Pointers and Linked Lists

15.1 Nodes and Linked Lists 828
- Nodes 828
- Linked Lists 834
- Inserting a Node at the Head of a List 835
  Pitfall: Losing Nodes 839
- Searching a Linked List 840
- Pointers as Iterators 843
- Inserting and Removing Nodes Inside a List 843
  Pitfall: Using the Assignment Operator with Dynamic Data Structures 846

15.2 A Linked List Application 850
- Stacks 850
  Programming Example: A Stack Class 850

Chapter Summary 857
Answers to Self-Test Exercises 857
Programming Projects 859
Pointers and Linked Lists

If somebody there chanced to be
Who loved me in a manner true
My heart would point him out to me
And I would point him out to you.

GILBERT AND SULLIVAN, Ruddigore

Introduction

A linked list is a list constructed using pointers. A linked list is not fixed in size but can grow and shrink while your program is running. This chapter shows you how to define and manipulate linked lists, which will serve to introduce you to a new way of using pointers.

Prerequisites

This chapter uses material from Chapters 2 through 12.

15.1 Nodes and Linked Lists

Useful dynamic variables are seldom of a simple type such as int or double, but are normally of some complex type such as an array, struct, or class type. You saw that dynamic variables of an array type can be useful. Dynamic variables of a struct or class type can also be useful, but in a different way. Dynamic variables that are either structs or classes normally have one or more member variables that are pointer variables that connect them to other dynamic variables. For example, one such structure, which happens to contain a shopping list, is diagrammed in Display 15.1.

Nodes

A structure like the one shown in Display 15.1 consists of items that we have drawn as boxes connected by arrows. The boxes are called nodes and the arrows represent pointers. Each of the nodes in Display 15.1 contains a string, an integer, and a pointer
that can point to other nodes of the same type. Note that pointers point to the entire node, not to the individual items (such as 10 or "rolls") that are inside the node.

Nodes are implemented in C++ as structs or classes. For example, the struct type definitions for a node of the type shown in Display 15.1, along with the type definition for a pointer to such nodes, can be as follows:

```c++
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};

typedef ListNode* ListNodePtr;
```

The order of the type definitions is important. The definition of ListNode must come first, since it is used in the definition of ListNodePtr.
The box labeled head in Display 15.1 is not a node but is a pointer variable that can point to a node. The pointer variable head is declared as follows:

\[\text{ListNodePtr head;}\]

Even though we have ordered the type definitions to avoid some illegal forms of circularity, the above definition of the \textit{struct} type ListNode is still blatantly circular. The definition of the type ListNode uses the type name ListNode to define the member variable link. There is nothing wrong with this particular circularity, and it is allowed in C++. One indication that this definition is not logically inconsistent is the fact that you can draw pictures, like Display 15.1, that represent such structures.

We now have pointers inside of \textit{struct}s and have these pointers pointing to \textit{struct}s that contain pointers, and so forth. In such situations the syntax can sometimes get involved, but in all cases the syntax follows those few rules we have described for pointers and \textit{struct}s. As an illustration, suppose the declarations are as above, the situation is as diagrammed in Display 15.1, and you want to change the number in the first node from 10 to 12. One way to accomplish this is with the following statement:

\[(*\text{head}).\text{count} = 12;\]

The expression on the left side of the assignment operator may require a bit of explanation. The variable head is a pointer variable. So, the expression \(*\text{head}\) is the thing it points to, namely the node (dynamic variable) containing "rolls" and the integer 10. This node, referred to by \(*\text{head}\), is a \textit{struct}, and the member variable of this \textit{struct}, which contains a value of type \textit{int}, is called \textit{count}, and so \((\star\text{head}).\text{count}\) is the name of the \textit{int} variable in the first node. The parentheses around \(*\text{head}\) are not optional. You want the dereferencing operator \(\star\) to be performed before the dot operator. However, the dot operator has higher precedence than the dereferencing operator \(\star\), and so without the parentheses, the dot operator would be performed first (and that would produce an error). In the next paragraph, we will describe a shortcut notation that can avoid this worry about parentheses.

C++ has an operator that can be used with a pointer to simplify the notation for specifying the members of a \textit{struct} or a class. The \textbf{arrow operator} \(\rightarrow\) combines the actions of a dereferencing operator \(\star\) and a dot operator to specify a member of a dynamic \textit{struct} or object that is pointed to by a given pointer. For example, the above assignment statement for changing the number in the first node can be written more simply as

\[\text{head->count} = 12;\]

This assignment statement and the previous one mean the same thing, but this one is the form normally used.
The string in the first node can be changed from "rolls" to "bagels" with the following statement:

```c
head->item = "bagels";
```

The result of these changes to the first node in the list is diagrammed in Display 15.2.

Look at the pointer member in the last node in the lists shown in Display 15.2. This last node has the word NULL written where there should be a pointer. In Display 15.1 we filled this position with the phrase "end marker," but "end marker" is not a C++ expression. In C++ programs we use the constant NULL as an end marker to signal the end of a linked list. NULL is a special defined constant that is part of the C++ language (provided as part of the required C++ libraries).

NULL is typically used for two different (but often coinciding) purposes. It is used to give a value to a pointer variable that otherwise would not have any value. This prevents an inadvertent reference to memory, since NULL is not the address of any memory location. The second category of use is that of an end marker. A program

**Display 15.2 Accessing Node Data**

```c
head->count = 12;
head->item = "bagels";
```
can step through the list of nodes as shown in Display 15.2, and when the program reaches the node that contains NULL, it knows that it has come to the end of the list.

The constant NULL is actually the number 0, but we prefer to think of it and spell it as NULL. That makes it clear that you mean this special-purpose value that you can assign to pointer variables. The definition of the identifier NULL is in a number of the standard libraries, such as <iostream> and <cstddef>, so you should use an include directive with either <iostream>, or <cstddef> (or other suitable library) when you use NULL. No using directive is needed in order to make NULL available to your program code. In particular, it does not require using namespace std; 1 although other things in your code are likely to require something like using namespace std; 1

A pointer can be set to NULL using the assignment operator, as in the following, which declares a pointer variable called there and initializes it to NULL:

double *there = NULL;

The constant NULL can be assigned to a pointer variable of any pointer type.

1The details are as follows: The definition of NULL is handled by the C++ preprocessor, which replaces NULL with 0. Thus, the compiler never actually sees "NULL" and so there are no namespace issues, and no using directive is needed.
**SELF-TEST EXERCISES**

1. Suppose your program contains the following type definitions:

   ```
   struct Box {
       string name;
       int number;
       Box *next;
   }
   
   typedef Box* BoxPtr;
   ```

   What is the output produced by the following code?

   ```
   BoxPtr head;
   head = new Box;
   head->name = "Sally";
   head->number = 18;
   cout << (*head).name << endl;
   cout << head->name << endl;
   cout << (*head).number << endl;
   cout << head->number << endl;
   ```

2. Suppose that your program contains the type definitions and code given in Self-Test Exercise 1. That code creates a node that contains the string "Sally" and the number 18. What code would you add in order to set the value of the member variable `next` of this node equal to `NULL`?

3. Suppose that your program contains the type definitions and code given in Self-Test Exercise 1. Assuming that the value of the pointer variable `head` has not been changed, how can you destroy the dynamic variable pointed to by `head` and return the memory it uses to the freestore so that it can be reused to create new dynamic variables?
Given the following structure definition:

```c
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};
```

```
ListNode *head = new ListNode;
```

Write code to assign the string "Wilbur's brother Orville" to the member `item` of the node pointed to by `head`.

**Linked Lists**

Lists such as those shown in Display 15.2 are called **linked lists**. A **linked list** is a list of nodes in which each node has a member variable that is a pointer that points to the next node in the list. The first node in a linked list is called the **head**, which is why the pointer variable that points to the first node is named `head`. Note that the pointer named `head` is not itself the head of the list but only points to the head of the list. The last node has no special name, but it does have a special property. The last node has `NULL` as the value of its member pointer variable. To test to see whether a node is the last node, you need only test to see if the pointer variable in the node is equal to `NULL`.

Our goal in this section is to write some basic functions for manipulating linked lists. For variety, and to simplify the notation, we will use a simpler type of node than that used in Display 15.2. These nodes will contain only an integer and a pointer. The node and pointer type definitions that we will use are as follows:

```c
struct Node
{
    int data;
    Node *link;
};
```

```c
typedef Node* NodePtr;
```

As a warm-up exercise, let’s see how we might construct the start of a linked list with nodes of this type. We first declare a pointer variable, called `head`, that will point to the head of our linked list:

```
NodePtr head;
```

To create our first node, we use the operator `new` to create a new dynamic variable that will become the first node in our linked list.
Nodes and Linked Lists

We then give values to the member variables of this new node:

```cpp
head->data = 3;
head->link = NULL;
```

Notice that the pointer member of this node is set equal to NULL. That is because this node is the last node in the list (as well as the first node in the list). At this stage our linked list looks like this:

![Linked List Diagram]

Our one-node list was built in a purely ad hoc way. To have a larger linked list, your program must be able to add nodes in a systematic way. We next describe one simple way to insert nodes in a linked list.

### Inserting a Node at the Head of a List

In this subsection we assume that our linked list already contains one or more nodes, and we develop a function to add another node. The first parameter for the insertion function will be a call-by-reference parameter for a pointer variable that points to the head of the linked list, that is, a pointer variable that points to the first node in the linked list. The other parameter will give the number to be stored in the new node. The function declaration for our insertion function is as follows:

```cpp
void head_insert(NodePtr& head, int the_number);
```

To insert a new node into the linked list, our function will use the `new` operator to create a new node. The data is then copied into the new node, and the new node is inserted at the head of the list. When we insert nodes this way, the new node will be the first node in the list (that is, the head node) rather than the last node. Since dynamic variables have no names, we must use a local pointer variable to point to

---

**Linked Lists as Arguments**

You should always keep one pointer variable pointing to the head of a linked list. This pointer variable is a way to name the linked list. When you write a function that takes a linked list as an argument, this pointer (which points to the head of the linked list) can be used as the linked list argument.
this node. If we call the local pointer variable temp_ptr, the new node can be referred to as *temp_ptr. The complete process can be summarized as follows:

**Pseudocode for head_insert Function**

1. Create a new dynamic variable pointed to by temp_ptr. (This new dynamic variable is the new node. This new node can be referred to as *temp_ptr.)
2. Place the data in this new node.
3. Make the link member of this new node point to the head node (first node) of the original linked list.
4. Make the pointer variable named head point to the new node.

Display 15.3 contains a diagram of this algorithm. Steps 2 and 3 in the diagram can be expressed by the C++ assignment statements given below:

```cpp
temp_ptr->link = head;
head = temp_ptr;
```

The complete function definition is given in Display 15.4.

You will want to allow for the possibility that a list contains nothing. For example, a shopping list might have nothing in it because there is nothing to buy this week. A list with nothing in it is called an **empty list**. A linked list is named by naming a pointer that points to the head of the list, but an empty list has no head node. To specify an empty list, you use the pointer **NULL**. If the pointer variable head is supposed to point to the head node of a linked list and you want to indicate that the list is empty, then you set the value of head as follows:

```cpp
head = NULL;
```

Whenever you design a function for manipulating a linked list, you should always check to see if it works on the empty list. If it does not, you may be able to add a special case for the empty list. If you cannot design the function to apply to the empty list, then your program must be designed to handle empty lists some other way or to avoid them completely. Fortunately, the empty list can often be treated just like any other list. For example, the function head_insert in Display 15.4 was designed with nonempty lists as the model, but a check will show that it works for the empty list as well.
Display 15.3 Adding a Node to a Linked List

1. Set up new node

2. temp_ptr->link = head;

3. head = temp_ptr;

4. After function call
Display 15.4 Function to Add a Node at the Head of a Linked List

Function Declaration

```c
struct Node
{
    int data;
    Node *link;
};
```

typedef Node* NodePtr;

```c
void head_insert(NodePtr& head, int the_number);
// Precondition: The pointer variable head points to
// the head of a linked list.
// Postcondition: A new node containing the_number
// has been added at the head of the linked list.
```

Function Definition

```c
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;
    temp_ptr->data = the_number;
    temp_ptr->link = head;
    head = temp_ptr;
}
```
PITFALL Losing Nodes

You might be tempted to write the function definition for `head_insert` (Display 15.4) using the pointer variable `head` to construct the new node, instead of using the local pointer variable `temp_ptr`. If you were to try, you might start the function as follows:

```c
head = new Node;
head->data = the_number;
```

At this point the new node is constructed, contains the correct data, and is pointed to by the pointer `head`, all as it is supposed to be. All that is left to do is to attach the rest of the list to this node by setting the pointer member given below so that it points to what was formerly the first node of the list:

```c
head->link
```

Display 15.5 shows the situation when the new data value is 12. That illustration reveals the problem. If you were to proceed in this way, there would be nothing pointing to the node containing 15. Since there is no named pointer pointing to it (or to a chain of pointers ending with that node), there is no way the program can reference this node. The node below this node is also lost. A program cannot make a pointer point to either of these nodes, nor can it access the data in these nodes, nor can it do anything else to the nodes. It simply has no way to refer to the nodes.

Display 15.5 Lost Nodes

```
  head
  12
  ?
15

Lost nodes
  3
  NULL
```
Such a situation ties up memory for the duration of the program. A program that loses nodes is sometimes said to have a “memory leak.” A significant memory leak can result in the program running out of memory, causing abnormal termination. Worse, a memory leak (lost nodes) in an ordinary user’s program can cause the operating system to crash. To avoid such lost nodes, the program must always keep some pointer pointing to the head of the list, usually the pointer in a pointer variable like head.

**Searching a Linked List**

Next we will design a function to search a linked list in order to locate a particular node. We will use the same node type, called Node, that we used in the previous subsections. (The definition of the node and pointer types are given in Display 15.4.) The function we design will have two arguments: the linked list and the integer we want to locate. The function will return a pointer that points to the first node that contains that integer. If no node contains the integer, the function will return the pointer NULL. This way our program can test to see whether the integer is on the list by checking to see if the function returns a pointer value that is not equal to NULL. The function declaration and header comment for our function is as follows:

```pseudocode
NodePtr search(NodePtr head, int target);
// Precondition: The pointer head points to the head of
// a linked list. The pointer variable in the last node
// is NULL. If the list is empty, then head is NULL.
// Returns a pointer that points to the first node that
// contains the target. If no node contains the target,
// the function returns NULL.
```

We will use a local pointer variable, called here, to move through the list looking for the target. The only way to move around a linked list, or any other data structure made up of nodes and pointers, is to follow the pointers. So we will start with here pointing to the first node and move the pointer from node to node following the pointer out of each node. This technique is diagrammed in Display 15.6. Since empty lists present some minor problems that would clutter our discussion, we will at first assume that the linked list contains at least one node. Later we will come back and make sure the algorithm works for the empty list as well. This search technique yields the following algorithm:

**Pseudocode for search Function**

1. Make the pointer variable here point to the head node (that is, first node) of the linked list.
Display 15.6 Searching a Linked List

1. target is 6

2. Not here

3. Not here

4. Found
while (here is not pointing to a node containing target 
and here is not pointing to the last node)
{
    Make here point to the next node in the list.
}
if (the node pointed to by here contains target)
    return here;
else
    return NULL;

In order to move the pointer here to the next node, we must think in terms of the 
named pointers we have available. The next node is the one pointed to by the pointer 
member of the node currently pointed to by here. The pointer member of the node 
currently pointed to by here is given by the expression

    here->link

To move here to the next node, we want to change here so that it points to the 
ode that is pointed to by the above-named pointer (member) variable. Hence, the 
following will move the pointer here to the next node in the list:

    here = here->link;

Putting these pieces together yields the following refinement of the algorithm 
pseudocode:

Preliminary Version of the Code for the search Function

    here = head;
    while (here->data != target &&
            here->link != NULL)
        here = here->link;
    if (here->data == target)
        return here;
    else
        return NULL;

Notice the Boolean expression in the while statement. We test to see if here is not 
pointing to the last node by testing to see if the member variable here->link is not 
equal to NULL.
We still must go back and take care of the empty list. If we check the above code we find that there is a problem with the empty list. If the list is empty, then here is equal to NULL and hence the following expressions are undefined:

```
here->data
here->link
```

When here is NULL, it is not pointing to any node, so there is no member named data nor any member named link. Hence, we make a special case of the empty list. The complete function definition is given in Display 15.7.

**Pointers as Iterators**

An iterator is a construct that allows you to cycle through the data items stored in a data structure so that you can perform whatever action you want on each data item. An iterator can be an object of some iterator class or something simpler, such as an array index or a pointer. Pointers provide a simple example of an iterator. In fact, a pointer is the prototypical example of an iterator. The basic ideas can be easily seen in the context of linked lists. You can use a pointer as an iterator by moving through the linked list one node at a time starting at the head of the list and cycling through all the nodes in the list. The general outline is as follows:

```
Node_Type *iter;
for (iter = Head; iter != NULL; iter = iter->Link)
    Do whatever you want with the node pointed to by iter;
```

where Head is a pointer to the head node of the linked list and Link is the name of the member variable of a node that points to the next node in the list.

For example, to output the data in all the nodes in a linked list of the kind we have been discussing, you could use

```
NodePtr iter; // Equivalent to: Node *iter;
for (iter = head; iter != NULL; iter = iter->Link)
    cout << (iter->data);
```

The definition of Node and NodePtr are given in Display 15.7.

**Inserting and Removing Nodes Inside a List**

We next design a function to insert a node at a specified place in a linked list. If you want the nodes in some particular order, such as numeric order or alphabetical order,
Display 15.7  Function to Locate a Node in a Linked List

Function Declaration

```c
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

NodePtr search(NodePtr head, int target);
// Precondition: The pointer head points to the head of // a linked list. The pointer variable in the last node // is NULL. If the list is empty, then head is NULL. // Returns a pointer that points to the first node that // contains the target. If no node contains the target, // the function returns NULL.

Function Definition

// Uses cstddef:
NodePtr search(NodePtr head, int target)
{
    NodePtr here = head;
    if (here == NULL)
    {
        return NULL;  // Empty list case
    }
    else
    {
        while (here->data != target &&
               here->link != NULL)
        {
            here = here->link;
        }
        if (here->data == target)
        {
            return here;
        }
        else
        {
            return NULL;
        }
    }
}
you cannot simply insert the node at the beginning or end of the list. We will therefore design a function to insert a node after a specified node in the linked list. We assume that some other function or program part has correctly placed a pointer called after_me pointing to some node in the linked list. We want the new node to be placed after the node pointed to by after_me, as illustrated in Display 15.8. The same technique works for nodes with any kind of data, but to be concrete, we are using the same type of nodes as in previous subsections. The type definitions are given in Display 15.7. The function declaration for the function we want to define is given below:

```c
void insert(NodePtr after_me, int the_number);
// Precondition: after_me points to a node in a linked list.
// Postcondition: A new node containing the_number has been added after the node pointed to by after_me.
```

A new node is set up the same way it was in the function head_insert in Display 15.4. The difference between this function and that one is that we now wish to insert the node not at the head of the list but after the node pointed to by after_me. The way to do the insertion is shown in Display 15.8 and is expressed as follows in C++ code:

```c
// add a link from the new node to the list:
    temp_ptr->link = after_me->link;
// add a link from the list to the new node:
    after_me->link = temp_ptr;
```

The order of these two assignment statements is critical. In the first assignment we want the pointer value after_me->link before it is changed. The complete function is given in Display 15.9.

If you go through the code for the function insert, you will see that it works correctly even if the node pointed to by after_me is the last node in the list. However, insert will not work for inserting a node at the beginning of a linked list. The function head_insert given in Display 15.4 can be used to insert a node at the beginning of a list.

By using the function insert you can maintain a linked list in numerical order or alphabetical order or other ordering. You can "squeeze" a new node into the correct position by simply adjusting two pointers. This is true no matter how long the linked list is or where in the list you want the new data to go. If you instead used an array, much, and in extreme cases all, of the array would have to be copied in order to make room for a new value in the correct spot. Despite the overhead involved in positioning the pointer after_me, inserting into a linked list is frequently more efficient than inserting into an array.
Removing a node from a linked list is also quite easy. Display 15.10 illustrates the method. Once the pointers before and discard have been positioned, all that is required to remove the node is the following statement:

\[ \text{before->link} = \text{discard->link}; \]

This is sufficient to remove the node from the linked list. However, if you are not using this node for something else, you should destroy the node and return the memory it uses to the freestore; you can do this with a call to delete as follows:

\[ \text{delete discard}; \]

**PITFALL** Using the Assignment Operator with Dynamic Data Structures

If head1 and head2 are pointer variables and head1 points to the head node of a linked list, the following will make head2 point to the same head node and hence the same linked list:

\[ \text{head2} = \text{head1}; \]
However, you must remember that there is only one linked list, not two. If you change the linked list pointed to by head1, then you will also change the linked list pointed to by head2, because they are the same linked list.

If head1 points to a linked list and you want head2 to point to a second, identical copy of this linked list, the above assignment statement will not work. Instead, you must copy the entire linked list node by node. Alternatively, you can overload

Display 15.9 Function to Add a Node in the Middle of a Linked List

Function Declaration

```c
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

void insert(NodePtr after_me, int the_number);
// Precondition: after_me points to a node in a linked list.
// Postcondition: A new node containing the_number has been added after the node pointed to by after_me.
```

Function Definition

```c
void insert(NodePtr after_me, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;
    temp_ptr->link = after_me->link;
    after_me->link = temp_ptr;
}
```
Display 15.10 Removing a Node

1. Position the pointer discard so that it points to the node to be deleted, and position the pointer before so that it points to the node before the one to be deleted.

2. before->link = discard->link;

3. delete discard;
the assignment operator = so that it means whatever you want it to mean. Overloading = is discussed in the subsection of Chapter 12 entitled “Overloading the Assignment Operator.”

**SELF-TEST EXERCISES**

5 Write type definitions for the nodes and pointers in a linked list. Call the node type NodeType and call the pointer type PointerType. The linked lists will be lists of letters.

6 A linked list is normally given by giving a pointer that points to the first node in the list, but an empty list has no first node. What pointer value is normally used to represent an empty list?

7 Suppose your program contains the following type definitions and pointer variable declarations:

```c
struct Node
{
    double data;
    Node *next;
};

typedef Node* Pointer;
Pointer p1, p2;
```

Suppose p1 points to a node of the above type that is on a linked list. Write code that will make p1 point to the next node on this linked list. (The pointer p2 is for the next exercise and has nothing to do with this exercise.)

8 Suppose your program contains type definitions and pointer variable declarations as in Self-Test Exercise 7. Suppose further that p2 points to a node of the above type that is on a linked list and is not the last node on the list. Write code that will delete the node after the node pointed to by p2. After this code is executed, the linked list should be the same, except that there will be one less node on the linked list. *Hint:* You may want to declare another pointer variable to use.

9 Choose an answer and explain it.

For a large array and large list holding the same type objects, inserting a new object at a known location into the middle of a linked list compared with insertion in an array is
15.2 A Linked List Application

"But many who are first now will be last, and many who are last now will be first."

Matthew 19:30

Linked lists have many applications. In this section we will give you only a small sample of what they can be used for. We will present one class definition that uses a linked list as the heart of its implementation.

Stacks

A stack is a data structure that retrieves data in the reverse of the order in which the data is stored. Suppose you place the letters 'A', 'B', and then 'C' in a stack. When you take these letters out of the stack, they will be removed in the order 'C', 'B', and then 'A'. This use of a stack is diagrammed in Display 15.11. As shown there, you can think of a stack as a hole in the ground. In order to get something out of the stack, you must first remove the items on top of the one you want. For this reason a stack is often called a last-in/first-out data structure.

Stacks are used for many language processing tasks. In Chapter 13 we discussed how the computer system uses a stack to keep track of C++ function calls. However, here we will do only one very simple application. Our goal in this example is to show you how you can use the linked list techniques to implement specific data structures; a stack is one simple example of the use of linked lists. You need not have read Chapter 13 to understand this example.

Programming EXAMPLE

A Stack Class

The interface for our stack class is given in Display 15.12. This particular stack is used to store data of type char. You can define a similar stack to store data of any other type. There are two basic operations you can perform on a stack: adding an
15.2 A Linked List Application

Display 15.11 A Stack

item to the stack and removing an item from the stack. Adding an item is called pushing the item onto the stack, and so we called the member function that does this push. Removing an item from a stack is called popping the item off the stack, and so we called the member function that does this pop.

The names push and pop derive from another way of visualizing a stack. A stack is analogous to a mechanism that is sometimes used to hold plates in a cafeteria. The mechanism stores plates in a hole in the countertop. There is a spring underneath the plates with its tension adjusted so that only the top plate protrudes above the countertop. If this sort of mechanism were used as a stack data structure, the data would be written on plates (which might violate some health laws, but still makes a good analogy). To add a plate to the stack, you put it on top of the other plates, and the weight of this new plate pushes down the spring. When you remove a plate, the plate below it pops into view.

Display 15.13 shows a simple program that illustrates how the stack class is used. This program reads a word one letter at a time and places the letters in a stack. The program then removes the letters one by one and writes them to the screen. Because data is removed from a stack in the reverse of the order in which it enters the stack, the output shows the word written backward.

As shown in Display 15.14, our stack class is implemented as a linked list in which the head of the list serves as the top of the stack. The member variable top is a pointer that points to the head of the linked list.
Display 15.12 Interface File for a Stack Class

// This is the header file stack.h. This is the interface for the class Stack, // which is a class for a stack of symbols.
#endif // STACK_H
#define STACK_H
namespace stacksavitch
{
    struct StackFrame {
        char data;
        StackFrame *link;
    };

typedef StackFrame* StackFramePtr;

class Stack {
    public:
        Stack();
        // Initializes the object to an empty stack.
        Stack(const Stack& a_stack);
        // Copy constructor.
        ~Stack();
        // Destroys the stack and returns all the memory to the freestore.
        void push(char the_symbol);
        // Postcondition: the_symbol has been added to the stack.
        char pop();
        // Precondition: The stack is not empty.
        // Returns the top symbol on the stack and removes that
        // top symbol from the stack.
        bool empty() const;
        // Returns true if the stack is empty. Returns false otherwise.

    private:
        StackFramePtr top;
    };
} // stacksavitch
#endif // STACK_H
Display 15.13 Program Using the Stack Class (part 1 of 2)

```cpp
// Program to demonstrate use of the Stack class.
#include <iostream>
#include "stack.h"
using namespace std;
using namespace stacksavitch;

int main()
{
    Stack s;
    char next, ans;

    do
    {
        cout << "Enter a word: ";
        cin.get(next);
        while (next != 'n')
        {
            s.push(next);
            cin.get(next);
        }

        cout << "Written backward that is: ";
        while ( ! s.empty() )
            cout << s.pop();
        cout << endl;

        cout << "Again?(y/n): ";
        cin >> ans;
        cin.ignore(10000, 'n');
    }while (ans != 'n' && ans != 'N');

    return 0;
}
```

The `ignore` member of `cin` is discussed in Chapter 11. It discards input remaining on the current input line up to 10,000 characters or until a return is entered. It also discards the return ('n') at the end of the line.
Display 15.13 Program Using the Stack Class (part 2 of 2)

Sample Dialogue

Enter a word: straw
Written backward that is: warts
Again?(y/n): y
Enter a word: C++
Written backward that is: ++C
Again?(y/n): n

Writing the definition of the member function push is Self-Test Exercise 10. However, we have already given the algorithm for this task. The code for the push member function is essentially the same as the function head_insert shown in Display 15.4, except that in the member function push we use a pointer named top in place of a pointer named head.

An empty stack is just an empty linked list, so an empty stack is implemented by setting the pointer top equal to NULL. Once you realize that NULL represents the empty stack, the implementations of the default constructor and of the member function empty are obvious.

The definition of the copy constructor is a bit complicated but does not use any techniques we have not already discussed. The details are left to Self-Test Exercise 11.

The pop member function first checks to see if the stack is empty. If the stack is not empty, it proceeds to remove the top character in the stack. It sets the local variable result equal to the top symbol on the stack. That is done as follows:

```
char result = top->data;
```

After the symbol in the top node is saved in the variable result, the pointer top is moved to the next node on the linked list, effectively removing the top node from the list. The pointer top is moved with the following statement:

```
top = top->link;
```

However, before the pointer top is moved, a temporary pointer, called temp_ptr, is positioned so that it points to the node that is about to be removed from the list. The node can then be removed with the following call to delete:

```
delete temp_ptr;
```
Each node that is removed from the linked list by the member function pop is destroyed with a call to delete. Thus, all that the destructor needs to do is remove each item from the stack with a call to pop. Each node will then have its memory returned to the freestore.

**SELF-TEST EXERCISES**

10 Give the definition of the member function push of the class Stack described in Display 15.12.

11 Give the definition of the copy constructor for the class Stack described in Display 15.12.
Display 15.14 Implementation of the Stack Class (part 2 of 2)

Stack::~Stack()
{
    char next;
    while (! empty())
        next = pop(); // pop calls delete.
}

// Uses cstddef:
bool Stack::empty() const
{
    return (top == NULL);
}

void Stack::push(char the_symbol)
<The rest of the definition is Self-Test Exercise 10.>

// Uses iostream:
char Stack::pop()
{
    if (empty())
    {
        cout << "Error: popping an empty stack.\n";
        exit(1);
    }

    char result = top->data;

    StackFramePtr temp_ptr;
    temp_ptr = top;
    top = top->link;

    delete temp_ptr;

    return result;
}
CHAPTER SUMMARY

- A node is a struct or class object that has one or more member variables that are pointer variables. These nodes can be connected by their member pointer variables to produce data structures that can grow and shrink in size while your program is running.

- A linked list is a list of nodes in which each node contains a pointer to the next node in the list.

- The end of a linked list (or other linked data structure) is indicated by setting the pointer member variable equal to NULL.

- A stack is a first-in/last-out data structure. A stack can be implemented using a linked list.

Answers to Self-Test Exercises

1 Note that (*head).name and head->name mean the same thing. Similarly, (*head).number and head->number mean the same thing.

2 The best answer is

   head->next = NULL;

However, the following is also correct:

   (*head).next = NULL;

3 delete head;

4 head->item = "Wilbur's brother Orville";
5 struct NodeType
   {
      char data;
      NodeType *link;
   };

typedef NodeType* PointerType;

6 The pointer value NULL is used to indicate an empty list.

7 p1 = p1-> next;

8 Pointer discard;
   discard = p2->next;
   // discard now points to the node to be deleted.
   p2->next = discard->next;

   This is sufficient to delete the node from the linked list. However, if you are
   not using this node for something else, you should destroy the node with a
   call to delete as follows:

   delete discard;

9 a. Inserting a new item at a known location into a large linked list is more
    efficient than inserting into a large array. If you are inserting into a list,
    you have about five operations, most of which are pointer assignments,
    regardless of the list size. If you insert into an array, on the average you
    have to move about half the array entries to insert a data item.

   For small lists, the answer is (c), about the same.

10 //Uses cstddef:
    void Stack::push(char the_symbol)
    {
       StackFramePtr temp_ptr;
       temp_ptr = new StackFrame;

       temp_ptr->data = the_symbol;

       temp_ptr->link = top;
       top = temp_ptr;
    }
Programming Projects

11 //Uses cstddef:
Stack::Stack(const Stack& a_stack)
{
    if (a_stack.top == NULL)
        top = NULL;
    else
    {
        StackFramePtr temp = a_stack.top;//temp moves
        //through the nodes from top to bottom of
        //a_stack.
        StackFramePtr end;//Points to end of the new stack.

        end = new StackFrame;
        end->data = temp->data;
        top = end;
        //First node created and filled with data.
        //New nodes are now added AFTER this first node.

        temp = temp->link;
        while (temp != NULL)
        {
            end->link = new StackFrame;
            end = end->link;
            end->data = temp->data;
            temp = temp->link;
        }
        end->link = NULL;
    }
}

Programming Projects

1 Write a void function that takes a linked list of integers and reverses the order of its nodes. The function will have one call-by-reference parameter that is a pointer to the head of the list. After the function is called, this pointer will point to the head of a linked list that has the same nodes as the original list, but in the reverse of the order they had in the original list. Note that your function will neither create nor destroy any nodes. It will simply rearrange nodes. Place your function in a suitable test program.
2 Write a function called `merge_lists` that takes two call-by-reference arguments that are pointer variables that point to the heads of linked lists of values of type `int`. The two linked lists are assumed to be sorted so that the number at the head is the smallest number, the number in the next node is the next smallest, and so forth. The function returns a pointer to the head of a new linked list that contains all of the nodes in the original two lists. The nodes in this longer list are also sorted from smallest to largest values. Note that your function will neither create nor destroy any nodes. When the function call ends, the two pointer variable arguments should have the value `NULL`.

3 A **queue** is a data structure that is similar to a stack except that whereas a stack is a last-in/first-out data structure, a queue is a first-in/first-out data structure. In a queue of `chars`, if you add the characters 'A', 'B', and 'C' in that order, then they are removed in the order 'A', 'B', and then 'C'. Write a class for a queue of characters with an interface similar to that given for a stack in Display 15.12. However, your member function for adding a character will be named `add` rather than `push`, and your member function for removing a character will be named `remove` rather than `pop`. Be sure to write a full implementation of all the member functions and overloaded operators. Write a program to test your class.

4 Design and implement a class whose objects represent polynomials. The polynomial

\[ a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 \]

will be implemented as a linked list. Each node will contain an `int` value for the power of `x` and an `int` value for the corresponding coefficient. The class operations should include addition, subtraction, multiplication, and evaluation of a polynomial. Overload the operators `+`, `−`, and `*` for addition, subtraction, and multiplication.

Evaluation of a polynomial is implemented as a member function with one argument of type `int`. The evaluation member function returns the value obtained by plugging in its argument for `x` and performing the indicated operations. Include four constructors: a default constructor, a copy constructor, a constructor with a single argument of type `int` that produces the polynomial that has only one constant term that is equal to the constructor argument, and a constructor with two arguments of type `int` that produces the one-term polynomial whose coefficient and exponent are given by the two arguments. (In the above notation, the polynomial produced by the one-argument constructor is of the simple form consisting of only \(a_0\). The polynomial
produced by the two-argument constructor is of the slightly more complicated form \( a_nx^n \). Include a suitable destructor. Include member functions to input and output polynomials.

When the user inputs a polynomial, the user types in the following:

\[ a_nx^n + a_{n-1}x^{n-1} + \ldots + a_0 \]

However, if a coefficient \( a_i \) is zero, the user may omit the term \( a_ix^i \). For example, the polynomial

\[ 3x^4 + 7x^2 + 5 \]

can be input as

\[ 3x^4 + 7x^2 + 5 \]

It could also be input as

\[ 3x^4 + 0x^3 + 7x^2 + 0x^1 + 5 \]

If a coefficient is negative, a minus sign is used in place of a plus sign, as in the following examples:

\[ 3x^5 - 7x^3 + 2x^1 - 8 \]
\[ -7x^4 + 5x^2 + 9 \]

A minus sign at the front of the polynomial, as in the second of the above two examples, applies only to the first coefficient; it does not negate the entire polynomial. Polynomials are output in the same format. In the case of output, the terms with zero coefficients are not output.

To simplify input, you can assume that polynomials are always entered one per line and that there will always be a constant term \( a_0 \). If there is no constant term, the user enters zero for the constant term, as in the following:

\[ 12x^8 + 3x^2 + 0 \]

In this project you will redo Programming Project 10 from Chapter 10 using a linked list instead of an array. As noted there, this is a linked list of \textit{double} items. This fact may imply changes in some of the member functions. The members are as follows: a default constructor; a member function named \textit{add_item} to add a \textit{double} to the list; a test for a full list that is a Boolean-valued function named \textit{full()}; and a \textit{friend} function overloading the insertion operator \texttt{<<}.
A harder version of Programming Project 5 would be to write a class named List, similar to Project 5, but with all the following member functions:

- Default constructor, List();
- double List::front();, which returns the first item in the list
- double List::back();, which returns the last item in the list
- double List::current();, which returns the “current” item
- void List::advance();, which advances the item that current() returns
- void List::reset(); to make current() return the first item in the list
- void List::insert(double after_me, double insert_me);, which inserts insert_me into the list after after_me and increments the private: variable count.
- int size();, which returns the number of items in the list
- friend istream& operator<< (istream& ins, double write_me);

The private data members should include the following:

```cpp
node* head;
node* current;
int count;
```

and possibly one more pointer.

You will need the following struct (outside the list class) for the linked list nodes:

```cpp
struct node
{
    double item;
    node *next;
};
```

Incremental development is essential to all projects of any size, and this is no exception. Write the definition for the List class, but do not implement any members yet. Place this class definition in a file list.h. Then #include "list.h" in a file that contains int main(). Compile your file. This will find syntax errors and many typographical errors that would cause untold difficulty if you attempted to implement members without this check. Then you should implement and compile one member at a time, until you have enough to write test code in your main function.
This is a harder version of Programming Project 6 that uses templates (which are covered in Chapter 14). Write a template class definition for a list of objects whose type is a type parameter. Be sure to write a full implementation of all the member functions and overloaded operators. Test your template list with a struct type such as the following:

```cpp
struct Person
{
    string name;
    string phone_number;
};
```