Data Structures — Homework Assignment #4

Write a C**++** non-recursive function int evaluateInfix( string &expr ) to evaluate an infix expression which is not necessarily fully parenthesized. You may assume all numbers in the expression are one digit integers, and your program should handle at least the following five operators: ****, ****, ****, **** and ****. Your program should contain three class templates Iterator, Vector and Stack. Their definitions are as follows:

template< typename T >

class Iterator

{

// friend ostream &operator<<( ostream &, const Iterator & );

public:

Iterator( T \*p = 0 ); // default constructor

Iterator( const Iterator< T > &iteratorToCopy ); // copy constructor

T& operator\*() const; // dereferencing operator

Iterator< T >& operator=( const Iterator< T > &right ); // assignment operator

Iterator< T >& operator++(); // prefix increment operator

Iterator< T > operator++( int ); // postfix increment operator

unsigned int operator-( Iterator< T > &right ) const; // subtraction operator; Iterator - Iterator

bool operator<( const Iterator< T > &right ) const; // less than

private:

T \*ptr; // keep a pointer to Vector

}; // end class Iterator

template< typename T >

class Vector

{

public:

typedef Iterator< T > iterator;

Vector(); // Constructs an empty container

Vector( unsigned int n, const T val ); // Constructs a container with n elements. Each element is a copy of val.

Vector( const Vector< T > &vectorToCopy ); // Constructs a container with a copy of each of the elements in x, in the same order.

~Vector(); // Destroys the container object.

const Vector< T > &operator=( const Vector< T > &right ); // assignment operator

T &operator[]( unsigned int ); // subscript operator for non-const objects returns modifiable lvalue

T operator[]( unsigned int ) const; // subscript operator for const objects returns rvalue

Iterator< T > begin(); // Returns a pointer pointing to the first element in the vector.

// If the container is empty, the returned pointer shall not be dereferenced.

Iterator< T > end(); // Returns an pointer referring to the past-the-end element in the vector container.

// The past-the-end element is the theoretical element that would follow the last element in the vector.

// It does not point to any element, and thus shall not be dereferenced.

unsigned int getSize() const; // Returns the number of elements in the vector.

bool empty() const; // Returns whether the vector is empty (i.e. whether its size is 0).

T& back(); // Returns a reference to the last element in the vector.

// Calling this function on an empty container causes undefined behavior.

void push\_back( const T val ); // Adds a new element at the end of the vector, after its current last element.

// The content of val is copied to the new element.

// This effectively increases the container size by one,

// which causes an automatic reallocation of the allocated storage space

// if -and only if- the new vector size surpasses the current vector capacity.

void pop\_back(); // Removes the last element in the vector, effectively reducing the container size by one.

private:

unsigned int size; // the number of elements in the vector

// This is the number of actual objects held in the vector, which is not necessarily equal to its storage capacity.

unsigned int capacity; // the size of the storage space currently allocated for the vector, expressed in terms of elements.

// This capacity is not necessarily equal to the vector size. It can be equal or greater,

// with the extra space allowing to accommodate for growth without the need to reallocate on each insertion.

T \*ptr; // points to a dynamically allocated array which is used to store the elements of the vector

}; // end class Vector

template< typename T >

class Stack

{

public:

Stack(); // Constructs a stack container adaptor object, which is initially empty.

bool empty() const; // Returns whether the stack is empty.

unsigned int size() const; // Returns the number of elements in the stack.

T& top(); // Returns a reference to the top element in the stack.

const T& top() const; // Inserts a new element at the top of the stack, above its current top element.

// The content of this new element is initialized to a copy of val.

void push( const T &val );

void pop(); // Removes the element on top of the stack, effectively reducing its size by one.

private:

Vector< T > stack; // a vector which is used to store the elements of the stack

}; // end class Stack

The suggested main function is as follows:

|  |
| --- |
| int main()  {  string expr;  cout << "Enter an infix expression: ";  cin >> expr;  cout << endl << expr << " = " << evaluateInfix( expr ) << endl << endl;  system( "pause" );  } |

Sample outputs:

|  |
| --- |
| **Enter an infix expression: 5\*((8+2\*3)/2)**  **5\*((8+2\*3)/2) = 35**  **請按任意鍵繼續 . . .** |