

## Chapter 16

# Working with Data Structures

---

- Linked Lists
- Adding a Node to the End of a Linked List
- Adding a Node to Maintain Sorted Order
- Removing a Node from a Linked List
- Linked List Operators
- More on Linked List Operators
- Stacks
- Implementing a Stack as an Array
- Implementing a Stack as a Linked List
- Using a Linked List Stack
- Queues
- Implementing a Queue as an Array
- Implementing a Queue as a Linked List
- Using a Linked List Queue

## Linked Lists

Various techniques that can be used to group data are presented in this book. One of the more useful methods is to assign data to elements of an array. Each array element can then be referenced. However, there is a serious limitation to using an array in that the maximum number of elements must be declared when the program is written.

For example, suppose an array of 10 elements is declared in the program. Later, the program requires the array have 11 elements. There is no easy or efficient way to affect such a change while the program is running. However, a different data structure called a *linked list* overcomes this problem.

A linked list is a self-referential structure. Each element of the list points to the next element in the list. An element in the list composed of data and a reference to another list element is often called a *node*. The last node in the list points (has a reference) to a NULL value indicating that the list has ended. A linked list allows elements to be added or removed without impacting the location of other elements.

The program example and figures on the opposite page demonstrate some basic linked list concepts and implementation. At the top of the page is the content of a header file, **Node.h**, which is used to define the **Node** class. In future examples this class will be used by the **LList** class, thus the friend statement at the top of the definition. As a friend, **LList** will have access to the private data members of **Node**. The **Node** class has one constructor, used when allocating a new **Node**, and a destructor, called when a **Node** is deleted. The constructor provides a default value should none be provided. However, if the data item to be stored is passed the constructor dynamically allocates sufficient room for the data and copies (using the **strcpy( )** function) the data to the allocated location. The **Link** field is used to reference the next element of the list. If an address is not passed in, a NULL value (indicating the end of the list) is assigned. The **Node** destructor deletes the dynamic memory that was allocated to store the **Node**'s data.

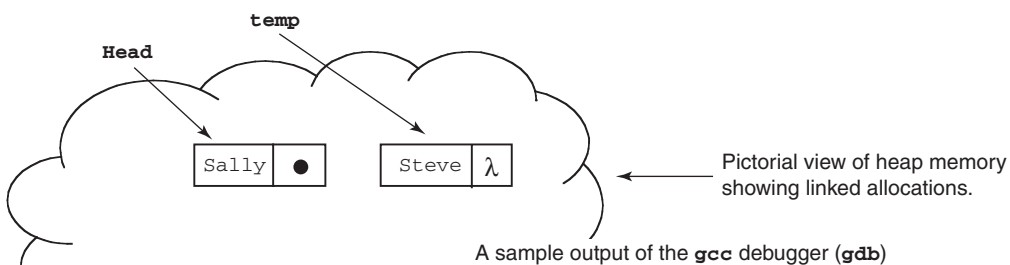
The sample program file (in the second box) makes use of the **Node** class and thus includes the file **Node.h**. A pointer to a **Node** called **Head** is declared. This will eventually be assigned the address of the starting node in the list. A new **Node**, referenced by the variable **temp**, is allocated and passed the data value "Steven" and a **Link** address of NULL. Following this, a second **Node** is allocated. This node is passed the data value "Sally" and the address of the previous node (with the data value "Steven"). The address of this new **Node** is assigned to the variable **Head**.

A pictorial representation of the allocated **Nodes** in heap memory is shown in the figure at the bottom left on the next page. The location of the actual nodes is system-dependent. The variables **Head** and **temp** are outside heap memory (in the program's data space) but they reference addresses in heap memory. Finally, if the program is compiled and run in a debugger (such as gcc's dbx debugger), the arrangements of elements in the list can be easily displayed. This is shown in the figure at the bottom right of the next page.

A linked list is a collection of elements that references one another by using a pointer that contains the address of the next element in the list. A linked list is used to organize data dynamically. The maximum size of a linked list is determined by the availability of free memory.

<pre>#include &lt;string.h&gt;  class Node {     friend class LList; public:     Node( char *s = "", Node *ptr = NULL ){         Item = new char [strlen(s)+1];         strcpy( Item, s );         Link = ptr;     };     ~Node(){ delete Item; }; private:     char *Item;     Node *Link; };</pre>	<p>A header file called <b>Node.h</b> that defines a node in the list.</p> <p>The class <b>LList</b> that manages nodes is declared a <b>friend</b>.</p> <p><b>Node</b> constructor allocates space to store the node's data, copies the data to the space, and assigns its <b>Link</b> field.</p> <p><b>Node</b> destructor deletes the memory allocated for the data.</p> <p>The data (<b>Item</b>) and <b>Link</b> for the node are private.</p>
--	---

<pre>#include "Node.h" int main(){     Node *Head;      Node *temp = new Node("Steven");      Head = new Node("Sally" , temp);     return 0; }</pre>	<p>This program makes use of the <b>Node</b> defined in the <b>Node.h</b> file.</p> <p>The pointer <b>Head</b> will reference the first node in the list.</p> <p>Allocate a <b>Node</b> and have <b>temp</b> reference the location.</p> <p>Allocate another <b>Node</b> and have it reference the previous node. Assign <b>Head</b> the location of this node.</p>
--	---



```
(gdb) print *Head
$1 = {Item = 0x27f18 "Sally", Link = 0x27ee8}

(gdb) print *(Head->Link)
$2 = {Item = 0x27ef8 "Steven", Link = 0x0}
```

## Adding a Node to the End of a Linked List

A variety of operations can be performed on a linked list. Good object-oriented design maintains these operations be grouped into a class. In the example on the opposite page, an **LList** class is declared for this purpose. The class declaration and the definition of its members are placed in the files **LList1.h** and **LList1.cxx**. Notice that as the **LList** class is a friend and makes use of the **Node** class, the file "**Node.h**" is included at the top of the declaration.

The initial pass at the **LList** public interface declares five methods and includes a single data member, a pointer (**Head**), which will be used to reference the first element of the linked list. The constructor, called when we instantiate a new **LList** object, sets the **Head** pointer to NULL. The destructor calls the **Empty( )** method that uses a **while** loop to step through all elements of the list, deleting each as it goes. Note the sequence of statements in the **while** loop is critical. It is important not to use the object that the pointer **curr** references once this object has been deleted.

Two other methods are defined for the **LList** class. The **Append( )** method is used to add a node to the end of the linked list. The logic for this method is straightforward. There are only two conditions to be considered. If the list is empty (**Head** is set to NULL), then the **Head** pointer is assigned the address of the newly allocated node. If there are already elements in the linked list, a **while** loop is used to move to the end of the list. Once at the end of the list, the **Link** field of the last element is set to reference the newly allocated node. In either case, the newly allocated node is passed the data for its **Item** field and a NULL (to indicate it is the last element in the list).

A **Print( )** method is also defined to display the contents of the list. If the list is empty, an error message is displayed on standard error. However, if there are elements in the list they are displayed in somewhat pictorial format. A temporary pointer (called **curr**) is used to reference elements in the list.

In a separate file is a small program that exercises the **LList** class. This is shown in the box on the right-hand side of the opposite page. The program includes the contents of the local **LList1.h** file, which has the **LList** class declaration. The first line of this program instantiates an **LList** object called **Roster**. This is followed by four calls to the **Append( )** method. Each time **Append( )** is called, a different name is passed as an argument. Each name is stored in a **Node** object. The **Node** objects are strung together using their **Link** fields. After all four names have been added to the list, the contents of the list are displayed with a call to the **Print( )** method. Note that **Print( )**, which does not modify the linked list, is declared as a constant function.

Remember that the program on the opposite page is for demonstration only. Keep in mind that in a production environment you would most likely test each call to **new** to verify it was able to properly allocate the memory requested.

Numerous operations can be performed on a linked list. One of the most basic is to add a **Node** to the list. The new **Node** can be added to the front of the list, in a location that maintains the sequencing of the list (based on the list's search key (see the next section), or at the end of the list. This example demonstrates how to add a **Node** to the end of the list (i.e., append).

```
#include "Node.h"
#include <iostream.h>
class LList {
public:
    LList ( ) : Head( NULL ) {}
    ~LList ( ) { Empty(); }
    void Append( char * );
    void Print( ) const;
    void Empty( );
private:
    Node *Head;
};
```

**LList** is a friend of the **Node** class and makes use of its public methods and private data.

There are five **LList** methods: a constructor, destructor, **Append( )**, **Print( )**, and **Empty( )**.

**Head** will reference the first element of the linked list.

```
void
LList::Empty ( ){
    Node *curr = Head, *next;
    while ( curr ){
        next = curr->Link;
        delete curr;
        curr = next;
    }
    Head = NULL;
}
void
LList::Append( char *ptr ){
    if ( !Head )
        Head = new Node( ptr, NULL );
    else {
        Node *curr = Head;
        while( curr->Link )
            curr = curr->Link;
        curr->Link = new Node( ptr, NULL );
    }
}
void
LList::Print( ) const {
    if ( Head ) {
        Node *curr = Head;
        while ( curr ){
            cout << "[" << curr->Item << "];";
            curr = curr->Link;
            cout << (curr ? "->" : "\n");
        }
    } else
        cout << "Empty list\n";
}
```

**Empty** deletes each **Node** in the list.

**Append( )** adds a node to the end of the list. If the list is empty, **Head** references the new node; otherwise, the **Link** field of the last item in the list references the new node.

```
#include <iostream.h>
#include "LList1.h"

int
main( ){
    LList Roster;
    Roster.Append("Samantha");
    Roster.Append("Jillian");
    Roster.Append("Wilma");
    Roster.Append("Sigmund");

    Roster.Print( );

    return 0;
}
```

Add (append) four names to the list and then display.

**Sample Output**

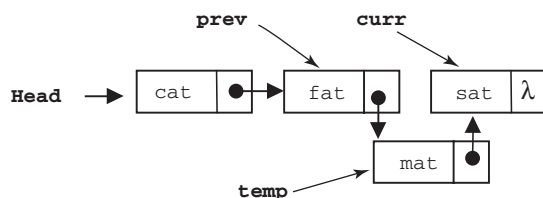
```
[Samantha]->[Jillian]->[Wilma]->[Sigmund]
```

## Adding a Node to Maintain Sorted Order

When adding a node to a linked list, we often want to maintain the order of the list. If the list is organized using a search key value rather than placing the new node at the end of the list, the new node must be inserted into the appropriate location in the list. To accomplish this the list is searched while the search key for the new node is less than the search key of the current node or the end of the list is encountered. By keeping track of the current and previous node locations, the newly instantiated node can be inserted at the correct location in the list. For example, say we have the linked list shown next and wish to add a node that contains the search key “mat”.



As we move along the list when the current node being examined is “sat”, the proper location for insertion is found. The new node should reference the node with the “sat” search key and the previous node, with the search key “fat”, should reference the new node. Pictorially this would be:



The example on the opposite page provides an implementation of an `Insert()` method for the `LList` class detailed in the “Linked List” section of this chapter. The program includes the contents of the local `LList2.h` file, which has the `LList` class declaration. The `Insert()` method begins by allocating a new `Node` object that is passed the search key value and a `NULL` pointer. The new node is referenced by the `temp Node` pointer variable. Two `Node` pointers, `curr` and `prev`, are assigned the address of the first node in the list and a `NULL` value, respectively. The list is then searched using a `while` loop. As long as the variable `curr` references a valid `Node` and the new search key to be added is less than the current key, the pair of pointers (`prev` and `curr`) are updated. When the loop exits, there are two conditions to be considered. The first condition to consider is if the new node should be placed at the front of the list. If this is the case, the `Link` field of the new node should reference the node currently at the head of the list, and the variable `Head` should be updated to reference the newly allocated node. The second condition is if the new node is to be placed in the middle (or at the end) of the list. In this situation, the `Link` field of the node referenced by the `prev` pointer is set to reference the new node and the `Link` field of the new node is set to reference the node

referenced by the **curr** pointer. In either case, the order of the statements to be performed is important. As a general rule of thumb, when adding a new node to a linked list, have the new node reference the proper location in the list and then have the correct part of the linked list reference the new node. If this sequence is done incorrectly, a portion of the list may be lost.

Notice, as shown, there is no check for duplicate nodes and dynamic allocations are not tested to determine if an error was encountered. For brevity, the code for previously defined methods is not shown in the example.

If we want to maintain a linked list that is in sorted order (based on its search key values), a method that inserts the node to the appropriate position in the list must be defined.

```
#include "Node.h"
#include <iostream.h>
class LList {
public:
    LList ( ) : Head( NULL ) {}
    ~LList ( ) { Empty( ); }
    void Append( char * );
    void Insert( char * );
    void Print( ) const ;
    void Empty( );
private:
    Node *Head;
};
```

An **Insert( )** method is added to the **LList** class declaration.

```
#include <iostream.h>
#include "LList2.h"

int
main( ){
    LList Roster;

    Roster.Insert("fat");
    Roster.Insert("cat");
    Roster.Insert("sat");
    Roster.Print( );

    Roster.Insert("mat");
    Roster.Print( );
    return 0;
}
```

```
void
LList::Empty( ){
}
void
LList::Append( char *ptr ){
}
void
LList::Insert( char *ptr ){
    Node *temp = new Node(ptr, NULL),
        *curr = Head, *prev = NULL;

    while( curr && strcmp(ptr, curr->Item) > 0){
        prev = curr;
        curr = curr->Link;
    }
    if ( curr == Head ){
        temp->Link = Head;
        Head = temp;
    } else {
        temp->Link = prev->Link;
        prev->Link = temp;
    }
}
void
LList::Print( ) const {
}
```

Defined previously.

Instantiate a new **Node** and pass it the search key value and a **NULL** value.

Step through the list while the search key of the new **Node** is less than the current search key or the end of the list is encountered.

Insert the node at either the front, middle, or end of the list. Always have the new **Node** reference the existing list before the list references the new **Node**.

```
void
LList::Print( ) const {
}
```

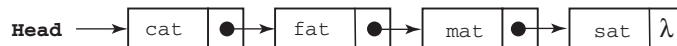
Defined previously.

**Sample Output**

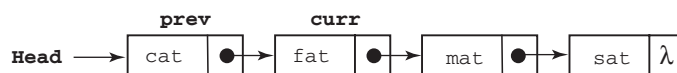
```
[cat]->[fat]->[sat]
[cat]->[fat]->[mat]->[sat]
```

## Removing a Node from a Linked List

Nodes that have been added to a linked list can be removed. To remove a node, the linked list is searched for the node (or nodes if duplicates are allowed) with the specified search key. When a node to be deleted is encountered, the **Link** fields of the elements involved are adjusted. If the node is at the front of the list, the pointer to the first element of the list must also be updated. Assume the linked list consists of the following nodes



and the node with the search key “fat” is to be removed. The list is traversed using two pointer variables to keep track of the position in the list. One pointer references the current node and a second pointer references the node before the current node. When the node with the search key “fat” is located, the pointers should be as shown:



To remove the node, the **Link** field in the previous node should be reassigned the **Link** field value of the current node. If, per chance, the node to be deleted was the first in the list, then the address stored in the **Head** pointer should be updated to the **Link** field value of the current node. This reassignment causes the list to point “around” the node to delete.

The example on the opposite page defines a **Remove( )** method for the **LList** class. The program includes the contents of the local **LList3.h** file, which has the **LList** class declaration. As written, the method will remove multiple occurrences (duplicates). If the list is not empty, it is searched for the node(s) to be removed. Two **Node** pointer variables, **curr** (initially assigned the starting address of the linked list) and **prev** (initialized as NULL), are declared. These variables keep track of the address of the current and previous node. A third pointer variable, **zombie**, is also declared. This variable is used to temporarily store a reference to the node to be deleted. A **while** loop is used to process the list. If a node with the search key is encountered, the variable **zombie** is assigned the node’s address. The position of the node in the list (either at the front or at the middle/end of the list) is determined and the appropriate pointers are updated. Notice the pointer **prev** is only updated when a node that is *not* to be deleted is encountered. In either case, the **curr** pointer is updated to the next element (node) in the list and the value in **zombie** is checked. If it is non-NULL (indicating a node to be deleted), a **delete** is issued for the object it references.

A small program to show how the **Remove( )** method works is provided on the next page. A linked list with the search key values of “fat”, “cat”, “mat”, and “sat” is established. As the **Insert( )** method is used, the list is in ascending alphabetic order. The list is displayed and the node with the key value “fat” is removed. The

list is then redisplayed and the node with “cat” (which is at the front of the list) is removed. The list is displayed one last time to verify its contents.

Nodes that are added to a linked list can be removed. Once the node is located, the appropriate references must be reassigned to point “around” the node to be removed. The removed node is deleted (the memory used by the node is returned to the system). If the list permits duplicates, then all occurrences of the node should be removed.

```
#include "Node.h"
#include <iostream.h>
class LList {
public:
    LList ( ) : Head( NULL ) {}
    ~LList ( ) { Empty(); }
    void Append( char * );
    void Insert( char * );
    void Remove( char * );
    void Print ( ) const ;
    void Empty();
private:
    Node *Head;
};
```

A `Remove ( )` method is added to the `LList` class declaration.

```
void
LList::Empty() {
}
void
LList::Append( char *ptr ) {
}
void
LList::Insert( char *ptr ) {
}
void
LList::Remove( char *ptr ) {
    if ( !Head ) {
        cerr << "Empty list\n";
        return;
    }
    Node *curr = Head, *prev = NULL, *zombie;
    while ( curr ) {
        zombie = NULL;
        if ( strcmp( curr->Item, ptr ) == 0 ) {
            zombie = curr;
            if ( curr == Head )
                Head = curr->Link;
            else
                prev->Link = curr->Link;
        } else
            prev = curr;
        curr = curr->Link;
        if ( zombie ) delete zombie;
    }
}
void
LList::Print( ) const {
}
```

Defined previously.

Search the list for all occurrences of the data. Use three pointers: `curr` references the current node, `prev` references the previous node, and `zombie` references the node for deletion.

If a node is to be deleted and it is at the head of list, it must be treated differently than if it is at the middle or end of the list.

Only update `prev` if this is not a node to be deleted.

If a node is marked for deletion, delete it.

```
#include <iostream.h>
#include "LList3.h"
int
main() {
    LList Roster;

    Roster.Insert("fat");
    Roster.Insert("cat");
    Roster.Insert("sat");
    Roster.Insert("mat");
    Roster.Print();

    Roster.Remove("fat");
    Roster.Print();

    Roster.Remove("cat");
    Roster.Print();

    return 0;
}
```

Sample Output

```
[cat]->[fat]->[mat]->[sat]
[cat]->[mat]->[sat]
[mat]->[sat]
```

# Linked List Operators

The list of operations that can be performed on a linked list is only limited by the needs of the user and the ability of the programmer to implement the operations as methods. In some settings, operators (as well as methods) are defined. To this end, the implementation of two operators, assignment (assigning one existing list into another) and indexing, are shown in the sample program on the opposite page.

## Assignment

Occasionally a user may need to assign a copy of one list into another existing list. A natural way to implement this is to overload the assignment operator (=). As with a standard assignment, the previous contents of the list on the left-hand side of the assignment are overwritten or replaced by the contents of the list on the right-hand side of the assignment.

The prototype for overloading the assignment operator to work with our linked list data structure is:

```
const LList & operator = ( const LList & );
```

The declaration indicates there is a return from the assignment operation. The return type is a constant reference to a linked list. A reference type was used, as value return type would have necessitated the generation of an additional copy of the list.

In the corresponding function (method) definition, an **if** statement is used to determine whether there is an attempt to assign a list into itself. Should the list on both sides of the assignment be the same, no further action is needed. If they are different, the list on the left-hand side of the assignment is emptied with a call to the **Empty( )** method. This statement will invoke the class destructor for the **LList** object. The contents of the list on the right-hand side of the assignment are appended (using a **for** loop) to the truncated list. A reference to the current list, using the **this** pointer, is returned.

## Indexing

The [ ] operator, which is normally used to index arrays, can be overloaded to return a member of the linked list at a specific offset in the list. In the class declaration on the opposite page, the statement

```
char * operator[ ] ( int ) const;
```

indicates the index operator is passed an integer value (remember, underneath C++ views operators the same as functions, thus the parenthesis notation). A reference to the search key data member at the requested offset is returned.

In the associated function definition, a **for** loop steps through the appropriate number of list elements. Once the loop is exited, a conditional is used to determine if the requested element is in the list. If the element is not present, a NULL value is returned in place of the search key data. As written, the first element of the list is at 1 (not 0 as in C and C++ arrays).

Operators may also be defined to work on user-defined data structures. The overloaded operators should function in a manner somewhat similar to their default counterpart.

```

#include "Node.h"
#include <iostream.h>
class LList {
public:
    LList ( ) : Head( NULL ) {}
    ~LList ( ) { Empty ( ); }
    const LList & operator = ( const LList & );
    char * operator [ ] ( int ) const;
    void Append ( char * );
    void Insert ( char * );
    void Remove ( char * );
    void Print ( ) const;
    void Empty ( );
private:
    LList ( const LList & ) {}
    Node *Head;
};

```

Overload the assignment operator. This operator is passed a reference to a linked list and returns a reference to a linked list.

Overload the index operator to return a reference to the search key data value at a specific location in the list.

Generate a dummy (empty) copy constructor to prevent the default copy constructor (which only does a *shallow* copy) from being called.

```

void
LList::Empty ( ) { ← Defined previously.
}
const LList &
LList::operator = ( const LList &rhs ) {
    if ( this != &rhs ) {
        Empty ( );
        for ( Node *curr=rhs.Head; curr ; curr=curr->Link)
            Append ( curr->Item );
    }
    return *this;
}
char *
LList::operator [ ] (int n) const {
    if ( Head ) {
        Node *temp = Head;
        for ( ; temp && n-1; --n )
            temp = temp->Link;
        return temp ? temp->Item : NULL;
    }
    return NULL;
}
void
LList::Append ( char *ptr ) {
}
void
LList::Insert ( char *ptr ) { ← Defined previously.
}
void
LList::Remove ( char *ptr ) { ← Defined previously.
}
void
LList::Print ( ) const { ← Defined previously.
}

```

If the two lists are not the same, begin by deleting the list on the left-hand side. Next, append the contents of the list on the right-hand side. Return a reference to the list.

If there are elements in the list (**Head** is not NULL), attempt to find the *n*th element (where the first element is at 1). If the specified element is present, return its data field; otherwise, return a NULL.

Test program example on next page.

## More on Linked List Operators

The program on the opposite page shows the use of the two linked list operators defined on the previous set of pages.

At the top of the program, the “**LList4.h**” file is included. This file contains the current class declaration and definitions. As noted before, the class declaration and definitions would normally be in separate files, but here they have been combined to simplify the example.

Beneath the include statements is an overloaded insertion operator function. This function allows the programmer to use the insertion operator with a **LList** object. Within this function, the **Print ( )** member function is called.

Following this, a **LList** object, called **A**, is instantiated. The **Insert ( )** method of the **LList** class is used to add three nodes to the list. By definition, **Insert ( )** adds the nodes in alphabetic order based on the passed search key value (i.e., “fat”, “cat”, and “sat”). The contents of the list are then displayed. A second **LList** object, called **B**, is allocated. One element, “mat”, is appended to this list.

Next, the contents of **A** are assigned to **B**. When this is done, the previous contents of list **B** are deleted and a copy of the contents of list **A** are appended to the empty list **B**.<sup>1</sup> When the two lists are printed a second time, we can see that both lists are the same (and “mat” has been deleted).

Two more linked lists are instantiated. Once generated, an assignment statement is used to assign the contents of list **B** into list **D**. The returned result of the assignment, a constant reference to a **LList** object, is then assigned to list **C** (as if the assignment was nested as **(C = (D = B))**). Further removals and additions are made to the lists to make their content unique. A display of all four lists confirms each is separate, and the modifications to one list do not impact the contents of the others.

The index operator is used to return the search key data at the specified location in the list. As written, the first element (node) is at 1, the second is at 2, and so forth. If we were to specify an element that was not in the list, such as the fifth element of list **A**, a NULL value would be returned. Some compilers will handle the attempt to display a NULL value and print something along the lines of (**nil**). Other compilers will choke when asked to do this and the executing program will crash. It is usually best to check what is returned before attempting the display.

---

<sup>1</sup>If we think about this some more, we realize that it might be better to copy the data from list **A** to **B**, adding or deleting nodes from **B** should **A** not have the same number of elements as **B**.

```

#include <iostream.h>
#include "LList4.h"

ostream &
operator << ( ostream &os, const LList &lst ){
    lst.Print( );
    return os;
}

int
main( ){
    LList A;

    A.Insert("fat");
    A.Insert("cat");
    A.Insert("sat");
    cout << "\nList A: " << A;

    LList B;
    B.Append("mat");
    B = A ;
    cout << "\nList A: " << A;
    cout << "List B: " << B;

    LList C, D;
    C = D = B;

    A.Remove("cat");
    B.Insert("bat");
    C.Remove("fat");
    D.Insert("zat");

    cout << "\nList A: " << A;
    cout << "List B: " << B;
    cout << "List C: " << C;
    cout << "List D: " << D;
    cout << "\nItem 1 in list A " << A[1] << endl;
    cout << "Item 1 in list B " << B[1] << endl;
    cout << "Item 1 in list C " << C[1] << endl;
    cout << "Item 1 in list D " << D[1] << endl;

    return 0;
}

```

Overload the insertion operator to allow a more C++-like syntax for output of the list.

Instantiate a linked list object and add three elements. As `Insert( )` is called, the list will be in ascending alphabetic order.

Create a second linked list object and append one element with the search key "mat".

Assign list A into list B. All elements currently in B will be deleted (such as "mat").

Instantiate two more lists, C and D. Assign a copy of list B into D and then assign the results of this into C. This effectively sets all three lists to be the same.

Modify the contents of each list. Changes in one should not impact the others.

After additions and removals, all four linked lists have slightly different contents.

The first element of each list.

#### Sample Output

List A: [cat]->[fat]->[sat]

List A: [cat]->[fat]->[sat]  
List B: [cat]->[fat]->[sat]

List A: [fat]->[sat]  
List B: [bat]->[cat]->[fat]->[sat]  
List C: [cat]->[sat]  
List D: [cat]->[fat]->[sat]->[zat]

Item 1 in list A fat  
Item 1 in list B bat  
Item 1 in list C cat  
Item 1 in list D cat

# Stacks

A stack is a restricted list in which a node (element) can only be added to or deleted from the top (front) of the list. This restriction is referred to as a last-in, first-out (LIFO) arrangement. The last node that is added to the list is always at the top and thus is always the first node to be removed from the stack.

Stacks maintain the order of sequence. For example, an instruction stack is used to assure that each instruction is performed in the proper order. The next instruction is always at the top of the stack. Stacks can also be used in expression evaluation and parsing programs used to check for the balance of paired symbols such as parentheses. When function calls are made by a program, the contents of local variables, the location in the program where the call was made, and so forth, are saved on a stack.

Programs that manipulate a stack must have a way of adding a node to the top of the stack and removing the top node of the stack. The addition of a node to the top of the stack is most commonly called a *push*. Operation of removing the top (first) node of the stack is most often called a *pop*. Another common stack operation is a check to determine if a stack is empty (i.e., contains no more nodes or elements).

A stack data structure, in the form of a C++ class, can be implemented in a number of ways. The two most prevalent approaches are to employ a dynamic array for the stack or to use a linked list. Each approach will be covered in detail over the next few pages. However, before we discuss the implementation specifics, let's assume that for the moment we have a stack class already available.

The program on the opposite page uses a stack to demonstrate how to perform a simple evaluation of an expression to determine if the parentheses in the expression are balanced. The program uses a character array (**e**) to store the expression to be evaluated. A stack (**AStack**) object called **s** is allocated to hold processing information.

The user is prompted to enter an expression with pairs of parentheses. To avoid problems with whitespace interpretation, the `getline( )` function is used to retrieve the expression. Once obtained, the expression is evaluated one character at a time. If the character is a left parenthesis '(', the value of the current index into the expression (+1) is converted to a string using the `sprintf( )` function. With `sprintf( )`, the integer expression is "printed" into the character array **temp** using the "%d" (decimal) format specifier.<sup>2</sup> The character representation of the evaluated expression is *pushed* onto the stack. If a right parenthesis ')' is encountered, the stack is checked for the location of its matching left parenthesis. If the stack is not empty, the location of the matching pair is *popped* off the stack. However, if the stack is empty then a match was not found. Once the entire expression has been parsed, if the stack is not empty there are remaining left parentheses that do not have a matching right parenthesis.

The sample output at the bottom of the next page provides a series of input sequences and the output displayed by the program.

---

<sup>2</sup>This technique is a common C programming shortcut for converting a numeric value to a character string.

A stack is a type of list that allows nodes to be inserted and deleted from the top (front) of the list. Two functions commonly used for manipulating the stack are **Push( )** and **Pop( )**. The **Push( )** function adds a node to the top the stack and **Pop( )** deletes the top node.

<pre> #include &lt;iostream.h&gt; #include &lt;stdio.h&gt; #include "AStack.h" int main( void ) {     char e[80], temp[10];     AStack S;     cout &lt;&lt; "Please enter in an expression with ( )'s: ";     cin.getline( e, 80 );      for (int i=0; i &lt; strlen(e); ++i){         switch( e[i] ) {             case '(':                 sprintf(temp, "%d", i+1 );                 S.Push( temp );                 break;             case ')':                 if ( S.Stack_Empty( ) )                     cout &lt;&lt; "Unmatched ) at " &lt;&lt; i+1 &lt;&lt; endl;                 else                     cout &lt;&lt; "Matched pair at " &lt;&lt; S.Pop( )                         &lt;&lt; " " &lt;&lt; i+1 &lt;&lt; endl;                 break;         }     }     while( !S.Stack_Empty( ) )         cout &lt;&lt; "Unmatched ( at " &lt;&lt; S.Pop( ) &lt;&lt; endl;     return 0; } </pre>	<p>The stack class declaration is in the include file <b>AStack.h</b> (see the next section for details).</p> <p>The expression to be evaluated is stored in the character array <b>e</b>. The <b>temp</b> array will hold a character representation of a numeric value.</p> <p>Check each character in the expression.</p> <p>If a left parenthesis is encountered, convert the current offset in the expression (+1) into a character string. Push this onto the stack.</p> <p>If a right parenthesis is encountered and the stack is not empty, pop the location of the matching parenthesis.</p> <p>If the stack is not empty, some unmatched left parentheses are present.</p>
--	--

#### Sample Output

```

Please enter in an expression with ( )'s: (((a-b)-c) * e ))
Matched pair at 3 7
Matched pair at 2 10
Matched pair at 1 16
Unmatched ) at 17

Please enter in an expression with ( )'s: (a-b)-c)*c)
Matched pair at 1 5
Unmatched ) at 8
Unmatched ) at 11

Please enter in an expression with ( )'s: (((a-b)-c * e
Matched pair at 3 7
Unmatched ( at 2
Unmatched ( at 1

```

## Implementing a Stack as an Array

One technique for implementing a stack class is to use a dynamically allocated array. The class declaration and definition on the opposite page show how this is done.

The class constructor can be passed an integer value that sets the maximum size of the stack. If no value is passed, a default value of 10 is used. Notice the way the enumerated type **Default\_Size** is slipped into the class declaration as a defined constant. In the constructor, the integer identifier **Top** is set to 0. This identifier will contain the index of the current top value in the stack. The variable **Limit** is also assigned the upper bound (index) of the stack. When implementing a stack using an array we must keep track of the upper bound of the array so that we do not attempt to push more items onto the stack than it can hold. The last statement in the constructor dynamically allocates an array of pointers to the data type character. The address of this array, which will reference the stack elements, is assigned to the pointer variable **sp**.

The class destructor steps through the array of pointers deleting each element referenced by the array. Once this has been accomplished, the array itself is passed to **delete** for removal.

The **Push( )** method adds an element to the top of the stack. It first checks to see if the stack is full. If it is, then an error message is displayed to standard error. If the stack is not full, space is allocated for the next item in the stack. The **new** operator is used to allocate memory for the item. The address of the allocated space is assigned to the stack array using the value in **Top** as an index into the array of pointers. The **strcpy( )** function is used to copy the passed item to the allocated space. The value in **Top** is then incremented to reference the location to store the next address of the next element of the stack.

The **Pop( )** method removes the top element from the stack. The stack is checked to determine if it is empty. If empty, a NULL value is returned to the calling statement. If not empty, the value in **Top** is decremented to reference the element at the top of the list. The element at the top of the list is copied to a **static**, temporary character variable so that it can be returned to the calling statement. The dynamic memory used to hold the element is returned to the system with a call to **delete**. The reference to the temporary variable is returned.

There are three other methods in this implementation of a stack. The inline methods **Stack\_Empty( )** and **Stack\_Full( )** are boolean functions that test whether or not the stack is empty or full. A **Print( )** method is also defined to display the current contents of the stack in a pictorial format.

A program that makes use of the **AStack** class can be found in the previous example.

```

#include <iostream.h>
#include <string.h>
class AStack {
public:
    AStack(int n = Default_SIZE);
    ~AStack( );
    void Push( char * );
    char *Pop ( );
    bool Stack_Empty( ) const { return bool(Top-1 < 0); }
    bool Stack_Full ( ) const { return bool(Top+1 >
Limit); }
    void Print( ) const ;
private:
    enum { Default_SIZE=10 };
    int Top,
        Limit;
    char **sp;
};

```

If the size of the stack is not passed, use the default size (set at 10).

Two inline boolean functions to test if the stack is empty or full.

Declare an integer constant using **enum**. *Note:* some compilers may need to have this declared before it is used in the constructor.

```

AStack::AStack( int size_passed ){
    Top = 0;
    Limit = size_passed;
    sp = new char * [Limit];
}
AStack::~AStack( ){
    while (!Stack_Empty( ))
        Pop( );
    delete [] sp;
}
void
AStack::Push( char *ptr ){
    if ( !Stack_Full( ) ) {
        sp[Top] = new char [strlen(ptr)+1];
        strcpy(sp[Top], ptr);
        ++Top;
    } else cerr << "Stack full.\n";
}
char *
AStack::Pop( ) {
    static char tmp[512];
    if ( !Stack_Empty( ) ) {
        Top--;
        strcpy(tmp, sp[Top]);
        delete sp[Top];
        return tmp;
    } else return (char *) NULL;
}
void
AStack::Print( ) const {
    if ( !Stack_Empty( ) ){
        cout << "Top -> ";
        for(int i=Top-1; i >= 0 ; --i)
            cout << "\t|" << sp[i] << "\n";
        cout << "\t+----\n";
    } else cerr << "Stack empty.\n";
}

```

When instantiated, set **Top** to zero, set the upper bound, and allocate the array of pointers that will act as the stack.

Remove each node in the stack and then the array of node pointers.

If the stack is not full, allocate space for the item passed in (**ptr**). Copy the item to the allocated space and increment **Top** to the next location.

If the stack is not empty, decrement **Top**. Copy the top item to a temporary location. Delete previously allocated memory. Return the copy of the top item to the calling statement.

## Implementing a Stack as a Linked List

More commonly, a linked list is used when implementing a stack class. With a linked list, the restriction of having to know in advance the maximum size of the stack is removed.<sup>3</sup> The example on the next page provides a stack class called **LStack** that is linked list-based. This implementation uses the **Node** class, described earlier in this chapter, to hold the data for each node. *Note:* if the **LStack** class is to use the **Node** class, the statement

```
friend class LStack;
```

must be placed in the **Node** class declaration.

The **LStack** class constructor sets **Head**, the pointer to the first node in the list, to NULL. The class destructor walks through the linked list representing the stack deleting each node in the list (which returns the heap memory it used to store the data item). Once the list is removed, **Head** is reset to NULL.

The **Push( )** method using a linked list implementation is much briefer than its array-based counterpart. Only one statement is needed:

```
Head = new Node( ptr, Head );
```

It consists of a call to instantiate a new **Node** object. Two arguments are passed. The first is a pointer to the data item and the second is a pointer to the next node in the list. As the call to **Node** constructor is done before the assignment, **Head** (at this point of construction) references either the NULL (the list was empty) or the existing linked list. Once the new node is instantiated, its address is assigned to **Head**, updating it to reference the newly allocated node.

The **Pop( )** member function removes a node from the list. Notice this time the check for an empty stack is a check to see if **Head** is NULL versus checking a value that acts as an index into an array. If the list is not empty, a temporary location is allocated and the top item of the stack is copied to this temporary location. As with the array implementation, this location is flagged as **static**. A temporary pointer, **zombie**, is used to hold the reference to the node that will be deleted. The **Head** pointer is updated to reference the next item in the list. A call to **delete** removes the node that was storing the data. The data in the temporary storage location is returned to the calling statement.

A **Print( )** method is also included in the class definition. It is similar in form and function to the **Print( )** method used in the **AStack** class.

A program that uses the **LStack** class can be found on the next set of pages.

---

<sup>3</sup>Technically, this restriction is not entirely removed as we can run out of memory if we attempt to add too many items to the stack and all of the heap memory is consumed.

```
#include <iostream.h>
#include <string.h>
#include "Node.h"
```

Each element in the linked list will be a node. The header file for **Node** must be included.

```
class LStack {
public:
    LStack ( ) : Head( NULL ) {}
    ~LStack( );
    void Push( char * );
    char *Pop( );
    bool Stack_Empty( ) const { return Head == NULL; }
    void Print( ) const ;
private:
    Node *Head;
};
```

When instantiating a **LStack** object, set **Head** to **NULL**.

Determining if the stack is empty checks if **Head** is set to **NULL**.

```
LStack::~LStack( ){
    Node *curr = Head, *next;
    while ( curr ){
        next = curr->Link;
        delete curr;
        curr = next;
    }
    Head = NULL;
}
```

To push a node onto the stack, a new **Node** object must be created. The new **Node** is passed the data item and a reference to the rest of the list. The address of the new **Node** is assigned to **Head**.

```
void
LStack::Push( char *ptr ){
    Head = new Node( ptr, Head );
}
```

If the stack is empty, display an error message to standard error.

```
char *
LStack::Pop( ){
    if ( Stack_Empty( ) ) {
        cerr << "Empty stack\n";
        return NULL;
    }
    static char *temp = new char[strlen(Head->Item)+1];
    strcpy( temp, Head->Item);
    Node *zombie = Head;
    Head = Head->Link;
    delete zombie;
    return temp;
}
```

The data item in the top (first) node in the list is to be returned. A temporary location is allocated to store this data. A temporary pointer is used to hold the node to be deleted. Update the **Head** pointer, delete the node, and return the data value.

```
void
LStack::Print( ) const {
    if ( !Stack_Empty( ) ){
        cout << "Top -> ";
        for (Node *temp = Head; temp ; temp = temp->Link)
            cout << "\t|" << temp->Item << endl;
        cout << "\t+-----\n";
    } else
        cerr << "Stack empty.\n";
}
```

## Using a Linked List Stack

Stacks are often used to evaluate expressions. The example on the next page uses a stack (that is linked list-based) to convert a postfix expression into its infix counterpart, adding parentheses to maintain the implied order of operations.

Before going over the program, a brief overview of expressions is in order. Most of us are familiar with algebraic expressions where binary operators are found in between their operands (e.g.,  $A + B$ ). This notation is known as infix notation. If the operators in the expression have different precedence, parentheses may be needed to specify the desired order of evaluation. For example:  $A * (B - C) / D$  indicates the subtraction should be done first. A Polish mathematician invented an alternate way of writing this expression whereby parentheses are not needed to specify the order of evaluation. This notation, called Reverse Polish Notation (RPN), would restate the previous expression to be:  $A B C - * D /$ . Expressions in this notation are more easily evaluated as only a single pass from left to right is required. In short, as operands are encountered they are pushed onto a stack. When an operator is detected, the last two operands are popped off the stack, the operation is performed, and the result is pushed back on the stack. When popping operands off the stack, the first operand popped is the right-hand operand and the second is the left-hand operand.

The process of converting an RPN expression (also called a postfix expression) to its infix counterpart is carried out by the sample program on the right. The program instantiates a single stack object. As the stack is linked list-based, a size (maximum number of elements) for the stack is not specified. Four temporary character variables are declared. The variable `item` will hold the current character operator or operand. The two variables `left_part` and `right_part` will store the left and right operand and `temp` will hold the temporary subexpression to be pushed back on the stack. As the data items in the stack are character array-based, these variables are also declared as character arrays.

A `do...while` loop is used to collect user input (the postfix expression to be converted). As each item is input, it is first checked to see if it's the letter 'Q', indicating processing is completed. If not a 'Q', the input item is checked to determine if it's an operator. The `strchr( )` function, which requires the inclusion of `stdio.h`, is used for the check. This function expects two arguments: the address of a C-type string of characters and a single character to find within the string. If the character is present, its address is returned; otherwise, a NULL is returned. The value returned is treated as a boolean true (nonnull) or false (NULL). If the character is an operator, the two operands are popped off the stack. These values, along with the operator, are "printed" into the `temp` character array using the `sprintf( )` function. Parentheses are also included to maintain the order of operations specified by the RPN form. The subexpression is then pushed on the stack. If the input item is an operand (anything that is not an operator or the letter 'Q'), it is pushed on the stack. When the RPN expression is fully entered, the last item in the stack is the equivalent parenthesized infix expression.

```

#include <iostream.h>
#include <stdio.h>
#include "LStack.h"
const int max = 25;
int
main( ) {
    LStack S;
    char item[max], temp[max],
        right_part[max], left_part[max];
    do {
        cout << "> ";
        cin >> item;
        if ( item[0] == 'Q' )
            break;
        else if ( strchr("+-/*", item[0]) ) {
            strcpy(right_part, S.Pop( ));
            strcpy(left_part, S.Pop( ));
            sprintf(temp, "(%s %s %s)",
                left_part, item, right_part);
            S.Push( temp );
        } else
            S.Push( item );
        cout << "Stack contains \n";
        S.Print( );
    } while (true);
    return 0;
}

```

Needed for the `strcpy( )` and `sprintf( )` functions.

Linked list stack class.

If the user enters a 'Q', processing stops.

Check if an operator has been entered. If yes, pop the operands off the stack.

Build the subexpression, use parentheses to maintain proper order, and push subexpression on the stack.

#### Sample Output

```

> A
Stack contains
Top -> |A
      +-----

> B
Stack contains
Top -> |B
      |A
      +-----

> C
Stack contains
Top -> |C
      |B
      |A
      +-----

> -
Stack contains
Top -> |(B - C)
      |A
      +-----

> *
Stack contains
Top -> |(A * (B - C))
      +-----

> D
Stack contains
Top -> |D
      |(A * (B - C))
      +-----

> /
Stack contains
Top -> |((A * (B - C)) / D)
      +-----

> Q

```

At the point where an operator is entered, the top two operands are popped off the stack. The first is the right-hand operand and the second is the left-hand operand. The resulting subexpression (with parentheses) is pushed back on the stack.

When the entire RPN expression is entered, the last thing left in the stack is the fully parenthesized infix expression equivalent.

# Queues

A *queue* is another special type of list that restricts the addition and removal of nodes. All nodes enter (are added to) the queue at the end (back) of the queue and are deleted from the top (front) of the queue. This concept is commonly referred to as first-in, first-out (FIFO) list.

Queues are found in a variety of programming-based environments. An operating system maintains a number of queues. In particular, queues are used to manage the print requests and process scheduling. Further, system-implemented queues are used for message passing. By their very nature, queues (FIFO lists) ensure that items in the list do not wait forever for attention.

Functions are needed to add and remove nodes (elements) from a queue. The process of adding a node to the end of the queue is most often termed *enqueue* (or *enterqueue*) and removing a node from the front of the queue called *dequeue* (or *exitqueue*). Actually, any name can be used to identify these functions because they are user-defined functions.

A queue data structure, in the form a C++ class, can be implemented in different ways. As with stacks, the two most prevalent approaches are to employ a dynamic array for the queue or to use a linked list. Each approach will be covered in detail in the following pages. However, before we discuss the specifics of implementation, let's assume that for the moment we have a queue class already available.

The program on the opposite page uses a queue to demonstrate reading groups of characters into a small 10-element input queue. Once the queue is filled, it is dumped to standard output and a bar character '|' is printed to delineate the 10-character grouping.

At the top of the `main( )` function, a queue object (called `q`) is instantiated. A temporary two-character array (called `ch`) is declared. The character array is needed because the `Enqueue( )` method expects a pointer to a null terminated C-style string. To "prime the pump," an initial character is read from standard input into the `ch` variable at index 0. In a `while` loop, the input character is added to the queue, provided the queue is not full. The state of the queue is determined by the call to the boolean method `Queue_Full( )`. The addition is accomplished by calling the `Enqueue( )` method associated with the queue object. If the queue is filled, its contents are removed using a second `while` loop. This loop calls the `Dequeue( )` method of the queue object. Once the queue is emptied, a bar character '|' is used to separate the character grouping. At the foot of the outer `while` loop, the next standard input character is retrieved. Once the end of file is encountered, the primary processing loop terminates. As there may still be data in the queue, a third `while` loop is used to dump this remaining data to the screen.

The sample output figure at the bottom left of the opposite page illustrates the working of this program. A text file called `spider.txt` is redirected (using standard command line redirection) to the executable version of the program. As can be seen, the input file is redisplayed in its entirety and in proper order.

A queue is a list-based data structure where the nodes (elements) are removed from the list in the same order in which they were added. Adding a node to the queue is most often called enqueue (enterqueue) whereas removing a node is called deque (exitqueue).

```

#include <iostream.h>
#include "AQueue.h"

int
main( ){
    AQueue Q;
    static char ch[2];

    cin.get( ch[0] );

    while( !cin.eof( ) ) {
        if ( !Q.Queue_Full( ) )
            Q.Enqueue( ch );
        else {
            while( !Q.Queue_Empty( ) )
                cout << Q.Dequeue( );

            cout << "|";
            Q.Enqueue( ch );
        }
        cin.get( ch[0] );
    }

    while( !Q.Queue_Empty( ) )
        cout << Q.Dequeue( );
    return 0;
}

```

The queue class declaration and definition are in the include file **AQueue.h**.

Instantiate an **AQueue** object.

Obtain a character from standard input. This will allow for a valid test of end-of-file. **ch[1]** holds the terminating '\0'.

If the queue is not full, call the **Enqueue( )** method to add the current character.

While the queue is not empty, remove the nodes, displaying them as they are removed.

Add a bar character '|' to show groupings of 10 characters.

Be sure to **Enqueue( )** the character that was held when queue full was encountered.

If elements are still in the queue, remove them.

The file: **spider.txt**

### Sample Output

```

% a.out < spider.txt
The itsy b|itsy spide|r
went up |the water |spout.

Do|wn came th|e rain
and| washed th|e spider o|ut.

Out c|ame the su|n
and drie|d out all |the rain.
|
And the i|tsy bitsy |spider
wen|t up the s|pout again|.

```

The itsy bitsy spider  
went up the water spout.

Down came the rain  
and washed the spider out.

Out came the sun  
and dried out all the rain.

And the itsy bitsy spider  
went up the spout again.

The contents of the file are broken into groups of 10. Note that some of the characters are "non-printing."

The period and newline at the end of the last sentence are still in the queue when the end of file is encountered.

## Implementing a Queue as an Array

A dynamically allocated array can be used to implement a queue data structure. The **AQueue** class declaration and definition on the opposite page show how this is done. Keep in mind that as elements are added to the end of the list and removed from the front, a reference to both ends of the list is needed to simplify processing.

The **AQueue** constructor can be passed an integer value that specifies the size of the queue. If no value is passed then a queue of size 10 (the **Default\_SIZE** value) is allocated. The constructor initializes an integer **Head** and **Tail** “pointer” to zero. These two variables are used as an index into the array to keep track of the front and end of the list, respectively. An integer variable, **N\_Items**, is set to zero. The **N\_Items** variable will store the number of elements in the list. This will simplify the process of determining when the list is either empty or full. The maximum number of elements the queue can store is stored in the **Q\_Size** variable. The last statement in the constructor allocates an array of pointers to the data items that will be stored in the queue. This array is referenced by the pointer variable **qp**.

The **AQueue** class destructor steps through the array of pointers deleting each element referenced by the array. Once this has been accomplished, the array itself is passed to **delete** for removal. The bracket notation notifies **delete** an array is referenced.

The **Enque( )** method adds a node to the end of the list. To accomplish this, the number of items in the queue is checked. If the queue is full, an error message is displayed to standard error. If there is more room in the queue, space is dynamically allocated to store the new element. The address of the newly allocated space is stored in an element of the **qp** array. The specific element of this array is indexed by the variable **Tail**. The **strcpy( )** function is used to copy the data passed to the allocated space. The variable **N\_Items** is incremented to reflect the addition. The **Tail** variable is then assigned the next valid location. This statement bears some further discussion. The pre-increment of **Tail** causes the value in **Tail** to be incremented by one. The modulus operator ensures that once **Tail** reaches the last valid index in the array, it wraps around to the first (0) index in the array.

The **Deque( )** method removes one element from the queue. To remove an item, the queue is checked to determine if it is empty. If the queue is empty, a NULL value is returned. If the queue is not empty, the element at the front of the queue, indexed by **Head**, is copied (using **strcpy( )**) to a temporary location. The variable **N\_Items** is decremented to reflect the removal. A call to **delete** removes the dynamically allocated memory where the queue data resided. The value in **Head** is then updated. Notice, like **Tail**, the value is pre-incremented and a modulus is used to keep the value in the proper range. The data, now in the temporary location, is returned to the calling statement.

The program example on the previous page used an **AQueue** object.

```

#include <iostream.h>
#include <string.h>
class AQueue {
public:
    AQueue( int n = Default_SIZE );
    ~AQueue( );
    void Enque( char * );
    char *Deque( );
    bool Queue_Empty( ) const { return N_Items <= 0; }
    bool Queue_Full( ) const { return N_Items >= Q_Size; }
    void Print( ) const;
private:
    enum { Default_SIZE=10 };
    int Head, Tail, N_Items, Q_Size;
    char **qp;
};

```

If the size of the queue is not passed, use the default size (set at 10).

Two inline boolean functions test if the queue is empty or full.

A **Head** and **Tail** variable are used to reference the front and end of the list.

When instantiated, **Top**, **Tail**, and **N\_Items** are set to zero. Set the upper bound for the queue and allocate the array of pointers that will act as the queue.

Remove each node in the queue and then the array of node pointers.

If queue is not full, allocate space for the item passed in (**ptr**). Copy the item to the allocated space and increment **Tail** to the next location. Use the modulus operator to treat the array as a circular list.

If the queue is not empty, copy the top item to a temporary location. Decrement the number of items in the queue. Delete previously allocated memory. Increment **Head** to the next location. Return the top item to the calling statement.

```

AQueue::AQueue( int size_passed ){
    Head = Tail = N_Items = 0;
    Q_Size = size_passed;
    qp = new char * [Q_Size];
}
AQueue::~AQueue( ){
    while ( !Queue_Empty( ) )
        Deque( );
    delete [] qp;
}
void
AQueue::Enque( char *ptr ){
    if ( !Queue_Full( ) ) {
        qp[Tail] = new char [strlen(ptr)+1];
        strcpy(qp[Tail], ptr);
        ++N_Items;
        Tail = ++Tail % Q_Size;
    } else cerr << "Queue full.\n";
}
char *
AQueue::Deque( ) {
    static char tmp[512];
    if ( !Queue_Empty( ) ){
        strcpy(tmp, qp[Head]);
        --N_Items;
        delete qp[Head];
        Head = ++Head % Q_Size;
        return tmp;
    } else return (char *) NULL;
}
void
AQueue::Print( ) const {
    if ( !Queue_Empty( ) ){
        int index = Head;
        cout << "\nHead-> [ ";
        for (int i=0; i < N_Items; ++i){
            cout << qp[index];
            if ( i < N_Items-1 ) cout << " | ";
            index = ++index % Q_Size;
        }
        cout << " ] <-Tail\n";
    } else cerr << "Queue empty.\n";
}

```

## Implementing a Queue as a Linked List

Most often, a linked list is used when implementing a queue class. With a linked list, the restriction of having to know in advance the maximum size of the queue is removed.<sup>4</sup> The example on the next page provides a queue class called **LQueue** that is based on a linked list data structure. This implementation uses the **Node** class, described earlier in this chapter, to hold the data for each node. Note, if the **LQueue** class is to use the **Node** class, the statement

```
friend class LQueue;
```

must be placed in the **Node** class declaration.

The constructor for **LQueue** sets two pointer variables to **NULL**. These pointers, **Head** and **Tail**, are used to reference the first and last element in the list. The **LQueue** destructor, which removes all of the elements in the linked list, is followed by resetting the **Head** and **Tail** pointers to **NULL**.

The **Enque( )** method adds an element to the end of the list (the queue). To accomplish this, a new **Node** object is allocated dynamically. This node is passed the data to be stored and is a **NULL** value (as it will be the last element in the list). Once allocated, an **if** statement determines whether the current list is empty (**Head** has the value **NULL**). If the list is empty (this is the first node to be added), the **Head** pointer is updated to reference the new node; otherwise, the **Link** field of the last element in the list references the new node.

The **Deque( )** member function removes a node from the list. Although there is no check for a full queue when adding elements, there is a check for an empty queue when removing an element. If the queue is empty (**Head** is **NULL**), a **NULL** value is returned. If the list is not empty, a temporary location is allocated and the data item from the first element of the queue is copied to the temporary location. As with the array implementation, this location is qualified as **static**. A temporary pointer, **zombie**, is used to hold the reference to the node in the queue that will be deleted. The **Head** pointer is updated to reference the next element in the list. A call to **delete** removes the node that was storing the data. The data in the temporary storage location is returned to the calling statement.

A **Print( )** method is defined to display the queue on standard output. The method labels the head and tail of the queue. The list is in a bracketed form with each data item in each element separated by a **|** character.

A program that uses the **LQueue** class can be found on the next set of pages.

---

<sup>4</sup>Technically, this restriction is not entirely removed as we can run out of memory if we attempt to add too many items to the queue and all of the heap memory is consumed.

```
#include <iostream.h>
#include "Node.h"
```

Each element in the linked list will be a node. The header file for **Node** must be included.

```
class LQueue{
public:
    LQueue( ): Head( NULL ), Tail( NULL ) {}
    ~LQueue( );
    void Enque( char * );
    char *Deque( );
    bool Queue_Empty( ) const { return Head == NULL; }
    void Print( ) const;
private:
    Node *Head, *Tail;
};
```

When instantiating a **LQueue** object, set **Head** and **Tail** to **NULL**.

Determines if the queue is empty by checking if **Head** is set to **NULL**.

```
LQueue::~LQueue( ) {
    while( !Queue_Empty( ) )
        Deque( );
    Head = Tail = NULL;
}
void
LQueue::Enque( char *ptr ){
    Node *temp = new Node( ptr, NULL );
    if ( Head == NULL )
        Head = temp;
    else
        Tail->Link = temp;
    Tail = temp;
}
```

To add a node to the queue, a new **Node** object must be created. The new **Node** is passed the data item and a **NULL** (it will be the last element of the list). The address of the new **Node** is assigned to **Head** if the list was empty; otherwise, it is assigned to the **Link** field of the last element in the list.

```
char *
LQueue::Deque( void ){
    if ( !Queue_Empty( ) ) {
        static char *temp = new char[strlen(Head->Item)+1];
        strcpy( temp, Head->Item);
        Node *zombie = Head;
        Head = Head->Link;
        delete zombie;
        return temp;
    }
    return NULL;
}
```

If the queue is not empty, return data; otherwise, return a **NULL**.

```
void
LQueue::Print( ) const {
    if ( Head ) {
        cout << "\nHead-> [ ";
        Node *curr = Head;
        while ( curr ){
            cout << curr->Item;
            curr = curr->Link;
            cout << (curr ? " | " : " ] <-Tail\n");
        }
    } else
        cout << "Empty list\n";
}
```

The data in the node at the front of the list is to be returned. A temporary location is allocated to store this data. A temporary pointer is used to store the reference to the node to be deleted. Update the **Head** pointer, **delete** the node, and return the data value.

## Using a Linked List Queue

Another use for a queue data structure is to convert expressions in one form to another. In the previous section on stacks, we discussed converting a postfix (RPN) expression to its fully parenthesized infix form.<sup>5</sup> Expressions may also be written in prefix form. As with postfix (RPN) form, prefix expressions do not need parentheses to specify the order of operations. In prefix form, the operator precedes its two operands. An infix expression and its postfix and prefix counterparts are shown here.

INFIX	A * ( B + C )
POSTFIX	A B C + *
PREFIX	* A + B C

As demonstrated in the example on the opposite page, a queue can be used to convert a properly formed infix expression to its fully parenthesized infix form. The algorithm used by the program, although simplistic, does accomplish the task at hand. Two queue objects are used. The initial prefix expression is added to the first queue. The queue is then searched for an operator. As each element in the queue is checked, it is added to the next queue. Once an operator is encountered, the next two elements of the queue are checked in order. If they are both operands, a subexpression with parentheses is generated and placed in the next queue. Once a subexpression has been added, the remainder of the first queue is added to the next queue. If either of the next two elements are not operands, the variables holding the current “operator” and/or left/right operand are updated accordingly. A switch from the first queue to the second is made each time a subexpression is added to the current queue. After multiple passes, the queue contains the expression in the desired format.

The program that implements the algorithm begins by having the user enter the infix expression from the keyboard. No attempt is made to determine if the expression is valid. The entry of the letter ‘q’ indicates and ends input.

A **do . . while** loop is used to process the input expression. If an operator is needed, the current queue is searched until one is found. Nonoperators are added to the next queue as encountered. Once an operator is encountered, the next element of the current queue is removed, stored as the left operand, and tested. If it’s truly an operand, another element is removed and stored as the right operand. If either the left or right “operands” are not operands but are operators, an adjustment is made in the corresponding **else** branch of their **if** statement. If an operator, operand, or operand sequence is found, it is turned into a subexpression using a **sprintf( )** function. This subexpression is added to the next queue along with the remains of the current queue. The indices used to reference the current and next queues are exchanged using a series of XOR statements. Once the correct number of subexpressions has been created (one per operator—the number of operators is one less than the number of operands), processing terminates and the contents of the queue with the final output is displayed.

<sup>5</sup>For those seeking additional information, any standard Data Structures text should contain a more detailed discussion of prefix, infix, and postfix expressions.

```

#include <iostream.h>
#include <stdio.h>
#include "LQueue.h"

inline bool is_op( char *c_ptr ) {
    return (bool) strchr("+-/*", c_ptr[0]);
}

int
main( ) {
    int          n = 0, count = 0, curr = 0, next = 1;
    LQueue       Q[2];
    static char  op[512], temp[512], left_side[512], right_side[512];
    bool         need_op = true;

    do {
        cout << "> ";
        cin >> temp;
        if (temp[0] == 'q')
            break;
        Q[curr].Enque(temp);
        ++n;
    } while (true);
    Q[curr].Print( );
    do {
        if ( need_op )
            do {
                strcpy(op, Q[curr].Deque( ));
                if ( is_op(op) ) break;
                else          Q[next].Enque(temp);
            } while (true);
        need_op = false;
        strcpy(left_side, Q[curr].Deque( ));

        if ( !is_op(left_side) ) {
            strcpy(right_side, Q[curr].Deque( ));

            if ( !is_op(right_side) ) {
                sprintf(temp, "(%s %s %s)", left_side, op, right_side);
                Q[next].Enque(temp);
                need_op = true;
                while (!Q[curr].Queue_Empty( ))
                    Q[next].Enque(Q[curr].Deque( ));
                curr ^= next, next ^= curr, curr ^= next;
                if ( ++count > n/2-1 ) break;
            } else {
                Q[next].Enque(op);
                Q[next].Enque(left_side);
                strcpy(op, right_side);
            }
        } else {
            Q[next].Enque(op);
            strcpy(op, left_side);
        }
    } while( true );
    cout << "Here is the expression with ( )'s\n";
    if ( !Q[curr].Queue_Empty( ) )
        cout << Q[curr].Deque( ) << endl;
    else
        cout << Q[next].Deque( ) << endl;
    return 0;
}

```

Boolean function that determines if first character passed is an operator.

Obtain the prefix expression from the user. Entry of 'q' stops processing. No error checking is done.

If an operator is needed, search the current queue. If removed item is an operator, stop processing; otherwise, add the item to the next queue.

Attempt to get the left and right operands of the operator.

Create the subexpression and add it to the next queue. Add the remains of the current queue to the next queue. Exchange indices for current and next queue. Exit loop if done.

If not an operand, update accordingly.

#### Sample Output

```

> *
> A
> +
> B
> C
> q

Head-> [ * | A | + | B | C ] <-Tail
Here is the expression with ( )'s
(A * (B + C))

```

