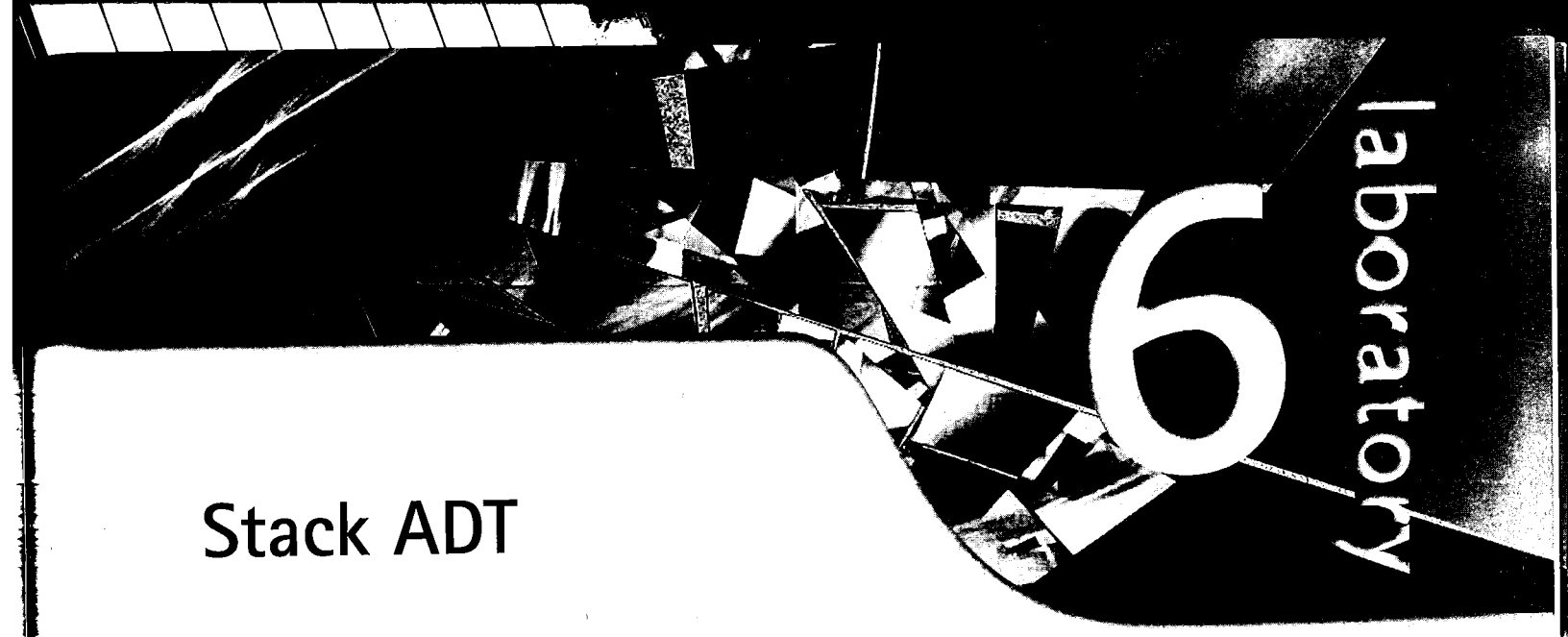


laboratory 6

Stack ADT

Objectives

In this laboratory you

- use an abstract base class as an ADT interface.
- create two implementations of the Stack ADT—one using an array representation of stack, the other using a singly linked list representation.
- analyze the kinds of permutations you can produce using a stack.

ADT Overview

Many applications that use a linear data structure do not require the full range of operations supported by the List ADT. Although you can develop these applications using the List ADT, the resulting programs are likely to be somewhat cumbersome and inefficient. An alternative approach is to define new linear data structures that support more constrained sets of operations. By carefully defining these ADTs, you can produce ADTs that meet the needs of a diverse set of applications but yield data structures that are easier to apply—and are often more efficient—than the List ADT.

The stack is one example of a constrained linear data structure. In a stack, the data items are ordered from most recently added (the top) to least recently added (the bottom). All insertions and deletions are performed at the top of the stack. You use the push operation to insert a data item onto the stack and the pop operation to remove the topmost stack data item. A sequence of pushes and pops is shown here.

Push a	*Push* b	*Push* c	*Pop*	*Pop*
		c		
	b	b	b	
a	a	a	a	a
—	—	—	—	—

These constraints on insertion and deletion produce the "last in, first out"—LIFO—behavior that characterizes a stack. Although the stack data structure is narrowly defined, it is so extensively used by systems software that support for a primitive stack is one of the basic data items of most computer architectures.

The stack is one of the most frequently used data structures. Although all programs share the same definition of stack—a sequence of homogeneous data items with insertion and removal done at one end—the type of data item stored in stacks varies from program to program. Some use stacks of integers, others use stacks of characters, floating-point numbers, points, and so forth.

C++ Concepts Overview

Abstract base classes: An abstract base class (ABC) is any class with at least one pure virtual function (see definition below). An ABC is called abstract because it cannot be directly instantiated. A common reason for creating an ABC is to specify an interface for a set of derived classes to implement. For instance, there are good reasons for implementing a stack with arrays or with linked data structures. We provide a stack ABC to specify the functionality that any implementation must provide. Then we derive the linked and array-based implementations from the stack ABC. The benefit is that code that uses a stack, but doesn't care about the stack's implementation can refer to the ABC, while the small portion of code that does care can explicitly reference the appropriate derived class.

Pure virtual function: This is a member function that is declared, but not implemented, in a base class. The function must be implemented in any derived class that will be instantiated. Because the base class does not implement any pure virtual functions, we cannot create an object of that type since there is no code for those member functions.

Stack ADT

Data items:

The data items in a stack are of generic type DataType.

Structure:

The stack data items are linearly ordered from most recently added (the top) to least recently added (the bottom). Data items are inserted onto (pushed) and removed from (popped) the top of the stack.

Operations:

```
Stack ( int maxNumber = MAX_STACK_SIZE )
```

Requirements:
None

Results:
Constructor. Creates an empty stack. Allocates enough memory for a stack containing `maxNumber` data items (if necessary).[1]

```
Stack ( const Stack& other )
```

Requirements:
None

Results:
Copy constructor. Initializes the stack to be equivalent to the `other` Stack object parameter.[1]

```
Stack& operator= ( const Stack& other )
```

Requirements:
None

Results:
Overloaded assignment operator. Sets the stack to be equivalent to the `other` Stack object parameter and returns a reference to the modified stack.[2]

```
~Stack ()
```

Requirements:
None

Results:
Destructor. Deallocates (frees) the memory used to store the stack.

---

[1]Because of the way ABCs work, constructors are actually declared and implemented in the derived classes, not in the base class. This relates to the requirement that constructor names match the class names; it is impossible for the base class and the derived class to have the same name.

[2]Like ABC constructors, `operator=` is declared and implemented in the derived classes with corresponding name changes.

```
void push ( const DataType& newDataItem ) throw ( logic_error )
```

Requirements:
Stack is not full.

Results:
Inserts `newDataItem` onto the top of the stack.

```
DataType pop () throw ( logic_error )
```

Requirements:
Stack is not empty.

Results:
Removes the most recently added (top) data item from the stack and returns the value of the deleted item.

```
void clear ()
```

Requirements:
None

Results:
Removes all the data items in the stack.

```
bool isEmpty () const
```

Requirements:
None

Results:
Returns `true` if the stack is empty. Otherwise, returns `false`.

```
bool isFull () const
```

Requirements:
None

Results:
Returns true if the stack is `full`. Otherwise, returns `false`.

```
void showStructure () const
```

Requirements:
None

Results:
Outputs the data items in a stack. If the stack is empty, outputs "Empty stack". Note that this operation is intended for testing/debugging purposes only. It only supports stack data items that are one of C++'s predefined data types (`int`, `char`, and so forth) or other data structures with an overridden ostream `operator<<`.

Implementation Notes

Pure virtual function: A member function is identified as a pure virtual function in the class declaration by the word `virtual`, followed by the class prototype, followed by the string "=0". For instance, the Stack ABC `isEmpty` function is declared as follows:

```
virtual bool isEmpty() const = 0;
```

Stack implementations: Multiple versions of an ADT may be necessary if the ADT is to perform efficiently in a variety of operating environments. Depending on the hardware and the application, you may want an implementation that reduces the execution time of some (or all) of the ADT operations, or you may want an implementation that reduces the amount of memory used to store the ADT data items. In this laboratory, you develop two implementations of the Stack ADT. One implementation stores the stack in an array, the other stores each data item separately and links the data items together to form a stack.

Array-Based Implementation

Step 1: Implement the operations in the Stack ADT using an array to store the stack data items. Stacks change in size, therefore you need to store the maximum number of data items the stack can hold (`maxSize`) and the array index of the topmost data item in the stack (`top`), along with the stack data items themselves (`dataItems`). Base your implementation on the declarations from the file *StackArray.h*. An implementation of the showStructure operation is given in the file *show6.cpp*.

Step 2: Save your array implementation of the Stack ADT in the file *StackArray.cpp*. Be sure to document your code.

Compilation Directions

Compile test6.cpp. The value of LAB6_TEST1 in *config.h* determines whether the array-based implementation or the linked-list implementation is tested. If the value is 0 (the default), the array implementation is tested. If the value is 1, then the linked implementation is tested.

Testing

Test your implementation of the array-based Stack ADT using the program in the file *test6.cpp*. The test program allows you to interactively test your ADT implementation using the commands in the following table.

Command	Action
+x	Push data item *x* onto the top of the stack.
-	Pop the top data item and output it.
E	Report whether the stack is empty.
F	Report whether the stack is full.
C	Clear the stack.
Q	Exit the test program.

Step 1: Download the online test plans for Lab 6.

Step 2: Complete the test plan for Test 6-1 by filling in the expected results for each given operation. Add test cases in which you

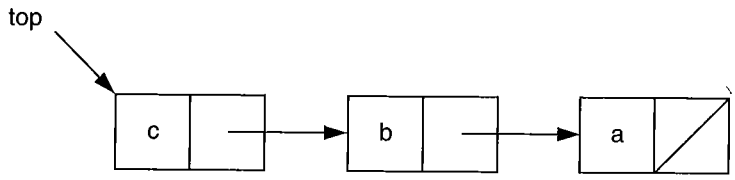
- pop a data item from a stack containing only one data item,
- push a data item onto a stack that has been emptied by a series of pops,
- pop a data item from a full stack (array implementation), and
- clear the stack.

Step 3: Execute Test Plan 6-1. If you discover mistakes in your implementation of the Stack ADT, correct them and execute the test plan again.

Linked-List Implementation

In your array implementation of the Stack ADT, you allocate the memory used to store a stack when the stack is declared (constructed). The resulting array must be large enough to hold the largest stack you might possibly need in a particular application. Unfortunately, most of the time the stack will not actually be this large and the extra memory will go unused.

An alternative approach is to allocate memory data item-by-data item as new data items are added to the stack. In this way, you only allocate memory when you actually need it. Because memory is allocated over time, however, the data items do not occupy a contiguous set of memory locations. As a result, you need to link the data items together to form a linked list representation of a stack, as shown in the following figure.



Creating a linked list implementation of the Stack ADT presents a somewhat more challenging programming task than did developing an array implementation. One way to simplify this task is to divide the implementation into two template classes: one focusing on the overall stack structure (the Stack class) and another focusing on the individual nodes in the linked list (the StackNode class).

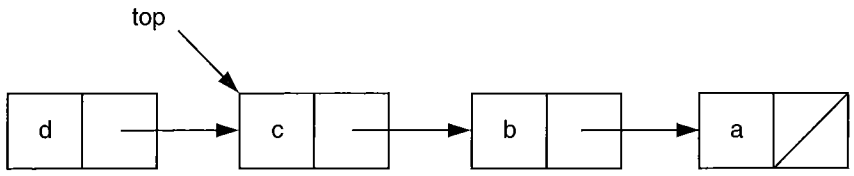
Let's begin with the StackNode class. Each node in the linked list contains a stack data item and a pointer to the node containing the next data item in the list. The only function provided by the StackNode class is a constructor that creates a specified node.

Access to the StackNode class is restricted to member functions of the Stack class. Other classes are blocked from referencing linked list nodes directly by declaring the StackNode as an inner class of Stack. (Refer to the Lab 5 C++ concepts and implementation sections for details.)

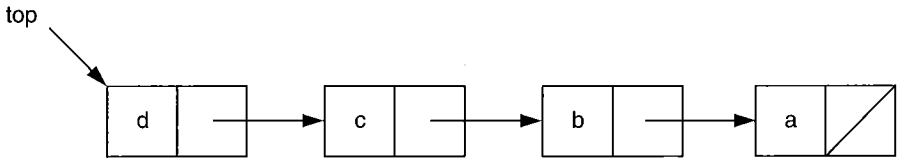
The StackNode class constructor is used to add nodes to the stack. The statement below, for example, adds a node containing `newDataItem` (`'d'` in this example) to a stack of characters. Note that `top` is of type `StackNode*`.

```
top = new StackNode<DataType>(newDataItem,top);
```

The `new` operator allocates memory for a linked list node and calls the StackNode constructor passing both the data item to be inserted (`'d'`) and a pointer to the next node in the list (`top`).



Finally, the assignment operator assigns a pointer to the newly allocated node to `top`, thereby completing the creation and linking of the node.



The member functions of the Stack class implement the operations in the Stack ADT. A pointer is maintained to the node at the beginning of the linked list or, equivalently, the top of the stack. The following declaration for the Stack class is given in the file *StackLinked.h*.

Step 1: Implement the operations in the Stack ADT using a singly linked list to store the stack data items. Each node in the linked list should contain a stack data item (`dataItem`) and a pointer to the node containing the next data item in the stack (`next`). Your implementation should also maintain a pointer to the node containing the topmost data item in the stack (`top`). Base your implementation on the class declarations in the file *StackLinked.h*. A linked-list implementation of the showStructure operation is given in the file *show6.cpp*.

Step 2: Save your linked list implementation of the Stack ADT in the file *StackLinked.cpp*. Be sure to document your code.

Compilation Directions

Edit *config.h* and change the value of LAB6_TEST1 to 1. (If the value is 0, then the array-based implementation is tested instead.) Recompile *test6.cpp*.

Testing

Test your implementation of the linked list Stack ADT using the program in the file *test6.cpp*.

Step 1: Re-execute Test Plan 6-1. If you discover mistakes in your linked-list implementation of the Stack ADT, correct them and execute your test plan again.

Programming Exercise 1

We commonly write arithmetic expressions in infix form, that is, with each operator placed between its operands, as in the following expression.

(3 + 4) * (5 / 2)

Although we are comfortable writing expressions in this form, infix form has the disadvantage that parentheses must be used to indicate the order in which operators are to be evaluated. These parentheses, in turn, greatly complicate the evaluation process.

Evaluation is much easier if we can simply evaluate operators from left to right. Unfortunately, this evaluation strategy will not work with the infix form of arithmetic expressions. However, it will work if the expression is in postfix form. In the postfix form of an arithmetic expression, each operator is placed immediately after its operands. The expression above is written in postfix form as

3 4 + 5 2 / *

Note that both forms place the numbers in the same order (reading from left to right). The order of the operators is different, however, because the operators in the postfix form are positioned in the order that they are evaluated. The resulting postfix expression is hard to read at first, but it is easy to evaluate. All you need is a stack on which to place intermediate results.

Suppose you have an arithmetic expression in postfix form that consists of a sequence of single digit, nonnegative integers and the four basic arithmetic operators (addition, subtraction, multiplication, and division). This expression can be evaluated using the following algorithm in conjunction with a stack of floating-point numbers.

Read in the expression character-by-character. As each character is read in:

- If the character corresponds to a single digit number (characters `'0'` to `'9'`), then push the corresponding floating-point number onto the stack.
- If the character corresponds to one of the arithmetic operators (characters `'+'`, `'-'`, `'*'`, and `'/'`), then
- Pop a number off of the stack. Call it *operand1*.
- Pop a number off of the stack. Call it *operand2*.
- Combine these operands using the arithmetic operator, as follows:
  Result = *operand2* operator *operand1*
- Push *result* onto the stack.

When the end of the expression is reached, pop the remaining number off the stack. This number is the value of the expression.

Applying this algorithm to the arithmetic expression

3 4 + 5 2 / *

yields the following computation

`'3'` : Push 3.0

`'4'` : Push 4.0

`'+'` : Pop, *operand1* = 4.0
Pop, *operand2* = 3.0
Combine, *result* = 3.0 + 4.0 = 7.0
Push 7.0

`'5'` : Push 5.0

`'2'` : Push 2.0

`'/'` : Pop, *operand1* = 2.0
Pop, *operand2* = 5.0
Combine, *result* = 5.0 / 2.0 = 2.5
Push 2.5

`'*'` : Pop, *operand1* = 2.5
Pop, *operand2* = 7.0
Combine, *result* = 7.0 * 2.5 = 17.5
Push 17.5

`'\n'` : Pop, Value of expression = 17.5

Step 1: Create a program that reads the postfix form of an arithmetic expression, evaluates it, and outputs the result. Assume that the expression consists of single-digit, nonnegative integers (`'0'` to `'9'`) and the four basic arithmetic operators (`'+'`, `'-'`, `'*'`, and `'/'`). Further assume that the arithmetic expression is input from the keyboard with all the characters separated by white space on one line. Save your program in a file called *postfix.cpp*.

Step 2: Complete Test Plan 6-2 by filling in the expected result for each arithmetic expression. You may wish to include additional arithmetic expressions in this test plan.

Step 3: Execute the test plan. If you discover mistakes in your program, correct them and execute the test plan again.

Programming Exercise 2

A classic computer science problem that can be solved with a stack is called the 8-queens problem. The question is whether it is possible to safely place eight queens on a chessboard. The answer is yes, so the question is often modified to list one or more safe scenarios.

The standard algorithm is to place a queen in a potentially safe spot and check whether it is safe. If it is safe, leave it and try placing another queen; otherwise, remove it and try placing it in another place. If no safe locations can be found, then a queen previously declared safe must also be removed. Continue until all eight queens are safely on the board. The process of trying a solution that may require undoing is called backtracking.

We use a stack to facilitate backtracking. When a queen is thought to be safely placed, we push the queen's location on to the stack. When we need to remove a queen, we pop the location from the stack.

Step 1: Write an implementation of the previous algorithm. We provide an implementation of a number of routines to represent and manipulate the queens on the board. You must maintain the stack and track queen placement.

Step 2: Save a copy of the file *queens.cs* as *queens.cpp*. Write your implementation of the 8-queens main algorithm in the file *queens.cpp*.

Step 3: Compile and run the program in *queens.cpp*.

Step 4: Visually verify that the printed solution is valid. Because visual inspection on the screen can make it hard to determine whether your solution is valid, Test Plan 6-3 is available for you to write the results if you wish.

Programming Exercise 3

One of the tasks that compilers and interpreters must frequently perform is deciding whether some pair of expression delimiters are properly paired, even if they are embedded multiple pairs deep. Consider the following C++ expression.

```
a=(f(b)-(c+d))/2;
```

The compiler has to be able to determine which pairs of opening and closing parentheses go together and whether the whole expression is correctly parenthesized. A number of possible errors can occur because of incomplete pairs of parentheses—more of one than the other—or because of improperly placed parentheses. For instance, the expression below lacks a closing parenthesis.

```
a=(f(b)-(c+d)/2;
```

A stack is extremely helpful in implementing solutions to this type of problem because of its LIFO—Last In, First Out—behavior. A closing parenthesis needs to be matched with the most recently encountered opening parenthesis. This is handled by pushing opening parentheses onto a stack as they are encountered. When a closing parenthesis is encountered, it should be possible to pop the matching opening parenthesis off the stack. If it is determined that every closing parenthesis had a matching opening parenthesis, then the expression is valid.

```
bool delimitersOk( const string& expression )
```

Requirements:
None

Results:
Returns `true` if all the parentheses and braces in the string are legally paired. Otherwise, returns `false`.

Step 1: Save a copy of the file *delimiters.cs* as *delimiters.cpp*. Implement the delimitersOk operation inside the *delimiters.cpp* program.

Step 2: Complete Test Plan 6-4 by adding test cases that check whether your implementation of the delimitersOk operation correctly detects improperly paired delimiters in input expressions. Note that it is not required that the input be valid C++ expressions, just the delimiters are correct.

Step 5: Execute your test plan. If you discover mistakes in your implementation of the delimitersOk operation, correct them and execute the test plan again.

Analysis Exercise 1

Given the input string `"abc"`, which permutations of this string can be output by a code fragment consisting of only the statement pairs

```
cin >> ch;    permuteStack.push(ch);
```

and

```
ch = permuteStack.pop();    cout << ch;
```

where `ch` is a character and `permuteStack` is a stack of characters? Note that each of the statement pairs may be repeated several times within the code fragment and that the statement pairs may be in any order. For instance, the code fragment

```
cin >> ch;    permuteStack.push(ch);
cin >> ch;    permuteStack.push(ch);
cin >> ch;    permuteStack.push(ch);
ch = permuteStack.pop();    cout << ch;
ch = permuteStack.pop();    cout << ch;
ch = permuteStack.pop();    cout << ch;
```

outputs the string `"cba"`.

Part A

For each of the permutations listed below, give a code fragment that outputs the permutation or a brief explanation of why the permutation cannot be produced.

`"abc"`	`"acb"`
`"bac"`	`"bca"`
`"cab"`	`"cba"`

Part B

Given the input string `"abcd"`, which permutations beginning with the character `'d'` can be output using the same code fragment combinations (e.g., cin/push, pop/cout) described previously? Why can only these permutations be produced?

Analysis Exercise 2

For each of the stack implementations, identify the performance order of magnitude (big-O value) for the listed operations. Then provide a justification for your big-O value.

Operation	Array-based	Linked
push	O( )	O( )
Justification		
pop	O( )	O( )
Justification		
clear	O( )	O( )
Justification		