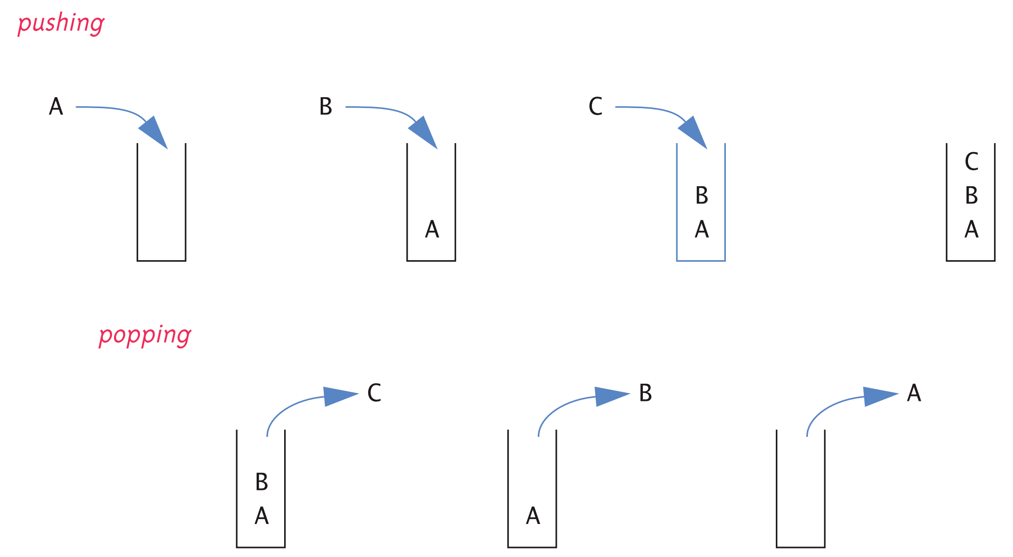
Display 17.12 A Stack



Display 17.13 Interface File for a Stack Template Class *(part 1 of 2)*

You might prefer to replace the parameter type T with const T&.

```
//This is the header file stack.h. This is the interface for the class
//Stack, which is a template class for a stack of items of type T.
#ifndef STACK_H
#define STACK_H

namespace StackSavitch
{
    template<class T>
    class Node
    {
    public:
        Node(T theData, Node<T>* theLink) : data(theData), link(theLink){}
        Node<T>* getLink( ) const { return link; }
        const T getData( ) const { return data; }
        void setData(const T& theData) { data = theData; }
        void setLink(Node<T>* pointer) { link = pointer; }
    private:
        T data;
        Node<T> *link;
    };

    template<class T>
    class Stack
    {
```

Display 17.13 Interface File for a Stack Template Class *(part 2 of 2)*

```
    public:
        Stack( );
        //Initializes the object to an empty stack.

        Stack(const Stack<T>& aStack);                Copy constructor

        Stack<T>& operator =(const Stack<T>& rightSide);

        virtual ~Stack( );                   The destructor destroys the stack and
                                             returns all the memory to the freestore.
        void push(T stackFrame);
        //Postcondition: stackFrame has been added to the stack.

        T pop( );
        //Precondition: The stack is not empty.
        //Returns the top stack frame and removes that top
        //stack frame from the stack.

        bool isEmpty( ) const;
        //Returns true if the stack is empty. Returns false otherwise.
    private:
        Node<T> *top;
    };

}//StackSavitch
#endif //STACK_H
```

operations you can perform on a stack: adding an 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`.

pushing

popping

The names `push` and `pop` derive from a particular 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 17.14 shows a simple program that illustrates how the `Stack` class is used. This program reads a line of text one character at a time and places the characters in a stack. The program then removes the characters one by one and writes them to the screen. Because data is removed from a

Display 17.14 **Program Using the Stack Template Class**

```
//Program to demonstrate use of the Stack template class.
#include <iostream>
#include "stack.h"
#include "stack.cpp"
using std::cin;
using std::cout;
using std::endl;
using StackSavitch::Stack;

int main( )
{
    char next, ans;

    do
    {
        Stack<char> s;
        cout << "Enter a line of text:\n";
        cin.get(next);
        while (next != '\n')
        {
            s.push(next);
            cin.get(next);
        }

        cout << "Written backward that is:\n";
        while ( ! s.isEmpty( ) )
            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 9. It discards input remaining on the line.

Sample Dialogue

```
Enter a line of text:
straw
Written backward that is:
warts
Again?(y/n): y
Enter a line of text:
I love C++
Written backward that is:
++C evol I
Again?(y/n): n
```

stack in the reverse of the order in which it enters the stack, the output shows the line written backward. We have #included the implementation of the Stack class in our application program, as we normally do with template classes. That means we cannot run or even compile our application program until we do the implementation of our Stack class template.

The definitions of the member functions for the template class Stack are given in the implementation file shown in Display 17.15. 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. The pointer top serves the same purpose as the pointer head did in our previous discussions of linked lists.

Push and Pop

Adding a data item to a stack data structure is referred to as *pushing* the data item onto the stack. Removing a data item from a stack is referred to as *popping* the item off the stack.

Display 17.15 Implementation of the Stack Template Class *(part 1 of 2)*

```
//This is the implementation file stack.cpp.
//This is the implementation of the template class Stack.
//The interface for the template class Stack is in the header file stack.h.

#include <iostream>
#include <cstdlib>
#include <cstddef>
#include "stack.h"
using std::cout;


namespace StackSavitch
{

   //Uses cstddef:
    template<class T>
    Stack<T>::Stack( ) : top(NULL)
    {
        //Intentionally empty
    }


    template<class T>
    Stack<T>::Stack(const Stack<T>& aStack)
        <The definition of the copy constructor is Self-Test Exercise 12.>
```

Display 17.15 Implementation of the Stack Template Class *(part 2 of 2)*

```
    template<class T>
    Stack<T>& Stack<T>::operator =(const Stack<T>& rightSide)
        <The definition of the overloaded assignment operator is Self-Test Exercise 13.>

    template<class T>
    Stack<T>::~Stack( )
    {
        T next;
        while (! isEmpty( ))
           next = pop( );//pop calls delete.
    }


    //Uses cstddef:
    template<class T>
    bool Stack<T>::isEmpty( ) const
    {
        return (top == NULL);
    }

    template<class T>
    void Stack<T>::push(T stackFrame)
       <The rest of the definition is Self-Test Exercise 11.>

    //Uses cstdlib and iostream:
    template<class T>
    T Stack<T>::pop( )
    {
        if (isEmpty( ))
        {
            cout << "Error: popping an empty stack.\n";
            exit(1);
        }

        T result = top->getData( );

        Node<T> *discard;
        discard = top;
        top = top->getLink( );

        delete discard;

        return result;
    }

}//StackSavitch
```

Writing the definition of the member function push is Self-Test Exercise 11. However, we have already given the algorithm for this task. The code for the push member function is essentially the same as the function headInsert shown in Display 17.11, except that in the member function push we use a pointer named top in place of a pointer named head.

empty stack

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 more complicated but does not use any techniques we have not already discussed. The details are left to Self-Test Exercise 12.

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 as follows:

```
T result = top->getData( );
```

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

```
top = top->getLink( );
```

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

```
delete discard;
```

destructor

Each node that is removed from the linked list by the member function pop has its memory recycled with a call to delete, so 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 for recycling.

Self-Test Exercises

11. Give the definition of the member function push of the template class Stack described in Displays 17.13 and 17.15.
12. Give the definition of the copy constructor for the template class Stack described in Displays 17.13 and 17.15.
13. Give the definition of the overloaded assignment operator for the template class Stack described in Displays 17.13 and 17.15.

Example A QUEUE TEMPLATE CLASS

queue

A stack is a last-in/first-out data structure. Another common data structure is a **queue**, which handles data in a *first-in/first-out* fashion. A queue can be implemented with a linked list in a manner similar to our implementation of the Stack template class. However, a queue needs a pointer at both the head of the list and at the end of the linked list, since action takes place in both locations. It is easier to remove a node from the head of a linked list than from the other end of the linked list. Therefore, our implementation will remove nodes from the head of the list (which we will now call the **front** of the list) and will add nodes to the other end of the list, which we will now call the **back** of the list (or the back of the queue).

front

back

The definition of the Queue template class is given in Display 17.16. A sample application that uses the class Queue is shown in Display 17.17. The definitions of the member functions are left as Self-Test Exercises (but remember that the answers are given at the end of the chapter should you have any problems filling in the details).

QUEUE

A queue is a first-in/first-out data structure; that is, the data items are removed from the queue in the same order that they were added to the queue.

Display 17.16 Interface File for a Queue Template Class *(part 1 of 2)*

```

//This is the header file queue.h. This is the interface for the class
//Queue, which is a template class for a queue of items of type T.
#ifndef QUEUE_H
#define QUEUE_H

namespace QueueSavitch
{
    template<class T>
    class Node
    {
    public:
        Node(T theData, Node<T>* theLink) : data(theData), link(theLink){}
        Node<T>* getLink( ) const { return link; }
        const T getData( ) const { return data; }
        void setData(const T& theData) { data = theData; }
        void setLink(Node<T>* pointer) { link = pointer; }
    private:
        T data;
```

This is the same definition of the template class Node that we gave for the stack interface in Display 17.13. See the tip "A Comment on Namespaces" for a discussion of this duplication.

Display 17.16 Interface File for a Queue Template Class *(part 2 of 2)*

```
        Node<T> *link;
    };

    template<class T>
    class Queue
    {
    public:
        Queue( );
        //Initializes the object to an empty queue.

        Queue(const Queue<T>& aQueue);

        Queue<T>& operator =(const Queue<T>& rightSide);

        virtual ~Queue( );


        void add(T item);
        //Postcondition: item has been added to the back of the queue.

        T remove( );
        //Precondition: The queue is not empty.
        //Returns the item at the front of the queue
        //and removes that item from the queue.

        bool isEmpty( ) const;
        //Returns true if the queue is empty. Returns false otherwise.
    private:
        Node<T> *front;//Points to the head of a linked list.
                       //Items are removed at the head
        Node<T> *back;//Points to the node at the other end of the linked list.
                      //Items are added at this end.
    };

}//QueueSavitch

#endif //QUEUE_H
```

You might prefer to replace the parameter type T with const T&.

Copy constructor

The destructor destroys the queue and returns all the memory to the freestore.

Display 17.17 Program Using the Queue Template Class *(part 1 of 2)*

```
//Program to demonstrate use of the Queue template class.
#include <iostream>
#include "queue.h"
#include "queue.cpp"
using std::cin;
using std::cout;
using std::endl;
using QueueSavitch::Queue;

int main( )
{
    char next, ans;

    do
    {
        Queue<char> q;
        cout << "Enter a line of text:\n";
        cin.get(next);
        while (next != '\n')
        {
            q.add(next);
            cin.get(next);
        }

        cout << "You entered:\n";
        while ( ! q.isEmpty( ) )
            cout << q.remove( );
        cout << endl;

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

    return 0;
}
```

Contrast this with the similar program using a stack instead of a queue that we gave in Display 17.14.

Display 17.17 Program Using the Queue Template Class *(part 2 of 2)*

SAMPLE DIALOGUE

```
Enter a line of text:
straw
You entered:
straw
Again?(y/n): y
Enter a line of text:
I love C++
You entered:
I love C++
Again?(y/n): n
```

Tip

A Comment on Namespaces

Notice that both of the namespaces StackSavitch (Display 17.13) and QueueSavitch (Display 17.16) define a template class called Node. As it turns out, the two definitions of Node are the same, but the point discussed here is the same whether the two definitions are the same or different. C++ does not allow you to define the same identifier twice, even if the two definitions are the same, unless the two names are somehow distinguished. In this case, the two definitions are allowed because they are in two different namespaces. It is even legal to use both the Stack template class and the Queue template class in the same program. However, you should use

```
using StackSavitch::Stack;
using QueueSavitch::Queue;
```

rather than

```
using namespace StackSavitch;
using namespace QueueSavitch;
```

Most compilers will allow either set of using directives if you do not use the identifier Node, but the second set of using directives provides two definitions of the identifier Node and therefore should be avoided.

It would be fine to also use either, but not both, of the following:

```
using StackSavitch::Node;
```

or

```
using QueueSavitch::Node;
```

Self-Test Exercises

14. Give the definitions for the default (zero-argument) constructor and the member functions `Queue<T>::isEmpty` for the template class `Queue` (Display 17.16).
15. Give the definitions for the member functions `Queue<T>::add` and `Queue<T>::remove` for the template class `Queue` (Display 17.16).
16. Give the definition for the destructor for the template class `Queue` (Display 17.16).
17. Give the definition for the copy constructor for the template class `Queue` (Display 17.16).
18. Give the definition for the overloaded assignment operator for the template class `Queue` (Display 17.16).

FRIEND CLASSES AND SIMILAR ALTERNATIVES

You may have found it a nuisance to use the accessor and mutator functions `getLink` and `setLink` in the template class `Node` (see Display 17.13 or Display 17.16). You might be tempted to avoid the invocations of `getLink` and `setLink` by simply making the member variable `link` of the class `Node` public instead of private. Before you abandon the principle of making all member variables private, note two things. First, using `getLink` and `setLink` is not really any harder for you the programmer than directly accessing the links in the nodes. (However, `getLink` and `setLink` do introduce some overhead and so may slightly reduce efficiency.) Second, there is a way to avoid using `getLink` and `setLink` and instead directly access the links of nodes without making the `link` member variable public. Let's explore this second possibility.

friend class

Chapter 8 discussed friend functions. As you will recall, if `f` is a friend function of a class `C`, then `f` is not a member function of `C`; however, when you write the definition of the function `f`, you can access private members of `C` just as you can in the definitions of member functions of `C`. A class can be a **friend** of another class in the same way that a function can be a friend of a class. If the class `F` is a friend of the class `C`, then every member function of the class `F` is a friend of the class `C`. Thus, if, for example, the `Queue` template class were a friend of the `Node` template class, then the private link member variables would be directly available in the definitions of the member functions of `Queue`. The details are outlined in Display 17.18.

forward declaration

When one class is a friend of another class, it is typical for the classes to reference each other in their class definitions. This requires that you include a **forward declaration** to the class or class template defined second, as illustrated in Display 17.18. Note that the forward declaration is just the heading of the class or class template definition followed by a semicolon. A complete example using a friend class is given in Section 17.4 (see "A Tree Template Class").

Display 17.18 A Queue Template Class as a Friend of the Node Class *(part 1 of 2)*

```

//This is the header file queue.h. This is the interface for the class
//Queue, which is a template class for a queue of items of type T.
#ifndef QUEUE_H
#define QUEUE_H

namespace QueueSavitch
{
    template<class T>
    class Queue;

    template<class T>
    class Node
    {
    public:
        Node(T theData, Node<T>* theLink) : data(theData), link(theLink){}
        friend class Queue<T>;
    private:
        T data;
        Node<T> *link;
    };

    template<class T>
    class Queue
    {
<The definition of the template class Queue is identical to the one given in Display 17.16. However, the
definitions of the member functions will be different from the ones we gave (in the Self-Test Exercises)
for the nonfriend version of Queue.>
}//QueueSavitch
#endif //QUEUE_H
```

A forward declaration. Do not forget the semicolon.

This is an alternate approach to that given in Display 17.16. In this version the Queue template class is a friend of the Node template class.

If Node<T> is only used in the definition of the friend class Queue<T>, there is no need for mutator or accessor functions.

```
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include "queue.h"
using std::cout;
namespace QueueSavitch
{
    template<class T> //Uses cstddef:
    void Queue<T>::add(T item)
    {
       if (isEmpty( ))
```

The implementation file would contain these definitions and the definitions of the other member functions similarly modified to allow access by name to the `link` and `data` member variables of the nodes.

Display 17.18 A Queue Template Class as a Friend of the Node Class *(part 2 of 2)*

```
                front = back = new Node<T>(item, NULL);
            else
            {
                back->link = new Node<T>(item, NULL);
                back = back->link;
            }
        }

        template<class T> //Uses cstdlib and iostream:
        T Queue<T>::remove( )
        {
            if (isEmpty( ))
            {
                cout << "Error: Removing an item from an empty queue.\n";
                exit(1);
            }

            T result = front->data;

            Node<T> *discard;
            discard = front;
            front = front->link;
            if (front == NULL) //if you removed the last node
                back = NULL;

            delete discard;
            return result;
        }
    }//QueueSavitch
```

If efficiency is a major issue, you might want to use `(front == NULL)` *instead of* `(isEmpty( ))`.

Contrast these implementations with the ones given as the answer to Self-Test Exercise 15.

Two approaches that serve pretty much the same purpose as friend classes and which can be used in pretty much the same way with classes and template classes such as `Node` and `Queue` are (1) using protected or private inheritance to derive `Queue` from `Node`, and (2) giving the definition of `Node` within the definition of `Queue`, so that `Node` is a local class (template) definition. (Protected inheritance is discussed in Chapter 14, and classes defined locally within a class are discussed in Chapter 7.)

17.3 Iterators

The white rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.
"Begin at the beginning," the King said, very gravely, "And go on till you come to the end: then stop."

Lewis Carroll, *Alice in Wonderland*

An important notion in data structures is that of an iterator. An **iterator** is a construct (typically an object of some iterator class) 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.

iterator

ITERATOR

An iterator is a construct (typically an object of some iterator class) 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 in the data structure.

POINTERS AS ITERATORS

The basic ideas, and in fact the prototypical model, for iterators can easily be seen in the context of linked lists. A linked list is one of the prototypical data structures, and a pointer is a prototypical example of an iterator. 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 *iterator;
for (iterator = Head; iterator != NULL; iterator = iterator->Link)
    Do whatever you want with the node pointed to by iterator;
```

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 discussed in Section 17.1, you could use the following:

```
IntNode *iterator;
for(iterator = head; iterator != NULL; iterator = iterator->getLink( ))
    cout << (iterator->getData( ));
```

The definition of `IntNode` is given in Display 17.4.

Note that you test to see if two pointers are pointing to the same node by comparing them with the equal operator, ==. A pointer is a memory address. If two pointer variables contain the same memory address, then they compare as equal and they point to the same node. Similarly, you can use `!=` to compare two pointers to see if they do not point to the same node.

ITERATOR CLASSES

iterator class

An **iterator class** is a more versatile and more general notion than a pointer. It very often does have a pointer member variable as the heart of its data, as in the next programming example, but that is not required. For example, the heart of the iterator might be an array index. An iterator class has functions and overloaded operators that allow you to use pointer syntax with objects of the iterator class no matter what you use for the underlying data structure, node type, or basic location marker (pointer or array index or whatever). Moreover, it provides a general framework that can be used across a wide range of data structures.

An iterator class typically has the following overloaded operators:

++	Overloaded increment operator, which advances the iterator to the next item.
--	Overloaded decrement operator, which moves the iterator to the previous item.
==	Overloaded equality operator to compare two iterators and return `true` if they both point to the same item.
!=	Overloaded not-equal operator to compare two iterators and return `true` if they do not point to the same item.
*	Overloaded dereferencing operator that gives access to one item. (Often it returns a reference to allow both read and write access.)

When thinking of this list of operators you can use a linked list as a concrete example. In that case, remember that the items in the list are the data in the list, not the entire nodes and not the pointer members of the nodes. Everything but the data items is implementation detail that is meant to be hidden from the programmer who uses the iterator and data structure classes.

An iterator is used in conjunction with some particular structure class that stores data items of some type. The data structure class normally has the following member functions that provide iterators for objects of that class:

`begin( )`: A member function that takes no argument and returns an iterator that is located at ("points to") the first item in the data structure.

`end( )`: A member function that takes no argument and returns an iterator that can be used to test for having cycled through all items in the data structure. If `i` is an iterator and `i` has been advanced *beyond* the last item in the data structure, then `i` should equal `end( )`.

Using an iterator, you can cycle through the items in a data structure ds as follows:

```
for (i = ds.begin( ); i != ds.end( ); i++)
    process *i //*i is the current data item.
```

where i is an iterator. Chapter 19 discusses iterators with a few more items and refinements than these, but these will do for an introduction.

This abstract discussion will not come alive until we give an example. So, let's walk through an example.

ITERATOR CLASS

An iterator class typically has the following overloaded operators: ++, move to next item; --, move to previous item; ==, overloaded equality; !=, overloaded not-equal operator; and *, overloaded dereferencing operator that gives access to one data item.

The data structure corresponding to an iterator class typically has the following two member functions: begin( ), which returns an iterator that is located at ("points to") the first item in the data structure; and end( ), which returns an iterator that can be used to test for having cycled through all items in the data structure. If i is an iterator and i has been advanced *beyond* the last item in the data structure, then i should equal end( ).

Using an iterator, you can cycle through the items in a data structure ds as follows:

```
for (i = ds.begin( ); i != ds.end( ); i++)
    process *i //*i is the current data item.
```

Example

AN ITERATOR CLASS

Display 17.19 contains the definition of an iterator class that can be used for data structures, such as a stack or queue, that are based on a linked list. We have placed the node class and the iterator class into a namespace of their own. This makes sense, since the iterator is intimately related to the node class and since any class that uses this node class can also use the iterator class. This iterator class does not have a decrement operator, because a definition of a decrement operator depends on the details of the linked list and does not depend solely on the type Node<T>. (There is nothing wrong with having the definition of the iterator depend on the underlying linked list. We have just decided to avoid this complication.)

As you can see, the template class ListIterator is essentially a pointer wrapped in a class so that it can have the needed member operators. The definitions of the overload operators are straightforward and in fact so short that we have defined all of them as inline functions. Note that

Display 17.19 An Iterator Class for Linked Lists *(part 1 of 2)*

```
//This is the header file iterator.h. This is the interface for the class
//ListIterator, which is a template class for an iterator to use with linked
//lists of items of type T. This file also contains the node type for a
//linked list.
#ifndef ITERATOR_H
#define ITERATOR_H

namespace ListNodeSavitch
{
    template<class T>
    class Node
    {
    public:
        Node(T theData, Node<T>* theLink) : data(theData), link(theLink){}
        Node<T>* getLink( ) const { return link; }
        const T getData( ) const { return data; }
        void setData(const T& theData) { data = theData; }
        void setLink(Node<T>* pointer) { link = pointer; }
    private:
        T data;
        Node<T> *link;
    };

    template<class T>
    class ListIterator
    {
    public:
        ListIterator( ) : current(NULL) {}

        ListIterator(Node<T>* initial) : current(initial) {}

        const T operator *( ) const { return current->getData( ); }
        //Precondition: Not equal to the default constructor object;
        //that is, current != NULL.

        ListIterator operator ++( ) //Prefix form
        {
            current = current->getLink( );
            return *this;
        }
        ListIterator operator ++(int) //Postfix form
        {
            ListIterator startVersion(current);
            current = current->getLink( );
```

*Note that the dereferencing operator * produces the data member of the node, not the entire node. This version does not allow you to change the data in the node.*

Display 17.19 An Iterator Class for Linked Lists *(part 2 of 2)*

```
                return startVersion;
            }
            bool operator ==(const ListIterator& rightSide) const
            { return (current == rightSide.current); }

            bool operator !=(const ListIterator& rightSide) const
            { return (current != rightSide.current); }

            //The default assignment operator and copy constructor
            //should work correctly for ListIterator.
        private:
            Node<T> *current;
        };

    }//ListNodeSavitch

    #endif //ITERATOR_H
```

the dereferencing operator, *, produces the data member variable of the node pointed to. Only the data member variable is data. The pointer member variable in a node is part of the implementation detail that the user programmer should not need to be concerned with.

You can use the ListIterator class as an iterator for any class based on a linked list that uses the template class Node. As an example, we have rewritten the template class Queue so that it has iterator facilities. The interface for the template class Queue is given in Display 17.20. This definition of the Queue template is the same as our previous version (Display 17.16) except that we have added a type definition as well as the following two member functions:

```
Iterator begin( ) const { return Iterator(front); }
Iterator end( ) const { return Iterator( ); }
//The end iterator has end( ).current == NULL.
```

Let's discuss the member functions first.

The member function begin( ) returns an iterator located at ("pointing to") the front node of the queue, which is the head node of the underlying linked list. Each application of the increment operator, ++, moves the iterator to the next node. Thus, you can move through the nodes, and hence the data, in a queue named q as follows:

```
for (i = q.begin( ); Stopping_Condition; i++)
    process *i //*i is the current data item.
```

Display 17.20 Interface File for a Queue with Iterators Template Class

```
//This is the header file queue.h. This is the interface for the class
//Queue, which is a template class for a queue of items of type T, including
//iterators.
#ifndef QUEUE_H
#define QUEUE_H
#include "iterator.h"
using namespace ListNodeSavitch;

namespace QueueSavitch
{
    template<class T>
    class Queue
    {
    public:
        typedef ListIterator<T> Iterator;

        Queue( );
        Queue(const Queue<T>& aQueue);
        Queue<T>& operator =(const Queue<T>& rightSide);
        virtual ~Queue( );
        void add(T item);
        T remove( );
        bool isEmpty( ) const;

        Iterator begin( ) const { return Iterator(front); }
        Iterator end( ) const { return Iterator( ); }
        //The end iterator has end( ).current == NULL.
        //Note that you cannot dereference the end iterator.
    private:
        Node<T> *front;//Points to the head of a linked list.
                       //Items are removed at the head
        Node<T> *back;//Points to the node at the other end of the linked
                      //list.
                      //Items are added at this end.
    };

}//QueueSavitch

#endif //QUEUE_H
```

The definitions of `Node<T>` and `ListIterator<T>` are in the namespace `ListNodeSavitch` in the file `iterator.h`.

where i is a variable of the iterator type.

The member function end( ) returns an iterator whose current member variable is NULL. Thus, when the iterator i has passed the last node, the Boolean expression

```
i != q.end( )
```

changes from true to false. This is the desired *Stopping_Condition*. This queue class and iterator class allow you to cycle through the data in the queue in the way we outlined for an iterator:

```
for (i = q.begin( ); i != q.end( ); i++)
    process *i //*i is the current data item.
```

Note that i is not equal to q.end( ) when i is at the last node. The iterator i is not equal to q.end( ) until i has been advanced one position past the last node. To remember this detail, think of q.end( ) as being an end marker like NULL; in this case it is essentially a version of NULL. A sample program that uses such a for loop is shown in Display 17.21.

typedef

Notice the type definition in our new queue template class:

```
typedef ListIterator<T> Iterator;
```

This typedef is not absolutely necessary. You can always use ListIterator<T> instead of the type name Iterator. However, this type definition does make for cleaner code. With this type definition, an iterator for the class Queue<char> is written

```
Queue<char>::Iterator i;
```

This makes it clear with which class the iterator is meant to be used.

The implementation of our new template class Queue is given in Display 17.22. Since the only member functions we added to this new Queue class are defined inline, the implementation file contains nothing really new, but we include the implementation file to show how it is laid out and to show which directives it would include.

Display 17.21 Program Using the Queue Template Class with Iterators *(part 1 of 2)*

```
//Program to demonstrate use of the Queue template class with iterators.
#include <iostream>
#include "queue.h"//not needed
#include "queue.cpp"
#include "iterator.h"//not needed
using std::cin;
using std::cout;
using std::endl;
using namespace QueueSavitch;

int main( )
{
    char next, ans;
    do
    {
        Queue<char> q;
        cout << "Enter a line of text:\n";
        cin.get(next);
        while (next != '\n')
        {
            q.add(next);
            cin.get(next);
        }

        cout << "You entered:\n";
        Queue<char>::Iterator i;

        for (i = q.begin( ); i != q.end( ); i++)
            cout << *i;
        cout << endl;

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

    return 0;
}
```

Even though they are not needed, many programmers prefer to include these `include` directives for the sake of documentation.

If your compiler is unhappy with `Queue<char>::Iterator i;` try `using namespace ListNodeSavitch;` `ListIterator<char> i;`

Display 17.21 Program Using the Queue Template Class with Iterators *(part 2 of 2)*

SAMPLE DIALOGUE

```
Enter a line of text:
Where shall I begin?
You entered:
Where shall I begin?
Again?(y/n): y
Enter a line of text:
Begin at the beginning
You entered:
Begin at the beginning
Again?(y/n): n
```

Display 17.22 Implementation File for a Queue with Iterators Template Class *(part 1 of 2)*

```
//This is the file queue.cpp. This is the implementation of the template
//class Queue. The interface for the template class Queue is in the header
//file queue.h.
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include "queue.h"
using std::cout;

using namespace ListNodeSavitch;
namespace QueueSavitch
{
    template<class T>
    Queue<T>::Queue( ) : front(NULL), back(NULL)
        <The rest of the definition is given in the answer to Self-Test Exercise 14.>

    template<class T>
    Queue<T>::Queue(const Queue<T>& aQueue)
        <The rest of the definition is given in the answer to Self-Test Exercise 17.>

    template<class T>
    Queue<T>& Queue<T>::operator =(const Queue<T>& rightSide)
        <The rest of the definition is given in the answer to Self-Test Exercise 18.>
```

The member function definitions are the same as in the previous version of the Queue template. This is given to show the file layout and use of namespaces.

Display 17.22 Implementation File for a Queue with Iterators Template Class *(part 2 of 2)*

```
    template<class T>
    Queue<T>::~Queue( )
        <The rest of the definition is given in the answer to Self-Test Exercise 16.>

    template<class T>
    bool Queue<T>::isEmpty( ) const
        <The rest of the definition is given in the answer to Self-Test Exercise 14.>

    template<class T>
    void Queue<T>::add(T item)
        <The rest of the definition is given in the answer to Self-Test Exercise 15.>

    template<class T>
    T Queue<T>::remove( )
       <The rest of the definition is given in the answer to Self-Test Exercise 15.>
}//QueueSavitch
#endif //QUEUE_H
```

Self-Test Exercises

19. Write the definition of the template function `inQ` shown below. Use iterators. Use the definition of `Queue` given in Display 17.20.

```
template<class T>
bool inQ(Queue<T> q, T target);
//Returns true if target is in the queue q;
//otherwise, returns false.
```

17.4 Trees

I think that I shall never see a data structure as useful as a tree.

Anonymous

A detailed treatment of trees is beyond the scope of this chapter. The goal of this chapter is to teach you the basic techniques for constructing and manipulating data structures based on nodes and pointers. The linked list served as a good example for our discussion. However, there is one detail about the nodes in a linked list that is quite restricted: They have only one pointer member variable to point to another node. A