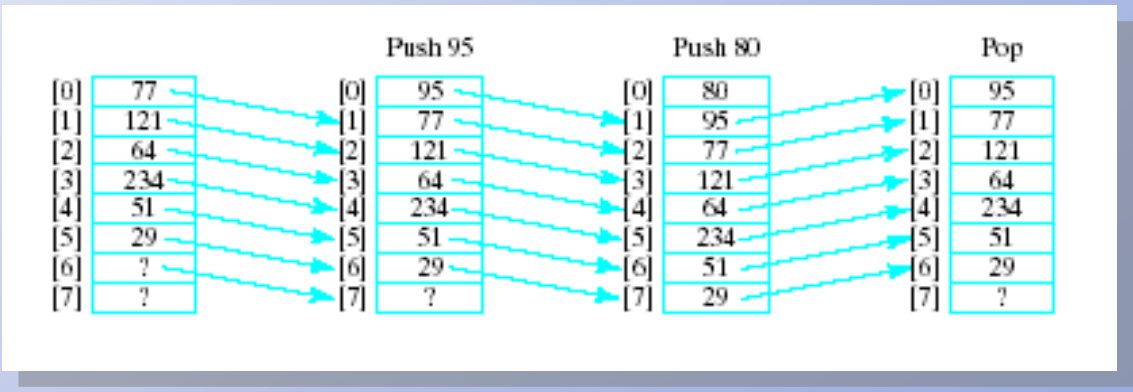
- Note program which assumes existence of a `Stack` class to accomplish this, Fig 7.2
  - Demonstrates `push`, `pop`, and `top`

Selecting Storage Structure

- Model with an array
  - Let position 0 be top of stack

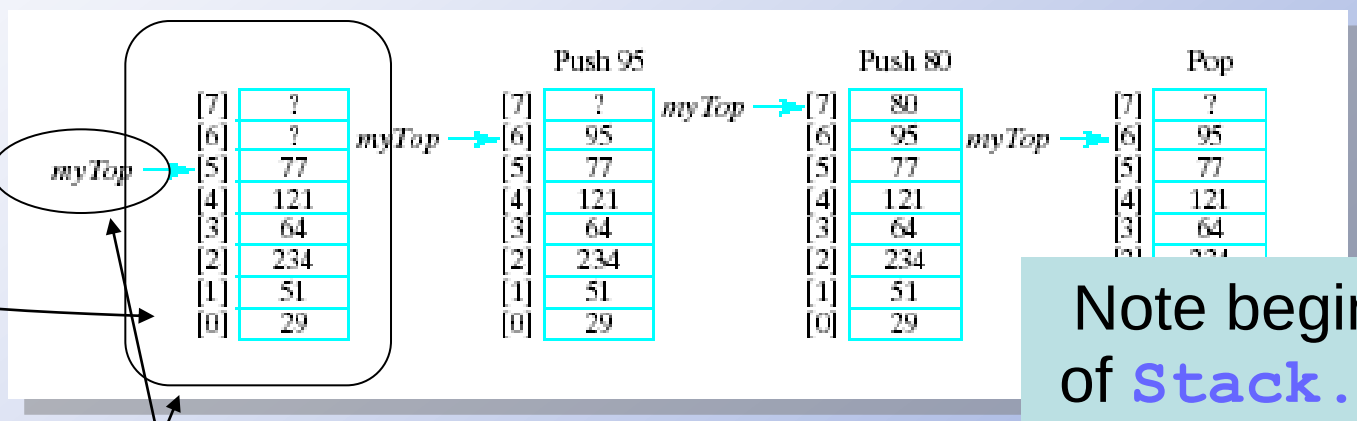


- Problem … consider pushing and popping
  - Requires much shifting



Selecting Storage Structure

- A better approach is to let position 0 be the bottom of the stack



Note beginning of `Stack.h` file, Fig. 7.3

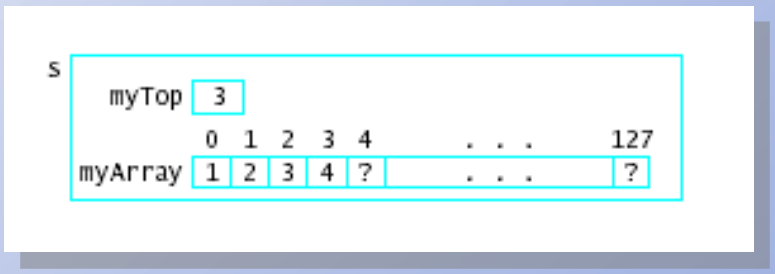
- Thus our design will include
  - An array to hold the stack elements
  - An integer to indicate the top of the stack

Implementing Operations

- Constructor
  - Compiler will handle allocation of memory
- Empty
  - Check if value of `myTop == -1`
- Push (if `myArray` not full)
  - Increment `myTop` by 1
  - Store value in `myArray [myTop]`
- Top
  - If stack not empty, return `myArray[myTop]`
- Pop
  - If array not empty, decrement `myTop`
- Output routine added for testing

The Stack Class

- The completed `Stack.h` file, Fig. 7.4A
  - All functions defined
  - Note use of `typedef` mechanism
- Implementation file, `Stack.cpp`, Fig 7.4B
- Driver program to test the class, Fig 7.5
  - Creates stack of 4 elements
  - Demonstrates error checking for stack full, empty



Dynamic Array to Store Stack Elements

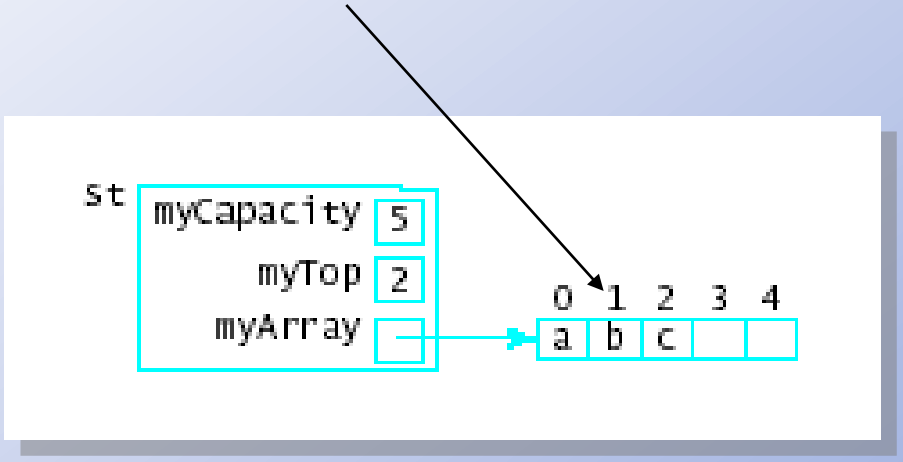
- Same issues regarding static arrays for stacks as for lists
  - Can run out of space if stack set too small
  - Can waste space if stack set too large
- As before, we demonstrate a dynamic array implementation to solve the problems
- Note additional data members required
  - DStack Data Members

Dynamic Array to Store Stack Elements

- Constructor must
  - Check that specified `numElements > 0`
  - Set capacity to `numElements`
  - Allocate an array pointed to by `myArray` with capacity = `myCapacity`
  - Set `myTop` to -1 if allocation goes OK
- Note implementation of constructor for DStack
- Fig 7.6A DStack.h, Fig. 7.6B DStack.cpp

Dynamic Array to Store Stack Elements

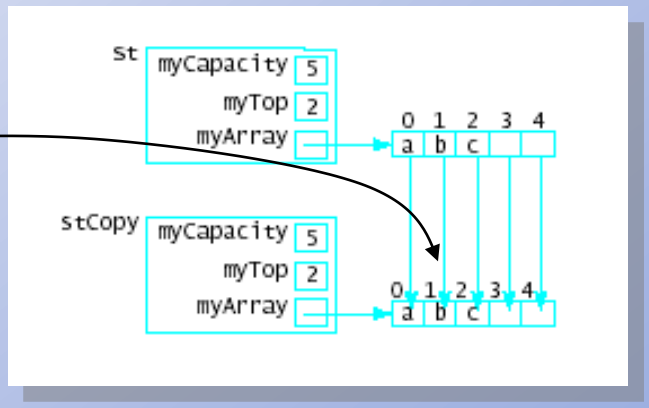
- Class Destructor needed
  - Avoids memory leak
  - Deallocates array allocated by constructor



- Note destructor definition

Dynamic Array to Store Stack Elements

- Copy Constructor needed for
  - Initializations
  - Passing value parameter
  - Returning a function value
  - Creating a temporary storage value
- Provides for deep copy
- Note definition



Dynamic Array to Store Stack Elements

- Assignment operator
  - Again, deep copy needed
  - copies member-by-member, not just address
- Note implementation of algorithm in `operator=` definition

- View driver program to test DStack class, Fig. 7.10

Further Considerations

- What if dynamic array initially allocated for stack is too small?
  - Terminate execution?

  - Replace with larger array!

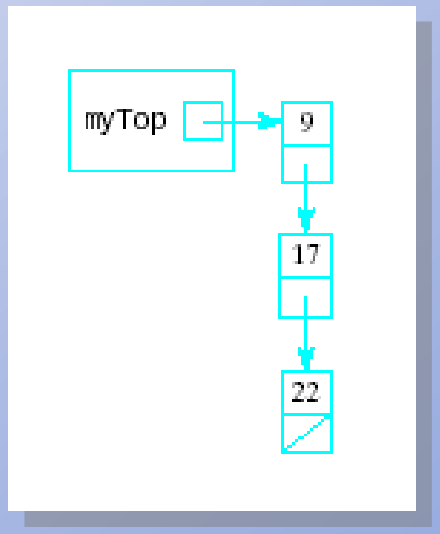
- Creating a larger array
  - Allocate larger array
  - Use loop to copy elements into new array
  - Delete old array
  - Point `myArray` variable at this new array

Further Considerations

- Another weakness – the type must be set with `typedef` mechanism
- This means we can only have one type of stack in a program
  - Would require completely different stack declarations and implementations
- Solution coming in Chapter 9
  - class templates

Linked Stacks

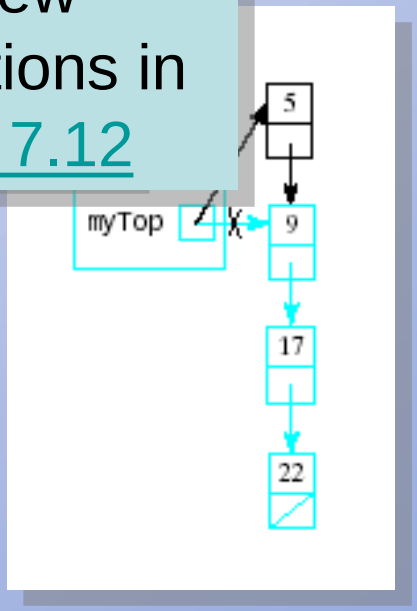
- Another alternative to allowing stacks to grow as needed
- Linked list stack needs only one data member
  - Pointer **`myTop`**
  - Nodes allocated (but not part of stack class)
- Note declaration, Fig. 7-11



Implementing Linked Stack Operations

- Constructor
  - Simply assign null pointer to `myTop`
- Empty
  - Check for `myTop == n`
- Push
  - Insertion at beginning of list
- Top
  - Return data to which `myTop` points

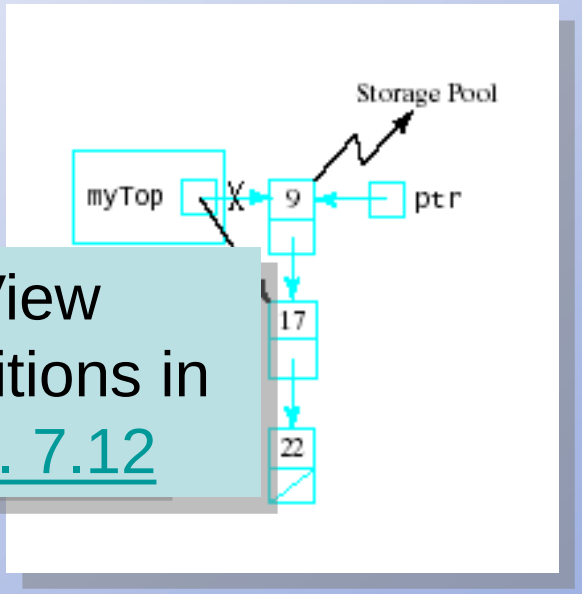
View definitions in Fig. 7.12



Implementing Linked Stack Operations

- Pop
  - Delete first node in the linked list

```
ptr = myTop;
myTop = myTop->next;
delete ptr;
```



View definitions in Fig. 7.12

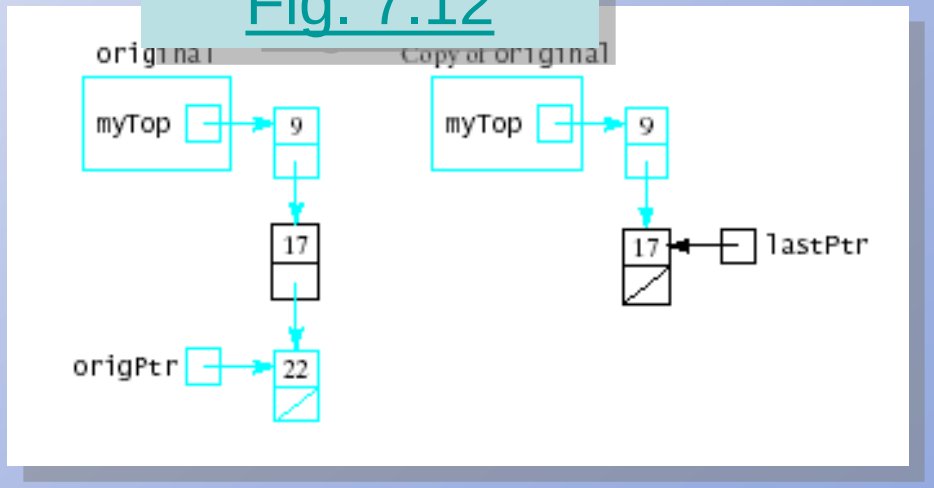
- Output
  - Traverse the list

```
for (ptr = myTop;
        ptr != 0; ptr = ptr->next)
   out << ptr->data << endl;
```

Implementing Linked Stack Operations

- Destructor
  - Must traverse list and deallocate nodes
  - Note need to keep track of `ptr->next` before calling `delete ptr;`
- Copy Constructor
  - Traverse linked list, copying each into new node
  - Attach new node to copy

View definitions in Fig. 7.12



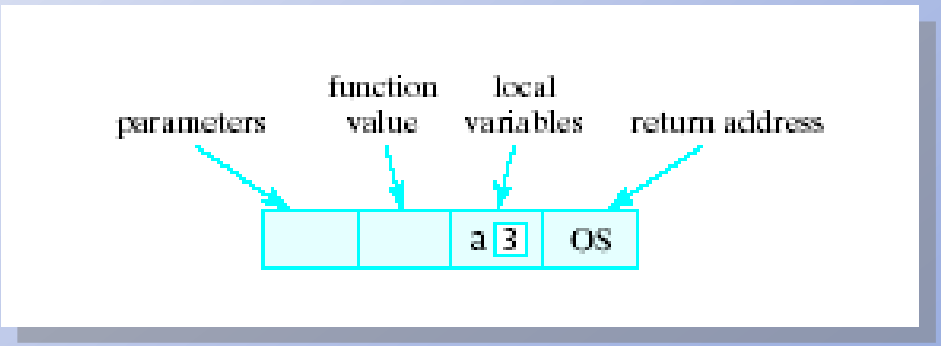
Implementing Linked Stack Operations

- Assignment operator
  - Similar to copy constructor
  - Must first rule out self assignment
  - Must destroy list in stack being assigned a new value
- View completed linked list version of stack class, Fig 7.12
- Note driver program, Fig. 7.12C

Application of Stacks

Consider events when a function begins execution

- *Activation record* (or *stack frame*) is created
- Stores the *current environment* for that function.
- Contents:



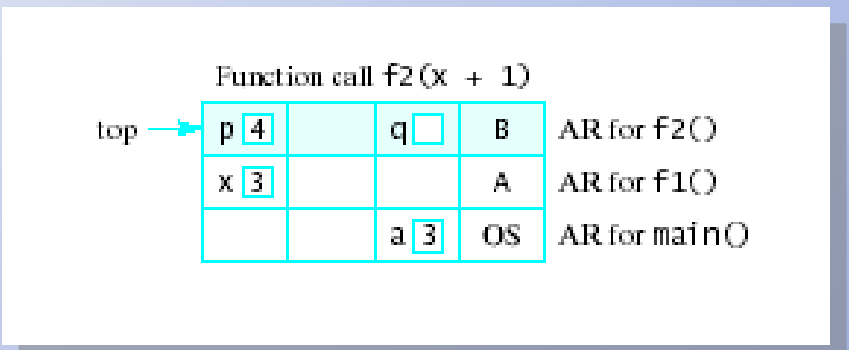
Run-time Stack

- Functions may call other functions
  - interrupt their own execution
- Must store the activation records to be recovered
  - system then reset when first function resumes execution
- This algorithm must have LIFO behavior
- Structure used is the run-time stack

Use of Run-time Stack

When a function is called …

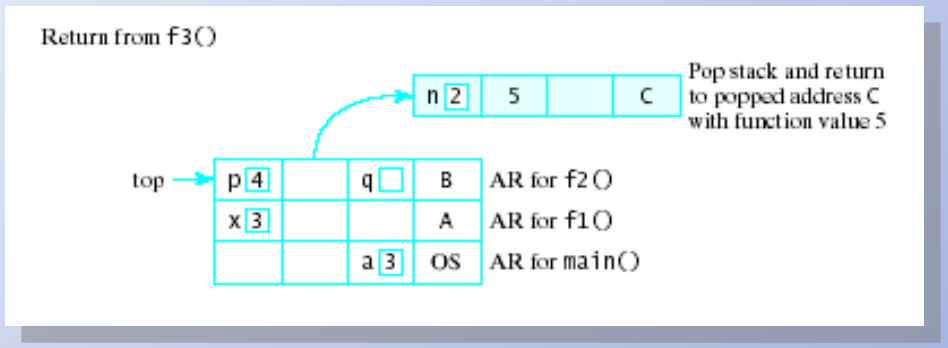
- Copy of activation record pushed onto run-time stack
- Arguments copied into parameter spaces
- Control transferred to starting address of body of function



Use of Run-time Stack

When function terminates

- Run-time stack popped
  - Removes activation record of terminated function
  - exposes activation record of previously executing function



- Activation record used to restore environment of interrupted function
- Interrupted function resumes execution

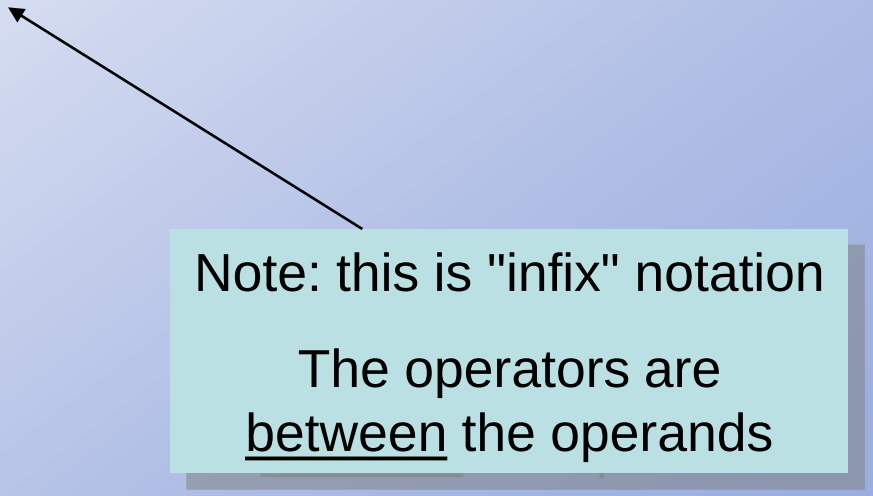
Application of Stacks

Consider the arithmetic statement in the assignment statement:

x = a * b + c

Compiler must generate machine instructions

1. LOAD a
2. MULT b
3. ADD c
4. STORE x



RPN or Postfix Notation

- Most compilers convert an expression in ***infix*** notation to ***postfix***
  - the operators are written after the operands
- So a * b + c becomes a b * c +
- Advantage:
  - expressions can be written without parentheses

Postfix and Prefix Examples

INFIX	RPN (POSTFIX)	PREFIX
A + B	A B +	+ A B
A * B + C	A B * C +	+ * A B C
A * (B + C)	A B C + *	* A + B C
A - (B - (C - D))	A B C D---	-A-B-C D
A - B - C - D	A B-C-D-	---A B C D

Prefix : Operators come before the operands

Evaluating RPN Expressions

"By hand" (Underlining technique):

1. Scan the expression from left to right to find an operator.
2. Locate ("underline") the last two preceding operands and combine them using this operator.
3. Repeat until the end of the expression is reached.

Example:

```
      2 3 4 + 5 6 - - *
  →   2 3 4 + 5 6 - - *
  →   2 7 5 6 - - *
  →   2 7 5 6 - - *
  →   2 7 -1 - *
  →   2 7 -1 - *   →  2 8 * → 2 8 * → 16
```

Evaluating RPN Expressions

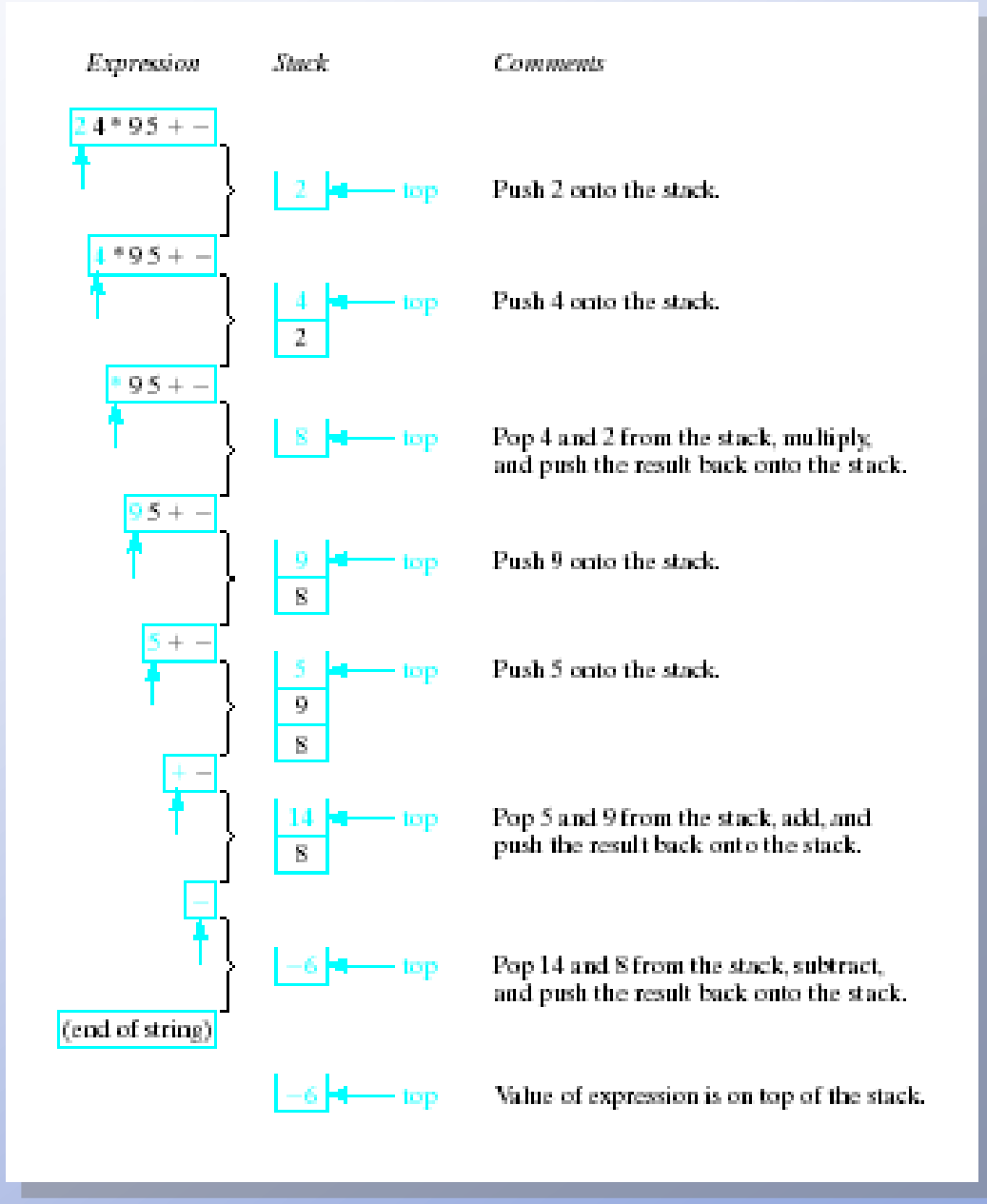
By using a stack algorithm

1. Initialize an empty stack
2. Repeat the following until the end of the expression is encountered
   a) Get the next token (const, var, operator) in the expression
   b) Operand – push onto stack
      Operator – do the following
      i. Pop 2 values from stack
      ii. Apply operator to the two values
      iii. Push resulting value back onto stack
3. When end of expression encountered, value of expression is the (only) number left in stack

Note: if only 1 value on stack, this is an invalid RPN expression

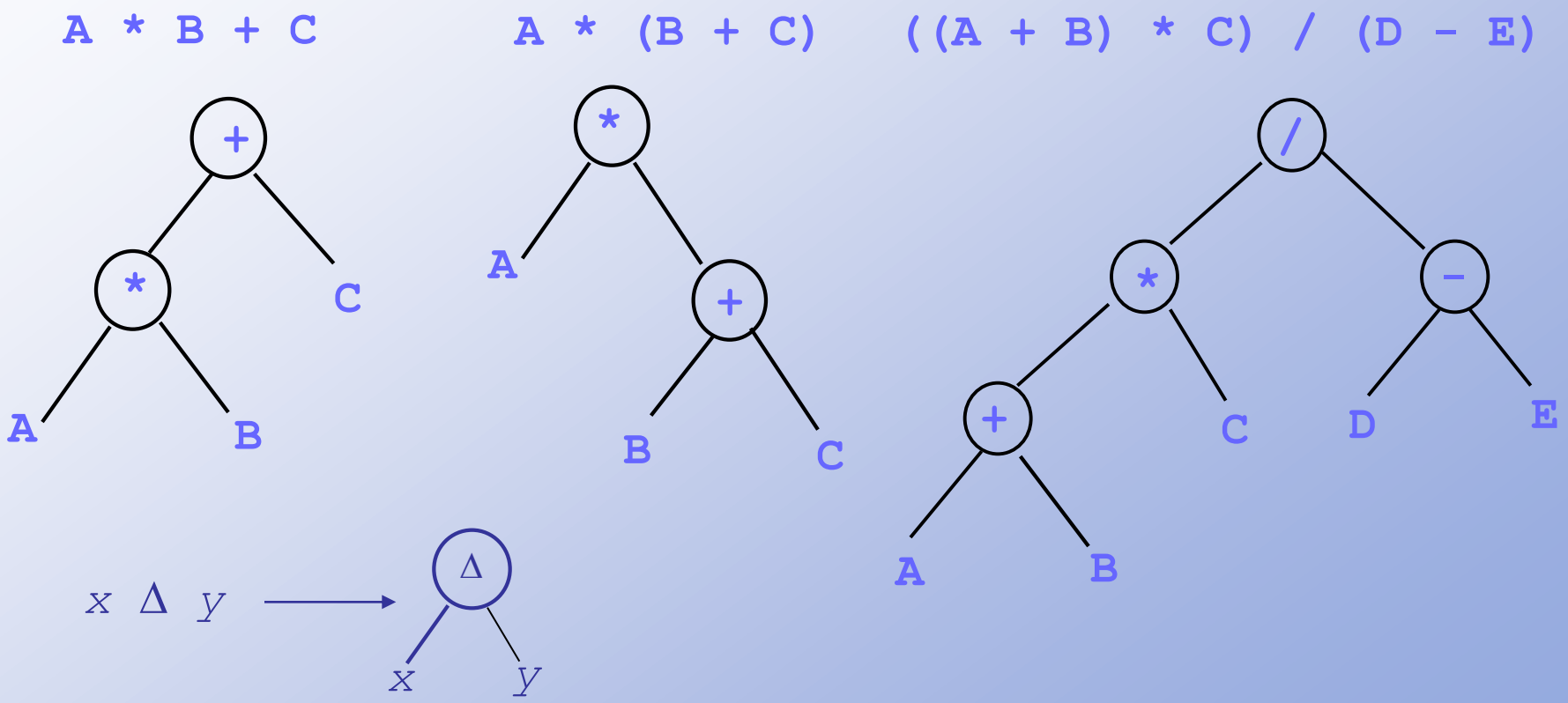
Evaluation of Postfix

- Note the changing status of the stack



Converting Infix to RPN

By hand: Represent infix expression as an *expression tree*:



Traverse the tree in *Left-Right-Parent* order (*postorder*) to get RPN:

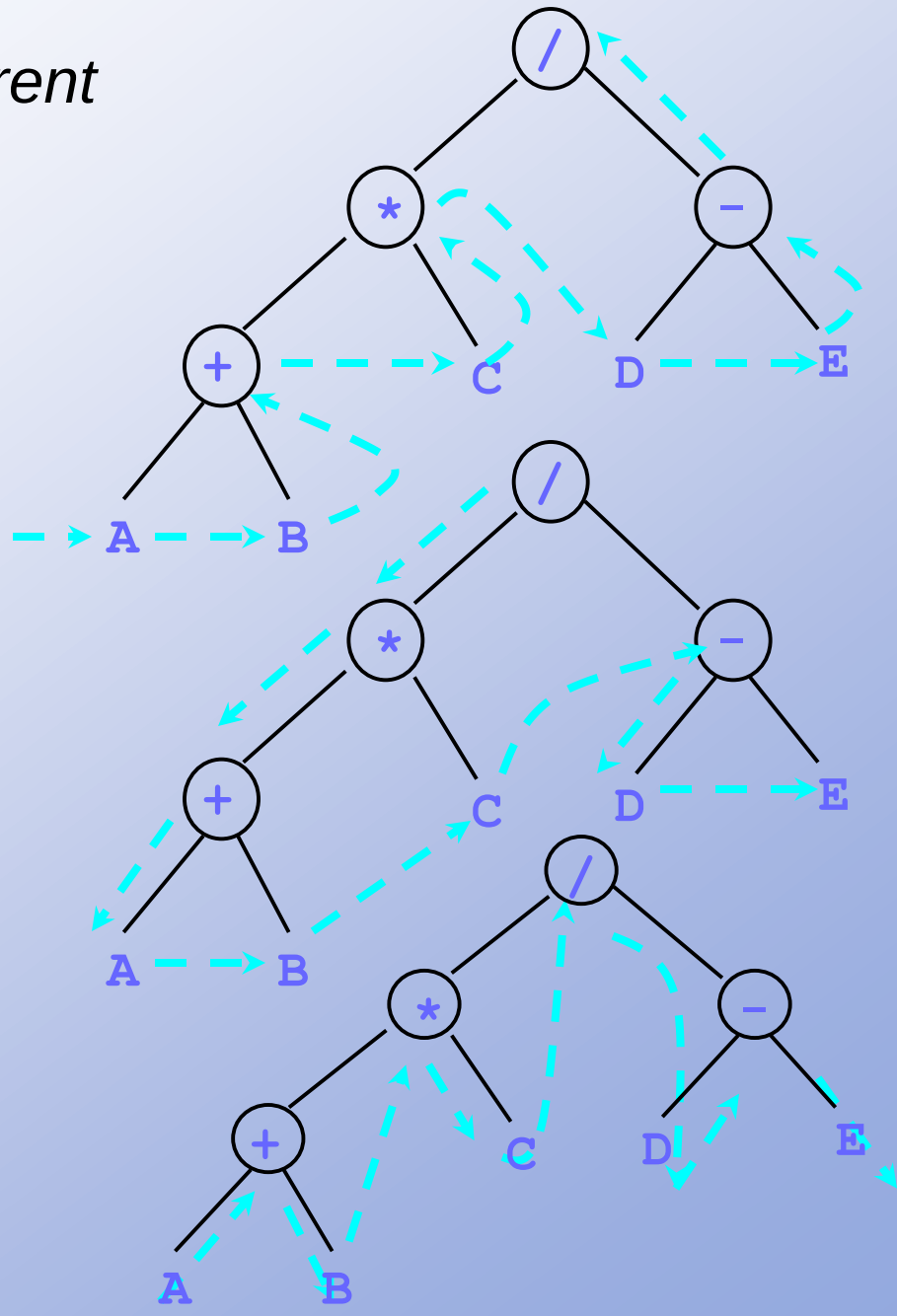
```
A B + C * D E - /
```

Traverse tree in *Parent-Left-Right* order (*preorder*) to get **prefix:**

```
/ * + A B C - D E
```

Traverse tree in *Left-Parent-Right* order (*inorder*) to get **infix:**
— must insert ()'s

```
( ((A + B) * C) / (D - E) )
```



Another RPN Conversion Method

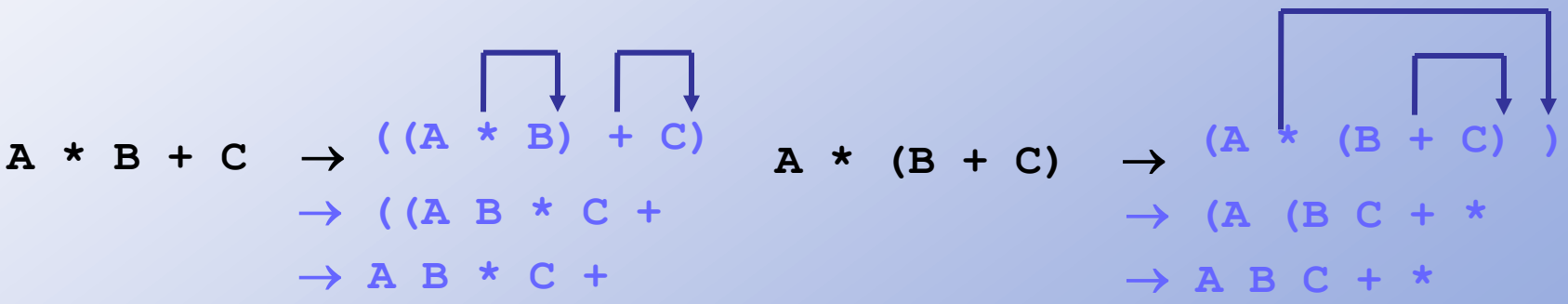
By hand: "Fully parenthesize-move-erase" method:

1. Fully parenthesize the expression.

2. Replace each right parenthesis by the corresponding operator.

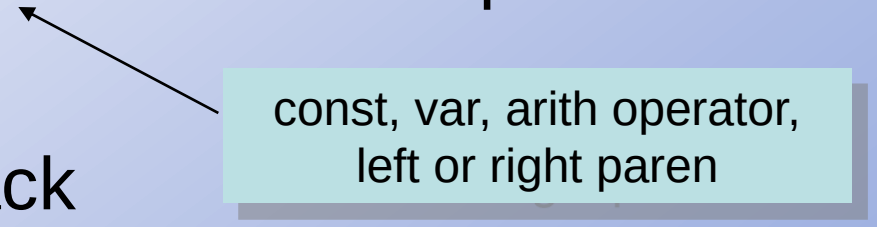
3. Erase all left parentheses.

Examples:



Stack Algorithm

1. Initialize an empty stack of operators
2. While no error && !end of expression
   a) Get next input "token" from infix expression
   b) If token is …
      i. "(" : push onto stack
      ii. ")" : pop and display stack elements until "(" occurs, do not display it



Stack Algorithm

Note: Left parenthesis in stack has lower priority than operators

iii. operator
if operator has higher priority than top of stack
    push token onto stack
else
    pop and display top of stack
    repeat comparison of token with top of stack

iv. operand display it

3. When end of infix reached, pop and display stack items until empty

Sample Program

- Converts infix expression to postfix
  - Uses Stack data type (dynamically allocated version)
  - View Fig 7.15
  - User enters elements of infix expressions separated by spaces
  - Program generates postfix expression
  - Also notes invalid infix expressions