COEN 79
Object-Oriented Programming and Advanced Data Structures
Department of Computer Engineering, Santa Clara University
Behnam Dezfouli, PhD

Stacks

Slide Set: 7

Learning Objectives

- Follow and explain stack based algorithms using the usual computer science terminology of push, pop, top, and peek
- Implement a stack class of your own using either an array or a linked list data structure
- Use the STL stack class to implement stack based algorithms such as the evaluation of arithmetic expressions

Introduction

- Entries in a stack are ordered: There is one that can be accessed first (the one on top), one that can be accessed second (just below the top), …
- **We do not require that the entries can be compared using the < operator**
- Stack entries must be removed in the reverse order
- Because of this property, a stack is called a **Last-In/First-Out** data structure (LIFO)
- Adding an entry to a stack is called a **push** operation, and removing an entry from a stack is called a **pop** operation

Implementations of The Stack Class

Array Implementation of a Stack

- Our stack template class definition uses two private member variables:
  - A partially-filled array, called `data`, that can hold up to `CAPACITY` items
  - A single member variable, `used`, that indicates how much of the partially-filled array is currently being used
    - `data[0]` is at "the bottom" of the stack
    - `data[used-1]` is at "the top" of the stack
    - If the value of `used` is zero, this will indicate an empty stack
- **Invariant of the `stack` Class** (Array Version)
  - The number of items in the stack is stored in the member variable `used`
  - The items in the stack are stored in a partially filled array called `data`, with the bottom of the stack at `data[0]`, the next entry at `data[1]`, and so on to the top of the stack at `data[used-1]`

Implementations of The Stack Class

Header File for the Array Version of the Stack Template Class

```
// FILE: stack1.h (part of the namespace scu_coen79_7A)
// TEMPLATE CLASS PROVIDED: stack<Item>
//
// TEMPLATE PARAMETER, TYPEDEFS and MEMBER CONSTANTS for the
// stack<Item> class:
// The template parameter, Item, is the data type of the items in the
// stack, also defined as stack<Item>::value_type. It may be any of
// the C++ built-in types (int, char, etc.), or a class with a default
// constructor, a copy constructor, and an assignment operator. The
// definition stack<Item>::size_type is the data type of any variable
// that keeps track of how many items are in a stack. The static const
// CAPACITY is the maximum capacity of a stack for this first stack
// implementation.
//
// NOTE: Many compilers require the use of the new keyword typename
// before using the expressions stack<Item>::value_type and
// stack<Item>::size_type. Otherwise the compiler doesn't have enough
// information to realize that it is the name of a data type.
```

Implementations of The Stack Class

Header File for the Array Version of the Stack Template Class (Cont'd)

```
//  CONSTRUCTOR for the stack<Item> template class:
//  stack( )
//  Postcondition: The stack has been initialized as an empty stack.

// MODIFICATION MEMBER FUNCTIONS for the stack<Item> class:
// void push(const Item& entry)
// Precondition: size( ) < CAPACITY.
// Postcondition: A new copy of entry has been pushed onto the stack.
//
// void pop( )
// Precondition: size( ) > 0.
// Postcondition: The top item of the stack has been removed.
//
// CONSTANT MEMBER FUNCTIONS for the stack<Item> class:
// bool empty( ) const
// Postcondition: Return value is true if the stack is empty.
//
// size_type size( ) const
// Postcondition: Return value is the total number of items in the
// stack.
```

Implementations of The Stack Class

Header File for the Array Version of the Stack Template Class (Cont'd)

```
//  Item top( ) const
//  Precondition: size( ) > 0.
//  Postcondition: The return value is the top item of the stack (but
//  the stack is unchanged. This differs slightly from the STL stack
//  (where the top function returns a reference to the item on top of
//  the stack).
//
//  VALUE SEMANTICS for the stack<Item> class:
//  Assignments and the copy constructor may be used with stack<Item>
//  objects.
```

Implementations of The Stack Class

Header File for the Array Version of the Stack Template Class (Cont'd)

```
#ifndef SCU_COEN79_STACK1_H
#define SCU_COEN79_STACK1_H
#include <cstdlib>   // Provides size_t

namespace scu_coen79_7A
{
    template <class Item>
    class stack
    {

    public:
        typedef std::size_t size_type;
        typedef Item value_type;
        static const size_type CAPACITY = 30;

        // CONSTRUCTOR
        stack( ) { used = 0; }

        // MODIFICATION MEMBER FUNCTIONS
        void push(const Item& entry);
        void pop( );
```

Implementations of The Stack Class

Header File for the Array Version of the Stack Template Class (Cont'd)

```
        // CONSTANT MEMBER FUNCTIONS
        bool empty( ) const { return (used == 0); }
        size_type size( ) const { return used; }
        Item top( ) const;


    private:
        Item data[CAPACITY];    // Partially filled array
        size_type used;         // How much of array is being used
    };
}

#include "stack1.template"      // Include the implementation.
#endif
```

Implementations of The Stack Class

Implementation of the Array Version of the Stack Template Class

```
// FILE: stack1.template

// INVARIANT for the stack class:
// 1. The number of items in the stack is in the member variable used.
// 2. The actual items of the stack are stored in a partially-filled
// array data[0]..data[used-1]. The stack elements appear from the
// bottom (at data[0]) to the top (at data[used-1]).


#include <cassert>  // Provides assert

namespace scu_coen79_7A
{
    template <class Item>
    const typename stack<Item>::size_type  stack<Item>::CAPACITY;
```

Implementations of The Stack Class

Implementation of the Array Version of the Stack Template Class

```
template <class Item>
void stack<Item>::push(const Item& entry)
// Library facilities used: cassert
{
    assert(size( ) < CAPACITY);
    data[used] = entry;
    ++used;
}
```

```
template <class Item>
void stack<Item>::pop( )
// Library facilities used: cassert
{
    assert(!empty( ));
    --used;
}
```

Implementations of The Stack Class

Implementation of the Array Version of the Stack Template Class

```
    template <class Item>
    Item stack<Item>::top( ) const
    // Library facilities used: cassert
    {
        assert(!empty( ));
        return data[used-1];
    }
}
```

Implementations of The Stack Class

Linked-List Implementation of a Stack

- To implement a stack as a dynamic structure
- Size can grow and shrink during execution
- **The head of the linked list serves as the top of the stack**
- Invariant of the Stack Class (Linked-List Version):
  - The items in the stack are stored in a linked list, with the top of the stack stored at the head node, down to the bottom of the stack at the tail node
  - The member variable `top_ptr` is the head pointer of the linked list of items

Implementations of The Stack Class

Header File for the Linked-List Version of the Stack Template Class

```
#ifndef SCU_COEN79_STACK2_H
#define SCU_COEN79_STACK2_H

#include <cstdlib> // Provides NULL and size_t
#include "node2.h" // Node template class

namespace scu_coen79_7B
{
    template <class Item>
    class stack
    {
    public:
        // TYPEDEFS
        typedef std::size_t size_type;
        typedef Item value_type;
```

Implementations of The Stack Class

Header File for the Linked-List Version of the Stack Template Class

```
// CONSTRUCTORS and DESTRUCTOR
stack( ) { top_ptr = NULL; }
stack(const stack& source);
~stack( ) { list_clear(top_ptr); }


// MODIFICATION MEMBER FUNCTIONS
void push(const Item& entry);
void pop( );
void operator =(const stack& source);
```

Implementations of The Stack Class

Header File for the Linked-List Version of the Stack Template Class

```
        // CONSTANT MEMBER FUNCTIONS
        size_type size( ) const
            { return scu_coen79_6B::list_length(top_ptr); }


        bool empty( ) const { return (top_ptr == NULL); }
        Item top( ) const;


    private:
        scu_coen79_6B::node<Item> *top_ptr;  // Points to top of stack
    };
}


#include "stack2.template" // Include the implementation
#endif
```

Implementations of The Stack Class

Implementing the Linked-List Version of the Stack Template Class

```
// TEMPLATE CLASS IMPLEMENTED: stack<Item>
#include <cassert>          // Provides assert
#include "node2.h"          // Node template class

namespace scu_coen79_7B
{
    template <class Item>
    stack<Item>::stack(const stack<Item>& source)
    // Library facilities used: node2.h
    {
        // Needed for argument of list_copy
        scu_coen79_6B::node<Item> *tail_ptr;

        list_copy(source.top_ptr, top_ptr, tail_ptr);
    }
```

Note that we must not have any `using` directive

- We do not need to write `scu_coen79_6B::list_copy`
- Compiler can tell which `list_copy` function is intended since some of its argument types are defined in the same `scu_coen79_6B`

Argument-Dependent Lookup (ADL) (a.k.a., Koening lookup): The use of arguments to determine which function to use

Implementations of The Stack Class

Implementing the Linked-List Version of the Stack Template Class

```
template <class Item>
void stack<Item>::push(const Item& entry)
// Library facilities used: node2.h
{
    list_head_insert(top_ptr, entry);
}
```

```
template <class Item>
void stack<Item>::pop( )
// Library facilities used: cassert, node2.h
{
    assert(!empty( ));
    list_head_remove(top_ptr);
}
```

Implementations of The Stack Class

Implementing the Linked-List Version of the Stack Template Class

```
template <class Item>
void stack<Item>::operator =(const stack<Item>& source)
// Library facilities used: node2.h
{
    scu_coen79_6B::node<Item> *tail_ptr;

    if (this == &source) // Handle self-assignment
        return;

    list_clear(top_ptr);
    list_copy(source.top_ptr, top_ptr, tail_ptr);
}

template <class Item>
Item stack<Item>::top( ) const
// Library facilities used: cassert
{
    assert(!empty( ));
    return top_ptr->data( );
}
}
```

Implementations of The Stack Class

The Koenig Lookup

- Some functions in the linked-list implementation require a local node variable, such as this:

```
scu_coen79_6B::node<Item> *tail_ptr;
```

- Since we are inside a template class, we must not have any `using` directives, and therefore we have the full name `scu_coen79_6B::node<Item>`
- When we use a node function such as `list_copy`, we do not always need to write the full name `scu_coen79_6B::list_copy`, because compilers can sometimes tell which `list_copy` function is intended since some of its arguments' types are defined in the same `scu_coen79_6B` namespace
- Use of arguments to determine which function to use is called the **Koenig lookup**
- Some compilers do not allow the Koenig lookup within template functions

STL Stack Class

The STL Stack Class

The Standard Library Stack Class

- The C++ Standard Template Library (STL) has a stack class
- Stack is specified as a template class
- The most important member functions are:
  - **`push:`** to add an entry at the top of the stack
  - **`pop:`** to remove the top entry
  - **`top:`** to get the item at the top of the stack without removing it
- There are no functions that allow a program to access entries other than the top entry
- **Stack underflow:** If a program attempts to pop an item off an empty stack
  - To help you avoid a stack underflow, the class provides a member function to test whether a stack is empty
- **Stack overflow**: If a program attempts to push an item onto a full stack

The STL Stack Class

The Standard Library Stack Class (Cont'd)

```
template < class T, class Container = deque<T> >
class stack;
```

- **stacks** are implemented as *containers adaptors*
- **Containers adaptors** are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements
- The container shall support the following operations:
  - `empty`
  - `size`
  - `back`
  - `push_back`
  - `pop_back`
- The standard container classes **vector**, **deque** and **list** fulfill these requirements

The STL Stack Class

The Standard Library Stack Class (Cont'd)

```
#include <iostream>         // std::cout
#include <stack>            // std::stack

int main ()
{
    std::stack<int> mystack;

    mystack.push(1);
    mystack.push(2);

   mystack.top() += 10;

   std::cout << "mystack.top() is now " << mystack.top() << '\n';

   return 0;
}
```

```
Output:
mystack.top() is now 12
```

Stack Applications

Stack Applications

- Most compilers use stacks to analyze the syntax of a program
- Stacks are used to keep track of local variables when a program is run
- Stacks can be used to search a maze or a family tree or other types of branching structures

Programming Example: Balanced Parentheses

- **Assume function `is_balanced()` checks expressions to see if the parentheses match correctly**

❑ Example 1: Consider the string "((X + Y∗(Z + 7)) ∗ (A + B))"
- Each of the left parentheses has a corresponding right parenthesis
- As the string is read from left to right, there is never an occurrence of a right parenthesis that cannot be matched with a corresponding left parenthesis
- Function call `is_balanced("((X + Y∗(Z + 7))∗(A + B))")` returns true

❑ Example 2: `is_balanced("((X + Y∗(Z + 7)∗(A + B))")` returns false

Programming Example: Balanced Parentheses (Cont'd)

- Scans the characters of the string from left to right
- Every time the function sees a **left parenthesis**, it is **pushed** onto the stack
- Every time the program reads a **right parenthesis**, the program **pops** a matching left parenthesis off the stack
  - **Parentheses match:** If stack is empty at the end of the expression
  - **Parentheses do not match:**
    - If the stack is empty when the algorithm needs to pop a symbol, or
    - If symbols are still in the stack after all the input has been read

Programming Example: Balanced Parentheses (Cont'd)

```
// FILE: parens.cxx
// A small demonstration program for a stack.
#include <cstdlib>     // Provides EXIT_SUCCESS
#include <iostream>    // Provides cin, cout
#include <stack>       // Provides stack
#include <string>      // Provides string

using namespace std;


// PROTOTYPE for a function used by this demonstration program
bool is_balanced(const string& expression);
// Postcondition: A true return value indicates that the parentheses
// in the given expression are balanced. Otherwise, the return value
// is false.
// Library facilities used: stack, string (and using namespace std)
```

Programming Example: Balanced Parentheses (Cont'd)

```
bool is_balanced(const string& expression){
  const char LEFT_PARENTHESIS = '(';
  const char RIGHT_PARENTHESIS = ')';

  stack<char> store;    // Stack to store the left parentheses as they occur
  string::size_type i; // An index into the string
  char next;           // The next character from the string
  bool failed = false; // Becomes true if a needed parenthesis is not found

  for (i = 0; !failed  &&  (i < expression.length( )); ++i)
   {
     next = expression[i];
     if (next == LEFT_PARENTHESIS)
        store.push(next);
     else if ((next == RIGHT_PARENTHESIS) && (!store.empty()))
       store.pop( ); // Pops the corresponding left parenthesis.
     else if ((next == RIGHT_PARENTHESIS) && (store.empty( )))
       failed = true;
   }
   return (store.empty( ) && !failed);
}
```

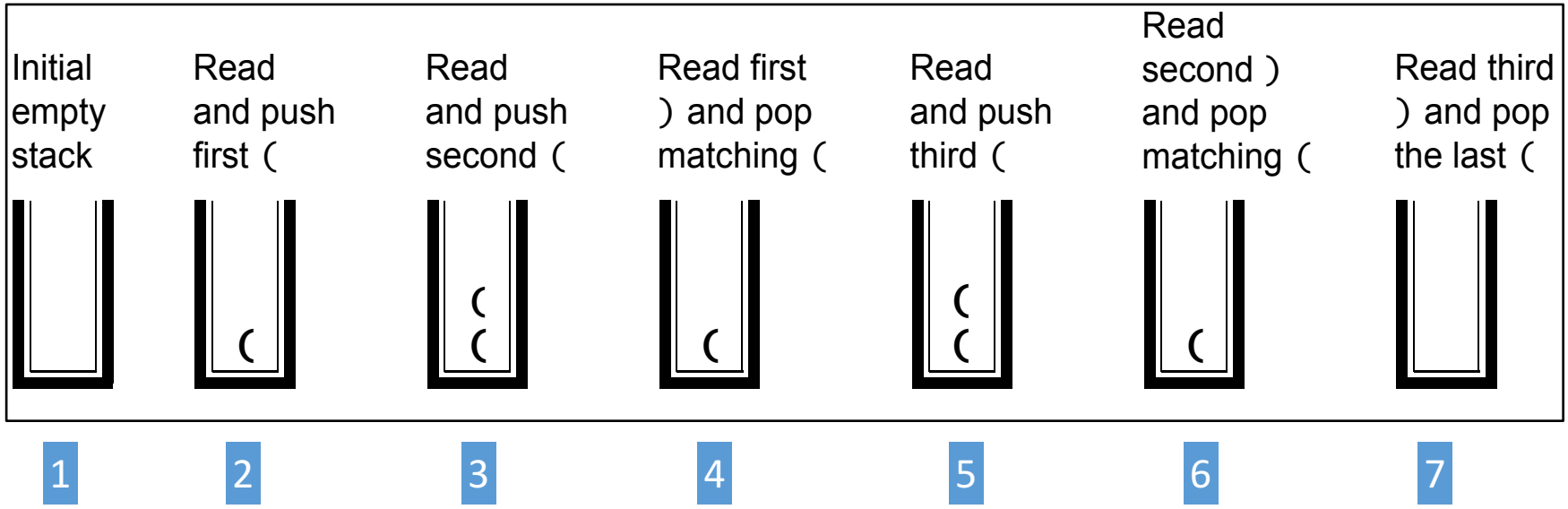
Programming Example: Balanced Parentheses (Cont'd)

```
int main( )
{
    string user_input;

    cout << "Type a string with some parentheses:\n";
    getline(cin, user_input);

    if ( is_balanced(user_input) )
        cout << "Those parentheses are balanced.\n";
    else
        cout << "Those parentheses are not balanced.\n";

    cout << "That ends this balancing act.\n";

    return EXIT_SUCCESS;
}
```

Programming Example: Balanced Parentheses (Cont'd)

- Example: Input (( ) ( ))

1 2 3 4 5 6 7



Programming Example: Evaluating Arithmetic Expressions

- We will design and write **a calculator program**
- Uses two stacks: **a stack of characters**, and **a stack of double numbers**
- **Specification:**
  - The program takes as input **a fully parenthesized** expression:

    `((((12 + 9)/3) + 7.2)*((6 - 4)/8))`
  - The expression consists of integers or double numbers, together with the operators +, −, ∗, and `/`, to focus on the use of the stack (rather than on input details)
  - We assume that:
    - The expression is formed correctly so that each operation has two arguments
    - The expression is fully parenthesized
    - Each input number is non-negative
  - The output will simply be the value of the arithmetic expression

Programming Example: Evaluating Arithmetic Expressions (Cont'd)

- **Design:** Most of the program's work will be carried out by a function that reads one line of input and evaluates that line as an arithmetic expression

❑ Example: Consider expression `(((6 + 9)/3)*(6 – 4))`

1. Evaluate the innermost expressions `(( 6 + 9 ) / 3) * (6 – 4))` to produce the simpler expression `( 15 /3 ) * 2 )`
2. Evaluate the expression (15 / 3) and replace this expression with its value of 5 to produce expression (5 * 2)
3. Evaluate the last expression `( 5*2 )` to obtain the final answer of 10

- **We need a specific way to find the expression to be evaluated next and a way to remember the results of our intermediate calculations**

Programming Example: Evaluating Arithmetic Expressions (Cont'd)

- How to choose the next expression to be evaluated?
- We know that the expression to be evaluated first must be one of the innermost expressions

❑ Example: In `(((6 + 9)/3)*(6 - 4))` the innermost expressions are `(6 + 9)` and `(6 - 4)`, and the leftmost one of these is `(6 + 9)`

  - If we evaluate this leftmost of the innermost expressions, we obtain `((15/3)*(6 - 4))`, now go back and evaluate the other innermost expression `(6 - 4)`

- There is a simpler approach that spares us the trouble of remembering any other expressions
  - After we evaluate the leftmost of the innermost expressions, we are left with another simpler arithmetic expression, namely `((15/3)*(6 - 4))`, so we can simply repeat the process with this simpler expression: We again evaluate the leftmost of the innermost expressions of our new simpler expression

Programming Example: Evaluating Arithmetic Expressions (Cont'd)

- **Design:** The entire process will look like
  1. Evaluate the leftmost of the innermost expressions in
     `((( 6 + 9 ) / 3) * (6 - 4))`
     to produce the simpler expression `(( 15 /3 ) * ( 6 - 4 ))`
  2. Evaluate the leftmost of the innermost expressions in
     `(( 15 /3 ) * ( 6 - 4 ))`
     to produce the simpler expression `( 5 * ( 6 - 4 ))`
  3. Evaluate the leftmost of the innermost expressions in
     `( 5 * ( 6 - 4 ))`
     to produce the simpler expression `( 5 * 2 )`
  4. Evaluate the leftmost of the innermost expressions in
     `( 5 * 2 )`
     to obtain the final answer of 10

Programming Example: Evaluating Arithmetic Expressions (Cont'd)

- **Question:** How to find the leftmost of the innermost expressions?
- According to the expression `((( 6 + 9 ) / 3) * (6 - 4))`, **the end of the expression to be evaluated is always a right parenthesis, ')', and moreover, it is always the first right parenthesis**
- After evaluating one of these innermost expressions, there is no need to back up; to find the next right parenthesis we can just keep reading left to right from where we left off
- The next right parenthesis will indicate the end of the next expression to be evaluated

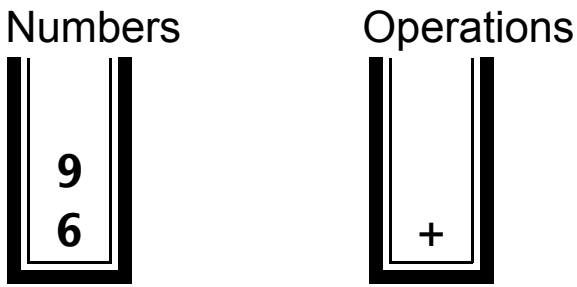
Programming Example: Evaluating Arithmetic Expressions (Cont'd)

- **Question:** How do we keep track of our intermediate values?
- **We use two stacks**
  - A stack contains numbers from the input as well as numbers that were computed when subexpressions were evaluated
  - The other stack will hold symbols for the operations that still need to be evaluated
- Because a stack processes data in a Last-In/First-Out manner, it will turn out that the correct two numbers are on the top of the numbers stack at the same time that the appropriate operation is at the top of the stack of operations

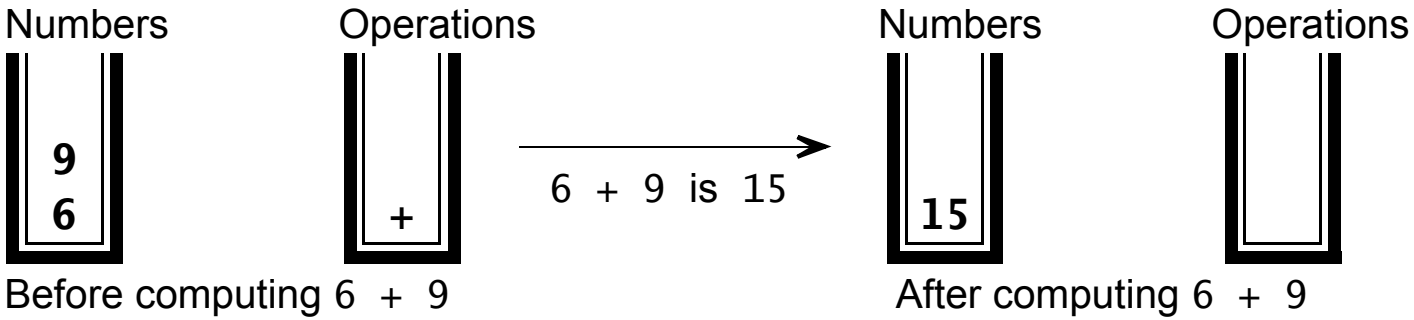
Programming Example: Evaluating Arithmetic Expressions (Cont'd)

- Example:
  - Characters read so far (shaded):

    (((6 + 9) / 3) * (6 - 4))

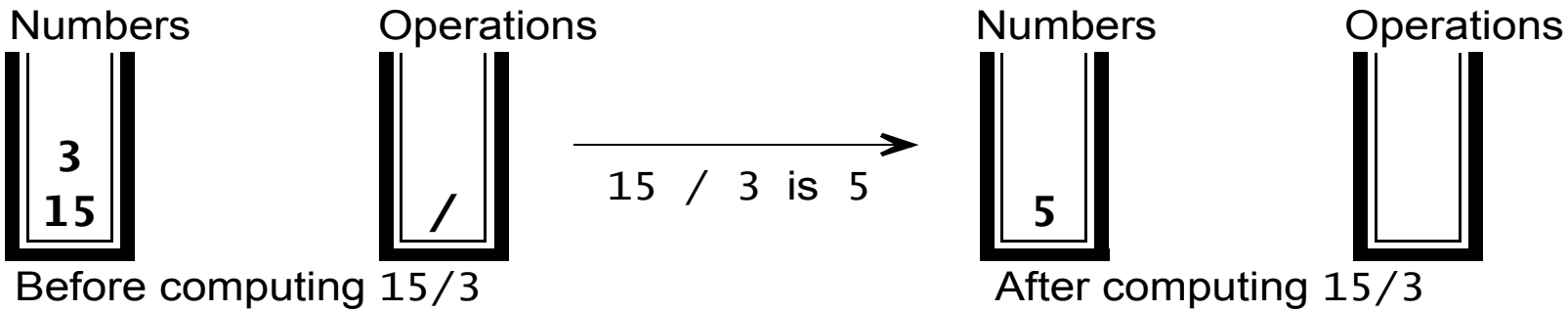


- **Whenever we reach a right parenthesis**, we combine the top two numbers (on the numbers stack) using the topmost operation (on the characters stack)

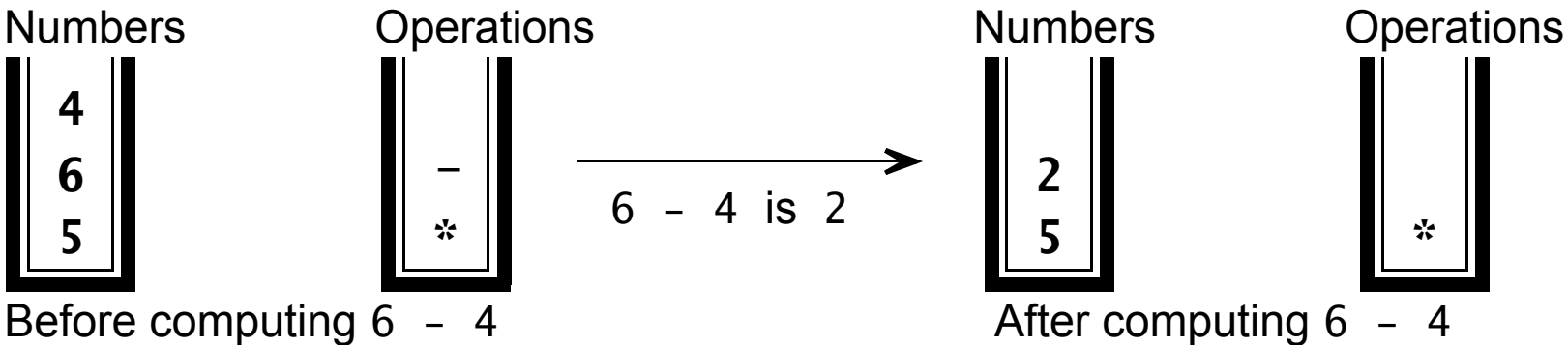


Programming Example: Evaluating Arithmetic Expressions (Cont'd)

- Simply continue the process:
  - Characters read so far (shaded): (((6 + 9) / 3) * (6 - 4))

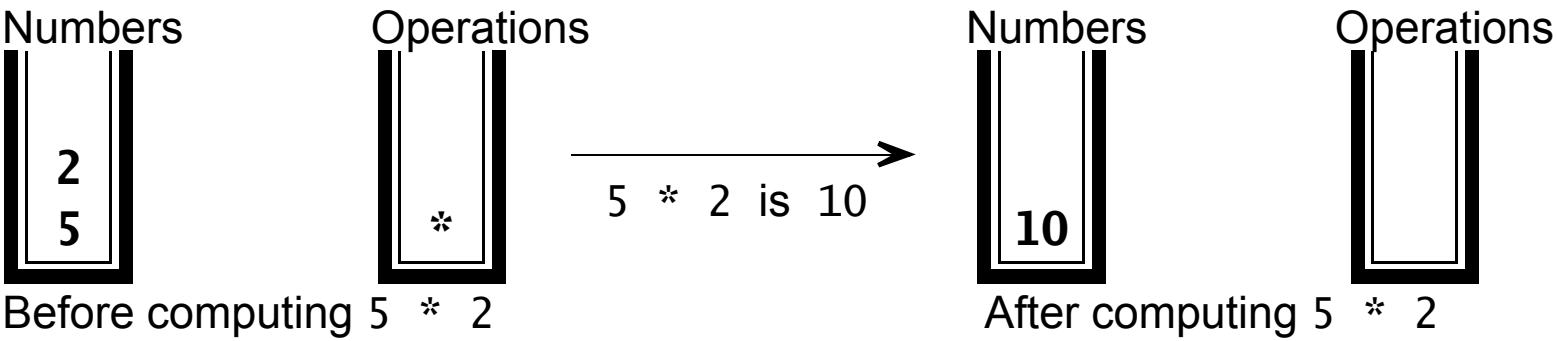


  - Characters read so far (shaded): (((6 + 9) / 3) * (6 - 4))



Programming Example: Evaluating Arithmetic Expressions (Cont'd)

- Simply continue the process:
  - Characters read so far (shaded): (((6 + 9) / 3) * (6 - 4))



- At this point there is no more input, and there is exactly one number in the numbers stack, namely 10
- 10 is the answer

Programming Example: Evaluating Arithmetic Expressions (Cont'd)

Handle each input item according to the following cases:

- **Number.** Read and push onto the numbers stack
- **Operation character.** Read and push onto the operations stack
- **Right parenthesis.** An **evaluation** takes place:
  - Takes the top two numbers from the numbers stack and the top operation from the operations stack
  - These items are removed from their stacks and the two numbers are combined using the operation (with the second number popped as the left operand)
  - The result of the operation is pushed back onto the numbers stack.
- **Left parenthesis or blank.** Read and throw away

More Complex Stack Applications

More Complex Stack Applications

Evaluating Postfix Expressions

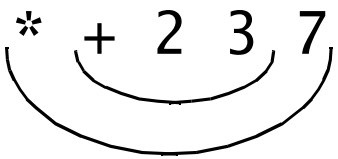
- We normally write an arithmetic operation between its two arguments

❑Example: The + operation occurs between the 2 and the 3 in the arithmetic expression `2 + 3`

  - This is called **infix notation**

- There is another way of writing arithmetic operations that **places the operation in front of the two arguments**
  - This is called **Polish prefix notation** or simply **prefix notation**
- Using prefix notation, parentheses are completely avoided

❑Example: The expression `(2 + 3) * 7`, when written in this **Polish prefix notation**:



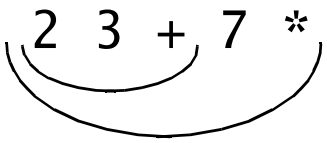
- The curved lines under the expression indicate groupings of subexpressions

More Complex Stack Applications

Evaluating Postfix Expressions (Cont'd)

- We can write the operations after the two numbers being combined
  - Called **Polish postfix notation**, or more simply **postfix notation** (or sometimes **reverse Polish notation**)

❑Example: The expression (2 + 3) * 7 when written in **Polish postfix notation** is:



❑Longer example: The **postfix expression** `7 3 5 * + 4 -` is equivalent to the infix expression `(7 + (3*5)) - 4`

- **Postfix notation** is handy because it does not require parentheses and because it is particularly easy to evaluate
  - **This notation often is used because of the ease of expression evaluation**

More Complex Stack Applications

Evaluating Postfix Expressions (Cont'd)

- There are two input format issues that we must handle:
  - When entering postfix notation we will require a space between two consecutive numbers
    - ❑Example: The input **35 6** consists of two numbers, **35** and **6**, with a space in between
  - For now, restrict the input to non-negative numbers in order to avoid the complication of distinguishing the negative sign of a number from a binary subtraction operation

- **Using postfix notations: Each operation is used as soon as it is read**
  - **Our algorithm for evaluating a postfix expression uses only one stack, which is a stack of numbers**
  - **There is no need for a second stack of operation symbols**

More Complex Stack Applications

Evaluating Postfix Expressions (Cont'd)

- **Pseudocode**

1. Initialize a stack of double numbers
2.
```
do
  if (the next input is a number)
     Read the next input and push it onto the stack

  else
  {
     - Read the next character, which is an operation symbol
     - Use top and pop to get the two numbers off the stack
     - Combine these two numbers with the operation (using the
       second number popped as the left operand), and push the
       result onto the stack
   }

while (there is more of the expression to read);
```
3. At this point, the stack contains one number, which is the value of the expression

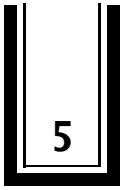
More Complex Stack Applications

Evaluating Postfix Expressions (Cont'd)

- Example: Evaluate the postfix expression 5 3 2 ∗ + 4 –5 +

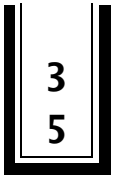
(a) Input so far (shaded):

5 3 2 * + 4 – 5 +



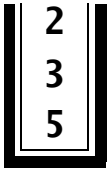
(b) Input so far (shaded):

5 3 2 * + 4 – 5 +



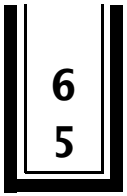
(c) Input so far (shaded):

5 3 2 * + 4 – 5 +



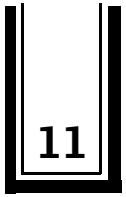
(d) Input so far (shaded):

5 3 2 * + 4 – 5 +



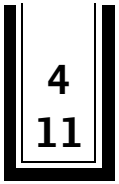
(e) Input so far (shaded):

5 3 2 * + 4 – 5 +



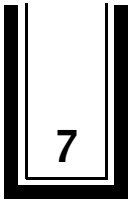
(f) Input so far (shaded):

5 3 2 * + 4 – 5 +



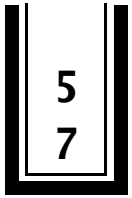
(g) Input so far (shaded):

5 3 2 * + 4 – 5 +



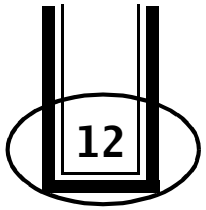
(h) Input so far (shaded):

5 3 2 * + 4 – 5 +



(i) Input so far (shaded):

5 3 2 * + 4 – 5 +



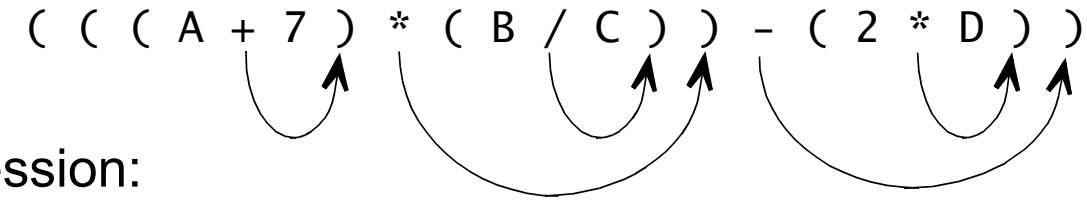
The result of the computation is 12.

More Complex Stack Applications

Translating Infix to Postfix Notation

- **One strategy for evaluating an ordinary infix expression is to first convert it to postfix notation and then evaluate the postfix expression**
  - This is what compilers often do
- **If the infix expression is fully parenthesized, the algorithm is simple:**
  - Move each operation symbol to the location of the right parenthesis corresponding to that operation and then remove all parentheses

❑Example: The following infix expression will have its operation symbols moved to the location indicated by the arrows:



- The result is the postfix expression:

```
A 7 + B C / * 2 D * -
```

- A complete algorithm should read the expression from left to right and must somehow remember the operations and then determine when the corresponding right parenthesis has been found

More Complex Stack Applications

Translating Infix to Postfix Notation (Cont'd)

- The problem is finding the location for inserting the operations in the postfix expression?
  - How do we save the operations?
  - How do we know when to insert them?

- **If we push the operations onto a stack, then the operations we need will always be on top of the stack**
- **The heart of the algorithm is to push the operations onto a stack and to pop an operation every time we encounter a right parenthesis**

Translating Infix to Postfix Notation (Cont'd)

- **Pseudocode**

1. Initialize a stack of characters to hold the operation symbols and parentheses

More Complex Stack Applications

Translating Infix to Postfix Notation (Cont'd)

```
2. do
     if (the next input is a left parenthesis)
       Read the left parenthesis and push it onto the stack

     else if (the next input is a number or other operand)
       Read the operand and write it to the output

     else if (the next input is one of the operation symbols)
       Read the operation symbol and push it onto the stack

     else
     {
        - Read and discard the next input symbol (which should be a right
        parenthesis
        - There should be an operation symbol on top of the stack, so write
        this symbol to the output and pop it from the stack
        - After popping the operation symbol, there should be a left
        parenthesis on the top of the stack, so pop and discard this left
        parenthesis
      }

   while (there is more of the expression to read);
```

Translating Infix to Postfix Notation (Cont'd)

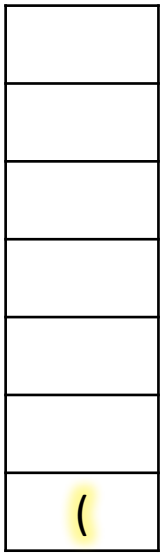
3. At this point, the stack should be empty; otherwise print an error message indicating that the expression was not fully parenthesized

More Complex Stack Applications

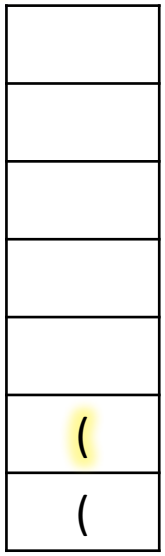
Translating Infix to Postfix Notation (Cont'd)

- Example: Converting ((A + B) * 3) to postfix expression

((A + B) * 3)



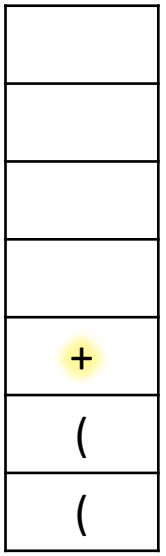
((A + B) * 3)



((A + B) * 3)

Output: A

((A + B) * 3)



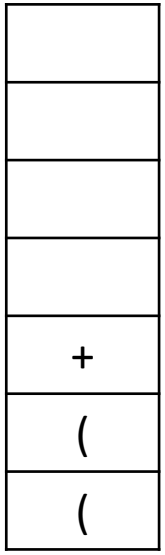
Output: A

More Complex Stack Applications

Translating Infix to Postfix Notation (Cont'd)

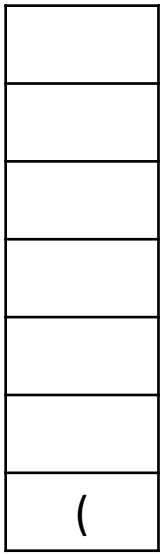
- Example: ((A + B) * 3)

((A + B) * 3)



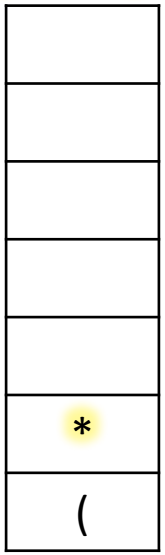
Output: A B

((A + B) * 3)



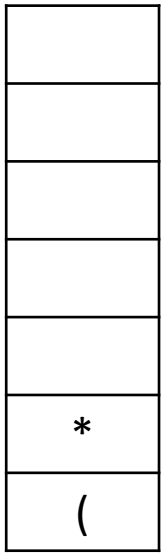
Output: A B +

((A + B) * 3)



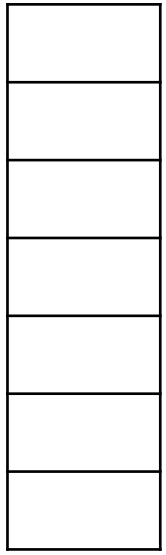
Output: A B +

((A + B) * 3)



Output: A B + 3

((A + B) * 3)



Output: A B + 3 *

Summary

Summary

- A stack is a Last-In/First-Out (**LIFO**) data structure
- The accessible end of the stack is called the **top**
- Adding an entry to a stack is called a **push** operation
- Removing an entry from a stack is called a **pop** operation
- Attempting to push an entry onto a full stack is an error known as a **stack overflow**
- Attempting to pop an entry off an empty stack is an error known as a **stack underflow**
- A stack can be implemented as a **partially filled array** or a **linked list**
- Stacks have many uses in computer science
  - The **evaluation and translation of arithmetic expressions** are two common uses

References

1) Data Structures and Other Objects Using C++, Michael Main, Walter Savitch, 4th Edition
2) Data Structures and Algorithm Analysis in C++, by Mark A. Weiss, 4TH Edition
3) C++: Classes and Data Structures, by Jeffrey Childs
4) http://en.cppreference.com
5) http://www.cplusplus.com
6) https://isocpp.org