laboratory 5

Stack ADT

Objectives

In this laboratory you will

- Create two implementations of the Stack ADT—one based on an array representation of stack, the other based on a singly linked list representation
- Use templates to produce a generic stack data structure
- Create a program that evaluates arithmetic expressions in postfix form
- Create a program that evaluates expressions for properly balanced pairs of parentheses and braces
- Analyze the kinds of permutations you can produce using a stack

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 below.

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 items 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.

In Lab 3 you dealt with this problem by using the C++ `typedef` statement. That approach can be made to work, but it is laborious and error-prone. We mentioned that a better approach would be introduced here, in Lab 5.

That better approach is the C++ **template class**. A template is something that serves as a pattern. The pattern is not the final product, but is used to enable faster production of a final product. C++ template classes—of which the stack is an example—save you from needing to create a different stack implementation for each type of stack data item and from constantly playing with `typedef`. Instead, you create the stack implementation in terms of a **generic** data type. Every place in your code where you would normally have to specify the data type, you instead use an arbitrary string to represent any actual data type that you might later wish to use. We will use the arbitrary string `"DT"`—for *D*ata *T*ype—to represent the generic data type. Nowhere does either the class declaration (the *class.h* file) or the class definition (the *class.cpp* file) specify an actual C++ data type. You can defer specifying the actual data type until it is time to **instantiate** (create) an object of that class.

Following are a few simple rules for creating and using a template class.

- The string "`template < class DT >`" must go right before the class declaration and before every class member function. Remember, `DT` is our arbitrary identifier that will represent any data type in the template class. So the lines

```
class Stack
{
  public:
    ...
```

  are changed to

```
template < class DT >
class Stack
{
  public:
    ...
```

  The start of a function definition that used to be

```
Stack:: Stack ( int maxNumber ) throw ( bad_alloc )
```

  now becomes

```
template < class DT >
Stack<DT>:: Stack ( int maxNumber ) throw ( bad_alloc )
```

  Every use of the class name now must include the generic data type name enclosed in angle brackets. Every instance of the string "`Stack`" becomes "`Stack<DT>`". In the example constructor definition, the class resolution "`Stack::`" becomes "`Stack<DT>::`". Also note that the exception to the rule is that the constructor name is not modified—it remains just "`Stack`".

```
template < class DT >
Stack<DT>:: Stack ( int maxNumber ) throw ( bad_alloc )
```

- Every occurrence of the data type name inside the class declaration and definition files gets replaced by the string chosen to represent the generic data type. For instance, the line

```
int *dataItems;  // Array containing the stack data items
                 //  (integers)
```

  becomes

```
DT *dataItems;   // Array containing the stack data items
                 //  (generic)
```

- When it is time to instantiate an object of that class, the real data type—inside angle brackets—is appended to the class name. So the lines

```
// A separate stack implementation just for integers
IntStack samples(10);

// Data type specified elsewhere by using typedef
Stack line(80);
```

now become

```
// We tell the compiler to make a copy of the generic stack just
// for integers and to make another just for characters.
Stack<int>  samples(10);
Stack<char> line(80);
```

- The code in the implementation file—the *classname.cpp* file—provides a template (or framework) for a set of implementations of the class ADT. The type of the data item is deliberately left unspecified in this framework and is not made specific until an object of the class is instantiated. As a result, the compiler must have access to the code in the .cpp file whenever it encounters a list declaration so that it can construct an implementation for the declared type of data item. Sophisticated program development environments provide a variety of mechanisms for ensuring access to this code. Unfortunately, these mechanisms are not yet standardized across systems. In this book you use a mechanism that is primitive but effective. You include the implementation file—*classname.cpp*—rather than the header file—*classname.h*—in any program that used this class. This approach violates the rule that you should never use a `#include` directive to include code. Most systems, however, provide no other means for using templated classes short of putting all the code for a program in one file (a much worse approach).

A partial template class declaration for the Stack ADT is shown below (the complete declaration is given in the Prelab Exercise).

```
template < class DT >
class Stack
{
  public:
    ...
    Stack ( int maxNumber = defMaxStackSize )       // Constructor
        throw ( bad_alloc );
    void push ( const DT &newDataItem )             // Push data item
        throw ( logic_error );
    DT pop ()                                       // Pop data item
        throw ( logic_error );
    ...
  private:
    ...
    DT *dataItems;  // Array containing the stack data items
};
```

Note the occurrences of the template parameter `DT` within the class declaration. This parameter is used to mark locations where explicit references are made to the stack data item type.

Stack ADT

Data items

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

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 = defMaxStackSize ) throw ( bad_alloc )
```

Requirements:
None

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

```
~Stack ()
```

Requirements:
None

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

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

Requirements:
Stack is not full.

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

```
DT pop () throw ( logic_error )
```

Requirements:
Stack is not empty.

Results:
Removes the most recently added (top) data item from a stack and returns it.

```
void clear ()
```

Requirements:
None

Results:
Removes all the data items in a stack.

```
bool isEmpty () const
```

Requirements:
None

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

```
bool isFull () const
```

Requirements:
None

Results:
Returns `true` if a 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).

Laboratory 5: Cover Sheet

Name _____ Date _____

Section _____

Place a check mark in the *Assigned* column next to the exercises your instructor has assigned to you. Attach this cover sheet to the front of the packet of materials you submit following the laboratory.

Activities	**Assigned:** Check or list exercise numbers	**Completed**
Prelab Exercise		
Bridge Exercise		
In-lab Exercise 1		
In-lab Exercise 2		
In-lab Exercise 3		
Postlab Exercise 1		
Postlab Exercise 2		
Total		

Laboratory 5: Prelab Exercise

Name _____ Date _____

Section _____

Multiple implementations of an ADT are 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 will 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.

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 following declarations from the file *stackarr.h*. An implementation of the `showStructure` operation is given in the file *show5.cpp*.

```
const int defMaxStackSize = 10;   // Default maximum stack size

template < class DT >
class Stack
{
  public:

    // Constructor
    Stack ( int maxNumber = defMaxStackSize ) throw ( bad_alloc );

    // Destructor
    ~Stack ();

    // Stack manipulation operations
    void push ( const DT &newDataItem )    // Push data item
        throw ( logic_error );
    DT pop ()                              // Pop data item
        throw ( logic_error );
    void clear ();                         // Clear stack

    // Stack status operations
    bool isEmpty () const;                 // Stack is empty
    bool isFull () const;                  // Stack is full

    // Output the stack structure -- used in testing/debugging
    void showStructure () const;
```

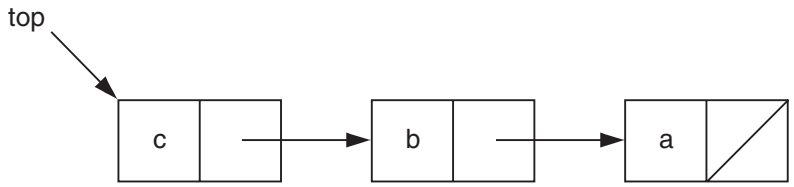
```
  private:

    // Data members
    int maxSize,    // Maximum number of data items in the stack
        top;        // Index of the top data item
    DT *dataItems;  // Array containing the stack data items
};
```

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

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 allocate memory only 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 templated 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 all the members of `StackNode` to be private. The members of `StackNode` are made accessible to the `Stack` class by declaring `Stack` to be a **friend** of `StackNode`. These properties are reflected in the following class declaration from the file *stacklnk.h*.

```
template < class DT >      // Forward declaration of the Stack class
class Stack;

template < class DT >
class StackNode            // Facilitator class for the Stack class
{
  private:
```

```
    // Constructor
    StackNode ( const DT &nodeData, StackNode *nextPtr );

    // Data members
    DT dataItem;           // Stack data item
    StackNode *next;       // Pointer to the next data item

  friend class Stack<DT>;
};
```

Notice the first two lines in the `StackNode` declaration above. The forward declaration is how C++ solves a classic compiler dilemma. `StackNode` makes reference to `Stack` in the statement

```
friend class Stack<DT>;
```

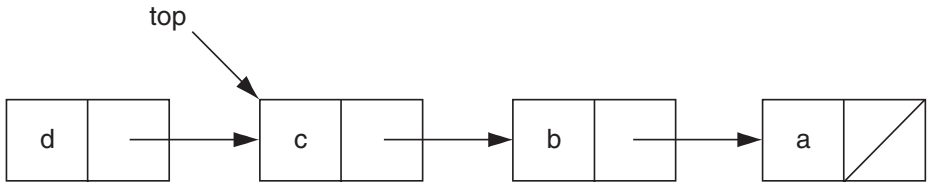
but the compiler has not yet encountered `Stack` and would normally issue an error message about referencing an unknown data type. We could move the declaration of `Stack` up above that of `StackNode`, thus ensuring that the compiler would have already seen `Stack` before it is referenced in `StackNode`. The problem that arises then is that the declaration of `Stack` contains a reference to `StackNode` before `StackNode` is declared. This is a catch-22, because they can't both occur first in the program.

C++ solves this problem by allowing the existence of a data type to be announced before it is actually declared. The compiler notes that it will be declared later and continues. This is similar to the introduction of a function prototype at the beginning of a program, well before the function definition is encountered.

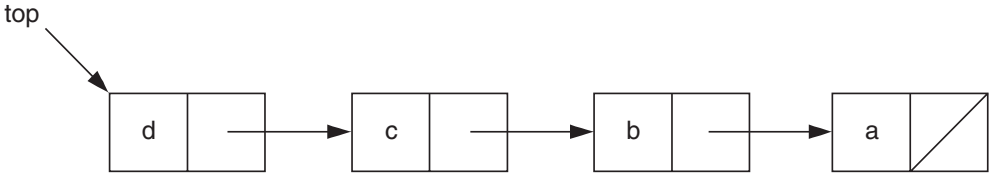
The `StackNode` class constructor is used to add nodes to the stack. The following statement, for example, adds a node containing '`d`' to a stack of characters. Note that template parameter `DT` must be equivalent to type `char` and `top` is of type `StackNode*`.

```
top = new StackNode<DT>('d',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 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 *stacklnk.h*.

```
template < class DT >
class Stack
{
  public:

    // Constructor
    Stack ( int ignored = 0 );

    // Destructor
    ~Stack ();

    // Stack manipulation operations
    void push ( const DT &newDataItem ) // Push data item
        throw ( bad_alloc );
    DT pop ()                           // Pop data item
        throw ( logic_error );
    void clear ();                      // Clear stack

    // Stack status operations
    bool isEmpty () const;              // Is stack empty?
    bool isFull () const;               // Is stack full?

    // Output the stack structure — used in testing/debugging
    void showStructure () const;

  private:

    // Data member
    StackNode<DT> *top;                 // Pointer to the top data item
};
```

Step 3: 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 also should 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 *stacklnk.h*. An implementation of the `showStructure` operation is given in the file *show5.cpp*.

Step 4: Save your linked list implementation of the Stack ADT in the file *stacklnk.cpp*. Be sure to document your code.

Laboratory 5: Bridge Exercise

Name _____ Date _____

Section _____

Check with your instructor whether you are to complete this exercise prior to your lab period or during lab.

The test program in the file *test5.cpp* allows you to interactively test your implementation of the Stack ADT using the following commands.

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: Compile and link the test program. Note that compiling this program will compile your array implementation of the Stack ADT (in the file *stackarr.cpp*) to produce an array implementation for a stack of characters.

Step 2: Complete the following test plan by adding 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)
- Clear the stack

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

Step 4: Modify the test program so that your linked list implementation of the Stack ADT in the file *stacklnk.cpp* is included in place of your array implementation.

Step 5: Recompile and relink the test program. Note that recompiling this program will compile your linked list implementation of the Stack ADT (in the file *stacklnk.cpp*) to produce a linked list implementation for a stack of characters.

Step 6: Use your test plan to check your linked list implementation of the Stack ADT. If you discover mistakes in your implementation, correct them and execute your test plan again.

Test Plan for the Operations in the Stack ADT

Test Case	*Commands*	*Expected Result*	*Checked*
Series of pushes	`+a +b +c +d`	`a b c d`	
Series of pops	`- - -`	`a`	
More pushes	`+e +f`	`a e f`	
More pops	`- -`	`a`	
Empty? Full?	`E F`	`False False`	
Empty the stack	`-`	Empty stack	
Empty? Full?	`E F`	`True False`	

Note: The topmost data item is shown in **bold**.

Laboratory 5: In-lab Exercise 1

Name _____ Date _____

Section _____

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 in which 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 on one line. Save your program in a file called *postfix.cpp*.

Step 2: Complete the following test plan 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.

Test Plan for the Postfix Arithmetic Expression Evaluation Program

Test Case	*Arithmetic Expression*	*Expected Result*	*Checked*
One operator	`34+`		
Nested operators	`34+52/*`		
Uneven nesting	`93*2+1-`		
All operators at end	`4675-+*`		
Zero dividend	`02/`		
Single-digit number	`7`		