

UCLA PIC 10 B

Problem Solving using C++ Programming

- **Instructor:** Ivo Dinov, Asst. Prof. in Mathematics, Neurology, Statistics
 - **Teaching Assistant:** Suzanne Nezzar, Mathematics
- University of California, Los Angeles, Summer 2001
<http://www.math.ucla.edu/~dinov/10b.1.011/>

1


Review, Wednesday, July 25, 2001 PIC 10 B

- Function Templates:
- Class templates:
- Shape template class for drawable N-tagons
- Matrix template class

2

Chapter 14

Pointers and Linked Lists

pnt →  dynamic array

Basic Problem:

Suppose we a data structure which is truly dynamic and changes extremely often and we do not know how many elements could this structure have even at run-time. E.g., An alphabetical ordering of all people affiliated with UCLA at any one time. This is extremely dynamic database. People come and go, still we need to be able to obtain, order and process individual's data (say payroll, enrollment, etc.) How should we design and implement such a complex data structure?

3

14 Pointers and Linked Lists

- Nodes and Linked Lists
 - Nodes
 - Linked Lists
 - Inserting a Node at the Head of a List
 - Searching a Linked List
 - Inserting and Removing Nodes Inside a List
- A Linked List Application
 - Stacks

4

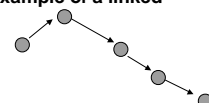
14 Pointers and Linked Lists - Notes

- Useful dynamically allocated variables are normally of complex type struct or class.
- We have already seen that dynamic variables are useful in solution to some problems. But these constructs are dynamic only at the time of memory allocation. As soon as memory is allocated dynamic objects become "static", well almost

5

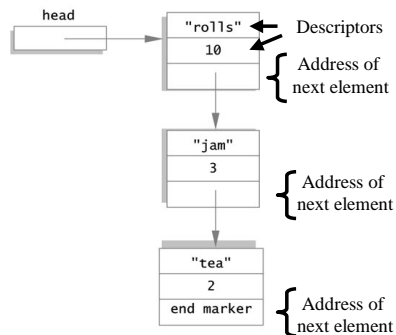
14.1 Nodes and Linked Lists

- A linked list is a list constructed of objects of type structs or classes with pointer(s) and data.
- The objects in the list are called nodes.
- The nodes are dynamically allocated and connected as the program requires (throughout run-time).
- The pointers enable connecting the nodes and traversal of the linked list.
- Display 14.1 provides an example of a linked (shopping) list.



6

Display 14.1 Nodes and Pointers



Nodes (1 of 6)

- Display 14.1 has boxes which we have connected by arrows.
- The boxes are called **nodes** and the arrows represent pointers.
- The value of the pointer is the **address of the next node**. Thus the pointer points to an **entire node**, not to the individual items ("rolls" or 10) stored in the node.
- Nodes may be implemented as struct or class objects.
- The nodes in Display 14.1 may be defined as:


```
const int STRING_SIZE = 10;
struct ListNode
{
    char item[STRING_SIZE];
    int count;
    ListNode *link;
};
typedef ListNode * ListNodePtr;
```
- Order of definitions is important: the definition of struct ListNode **must precede** the type definition.

8

Nodes (2 of 6)

- How do we get to the list?
- In Display 14.1, the box labeled head is not a node, rather it is a pointer variable of type allowing it to point to a node. It **points to the start of the list**. It may be declared by:


```
ListNodePtr head;
```
- We observe that the **definition** of ListNode is inductive (recursive).
- The trouble with recursive definitions is **when things do not stop**.
- This is a **legal circularity**.
- An indication that this is a legal circularity is the fact that we can draw pictures of objects that are defined.
- In Display 14.1 and in the running program:
 - There is a pointer variable, head, that points to a node
 - that contains a pointer that points to another node
 - that contains a pointer that points to another node and so on.
 - Things stop when a link node contains a NULL pointer.

9

Nodes (3 of 6)

- How do we access data in a node?
- The pointer variable **head** points to the first node. We illustrate:
- The variable head **points to the first node**, so ...
- The expression ***head is the first node**, i.e., a dynamic variable, so ...
- The expression ***head** can be used to access members using the dot operator:


```
(*head).count = 12;
```
- Why the parentheses? The short answer is:
- The dot operator **.** has higher precedence than the dereference operator *****.
- The long answer is:
 - In the table in Appendix 2, on Page 913, we see that the dot operator **.** is a postfix operator (second box in the table). These operators have higher precedence than the prefix operators (in the third box in the table), including the ***** operator.

10

Nodes (4 of 6)

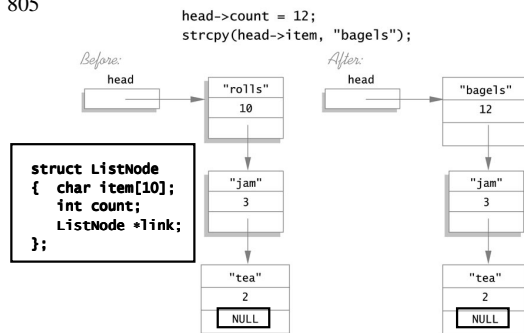
- C++ provides the arrow operator, **->** which is composed of the minus (-) followed by the less than (<) **to combine the effect of the * and . operators**:


```
head->count = 12; // EQ. (*head).count=12;
```
- Spaces are usually placed around operators such as +, /, == and =, but with **->** this is not usually done.
- The expression **head->count** may be used to **fetch data** from or **assign data** to the **count** member of the dynamic variable pointed to by head.
- This works the same for any member of a struct or class object pointed to by a pointer variable.
- Display 14.2 uses the predefined (library) function **strcpy** declared in the **<cstring>** header to copy "bagels" into the item member:


```
strcpy(head->item, "bagels");
```
- Remember that **= does not work for cstrings**. You must use **strcpy** instead.

11

Display 14.2 Accessing Node Data



Nodes (5 of 6)

- We said that inductive (circular) reference was OK as long as things terminate.
- Remember that recursion has base cases that must be guaranteed to be reached.
- Notice the list pointer member in the last node in Display 14.2.
- We do not see a pointer, rather we see NULL.
- In Display 14.1 this is written as the non-C++ "end marker", for that is what NULL does. NULL signals code that traverses the list that this is the end of the list.
- We introduced NULL in Chapter 11 where we noted that NULL is defined as the `const int 0`. Our author and many other writers prefer that you write this NULL rather than 0.

13

806

The Arrow Operator ->

The arrow operator `->` specifies a member of a *struct* (or a member of a class object) that is pointed to by a pointer variable. The syntax is:

Pointer_Variable -> *Member_Name*

The above refers to a member of the *struct* or object pointed to by the *Pointer_Variable*. Which member it refers to is given by the *Member_Name*. For example, suppose you have the following definition:

```
struct Record
{
    int number;
    char grade;
};
```

The following creates a dynamic variable of type *Record* and sets the member variables of the dynamic *struct* variable to 2001 and 'A':

```
Record *p;
p = new Record;
p->number = 2001;
p->grade = 'A';
```

14

Nodes (6 of 6)

- NULL is used for a number of purposes.
- NULL is defined as a pointer value that is different from any other pointer value.
- NULL is used to assign a value to a pointer that otherwise would not have a value (i.e. would have a garbage value left by the previous user of that memory location.)
- NULL is used to signal the end of a linked list.
- Technically:
 - A pointer has the null pointer value if it compares true (`==`) to the `int const 0`.
 - If a pointer is assigned the value of an expression that has the value 0, C++ converts the `int const 0` to the null pointer value that is appropriate for the system.
- It is illegal to dereference a pointer that has the NULL pointer value.
- "NULL pointer dereferenced" tends to be a difficult error to find and fix.

15

807

NULL

NULL is a special constant pointer value that is used to give a value to a pointer variable that would not otherwise have a value. NULL can be assigned to a pointer variable of any type. The identifier NULL is defined in a number of libraries including the library with header file `cstddef`. With earlier compilers, the operator `new` returned a NULL pointer value whenever `new` failed in its attempt to create a dynamic variable. Current compilers "throw the exception `std::bad_alloc`." The effect is to abort the program with an error message.

16

Linked Lists(1 of 2)

- A linked list is a list of nodes where each node has a member variable that is a pointer that points to the next node in the list. (Let's call this member *link*.)
- The first node is called the head.
- There is a simple pointer variable that points to the head called the head pointer.
- The last node is not named, but the link member of the last node has the NULL pointer value.

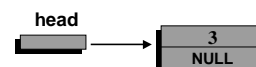
17

Linked Lists(2 of 2)

- A simplified node struct and a pointer type definition:


```
struct Node
{
    int data;
    Node *link;
};
typedef Node * NodePtr;
```
- We construct the start of a linked list of nodes of this type:


```
NodePtr head; // only a pointer so far
head = new Node; // allocate a node
head->data = 3; // set the data member
head->link = NULL; // set the link field to make
// this the end of the list.
```



18

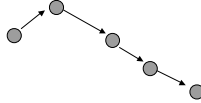
Review, Thursday, July 26, 2001 PIC 10 B

Started Ch. 14, linked-lists:

```
struct ListNode
{
    char item[10];
    int count;
    ListNode *link;    // recursive definition?!?
};

ListNode * head;
head = new ListNode;

strcpy(head->item, "bagels"); // vs.
strcpy((*head).item, "bagels");
```



19

Inserting a Node at the Head of a List (1 of 3)

- This section assumes that we have a linked list that contains several nodes.
- We write a function having two parameters: a node pointer that points to the head of a list, and an int value for data.
- This function inserts a node as the new head of the list with the int value in the data field.

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

Linked List 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 (that points to the head of the linked list) can be used as the linked list argument.

```
struct Node
{
    int data;
    Node * link;
};
typedef Node* NodePtr;
```

20

Inserting a Node at the Head of a List (2 of 3)

- Pseudo code for **head_insert** function
 - Create a new dynamic variable of type Node pointed to by **temp_ptr**. (This is the new node. It can be referred to by ***temp_ptr**)
 - Assign the **data** member of the new node the new data.
 - Make the **link** member of the new node point to the head node.
 - Make the pointer variable **head** point to the new node.
- Display 14.3 has a diagram of this algorithm.
- Steps 2 and 3 are expressed by the C++ code:


```
temp_ptr->link = head;
head = temp_ptr;
```
- Display 14.4 has the complete code.

```
1. T = new Node;
2. T->data=5;
3. T->link=head;
4. Head=T;
```

21

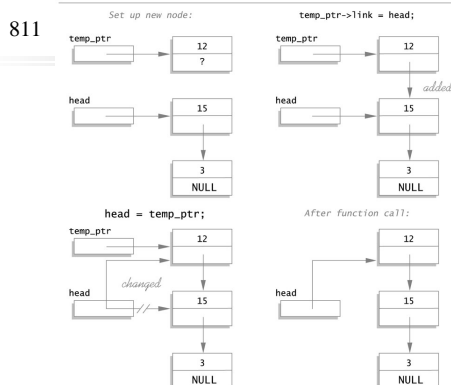
Inserting a Node at the Head of a List (3 of 3)

- A list with nothing in it is called an empty list.
- A list is named by the pointer that points to the head of the list but an empty list does not have any nodes at all.
- To specify an empty list, the head pointer has the NULL pointer value.
- Create an empty list by declaring a head pointer initialized with the pointer the value NULL.


```
NodePtr empty_head = NULL;
```
- When designing a function to manipulate a linked list, always check to see if it works on the empty list.
- Display 14.4 was designed with the non-empty list as model.
- You should check that it works for an empty list as well.

22

Display 14.3 Adding a Node to a Linked List



23

Display 14.4 Function to Add a Node at the Head of a Linked List (1 of 2) Prototype:

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

```
typedef Node* NodePtr;
```

```
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.
```

24

Display 14.4 Function to Add a Node at the Head of a Linked List (2 of 2)

Function Definition:

```
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;
    temp_ptr->data = the_number;
    temp_ptr->link = head;          //!!!!!! See next slide
    head = temp_ptr;
    //NOTE: that temp_ptr is deleted as soon as head_insert completes ...
    // But, before that in addr(head) we have the temp_ptr node saved ...
}
```

25

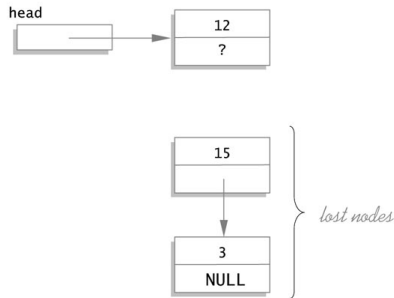
Pitfall: Losing Nodes

- If you write the assignment statements in your version of **head_insert**, you will chop off the end of your list and lose it. Display 14.5 shows this situation.
- Do NOT assign the head pointer before assigning the link:

```
head = new Node;          // temp_ptr = new Node;
head->data = the_number;  // temp_ptr->data = the_number;
head->link = head;        // head = temp_ptr;
                          // temp_ptr->link = head;
                          //loses the rest of the list
```
- The link pointer points to the node, not to the rest of the list.
- The rest of the list is an orphan. Your program has no way to access the rest of the list.
- A carefully drawn sketch is worth a thousand lines of code.

26

814 Display 14.5 Lost Nodes



27


Searching a Linked List(1 of 5)

- We use the same node type we have seen.

```
struct Node
{ int data;
  Node * link;
};
typedef Node* NodePtr;
```
- The function search will have two arguments:
 - **head**, of type **NodePtr**, that points to the head of a list to be searched.
 - **target**, an **int** that is the value being sought.
- If **target** is present, the function returns a pointer to the first node that contains **target**, otherwise the function returns the **NULL** pointer value.

28

Searching a Linked List(2 of 5)

- We use a local pointer, **here**, to step through the list in the search.
- To step through the list we must follow the pointers.
- We start by assigning our local pointer, **here**, the value of the head pointer. The details are in Display 14.6.
- The Prototype is: 

```
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.
```

29

Searching a Linked List(3 of 5)

- Pseudocode for search:

Make the local pointer variable, here, point to the head node of the linked list.

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;

30

Searching a Linked List(4 of 5)

- To move a pointer in the list, we must use the pointers available.
- The pointer to the node after here is `here->link`
- To move here to the next node after the node here points to, we make the assignment:
`here = here->link;`
- We have this preliminary version of the search function body:

```
here = head;
while(here->data != target && here->link != NULL)
    here = here->link;

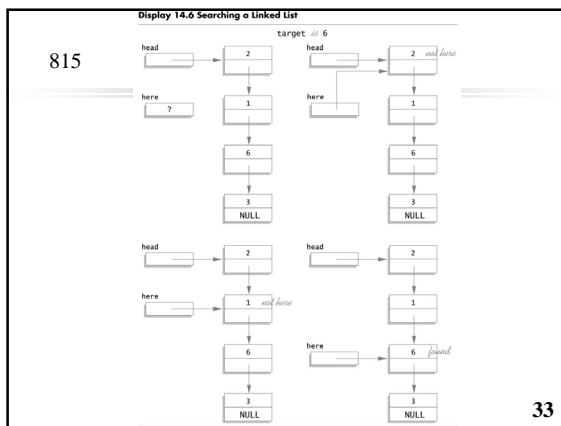
if (here->data == target)    return here;
else    return NULL;
```

31

Searching a Linked List(5 of 5)

- We still must handle the special case of an empty list.
- The preliminary code fails for an empty list.
- The error is caused by a null pointer dereference.
- When we search an empty list, the local pointer variable here is assigned from the parameter head which has the null pointer value.
- This makes the following expressions are illegal:
`here->data`
and
`here->link`
- because both are null pointer dereference errors.
- Display 14.7 presents the complete function definition.

32



Display 14.7 Function to Locate a Node in a linked list (1 of 2)

Prototypes:

```
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.
```

34

Display 14.7 Function to Locate a Node in a linked list (2 of 2)

Function Definition:

```
//Uses cstddef:
NodePtr search(NodePtr head, int target)
{
    NodePtr here = head;
    if (here == NULL)
    {
        return NULL;
    }
    else
    {
        while (here->data != target && here->link != NULL)
            here = here->link;
        if (here->data == target)
            return here;
        else
            return NULL;
    }
}
```

Empty list case

35

Inserting and Removing Nodes Inside a List (1 of 2)

- There are many reasons for inserting into a list other than at the ends. E.g., maintaining a list in some order will necessitate insertion in the middle.
- We design a function to insert a node after a specified node in a linked list (named appropriately) insert.
- The function returns `void`, and takes two parameters:
 - a type `NodePtr` parameter `after_me` that specifies the node after which the new node is to be inserted.
 - an `int` parameter `the_number` that is used to initialize the data member of the new node.

36

Inserting and Removing Nodes Inside a List (2 of 2)

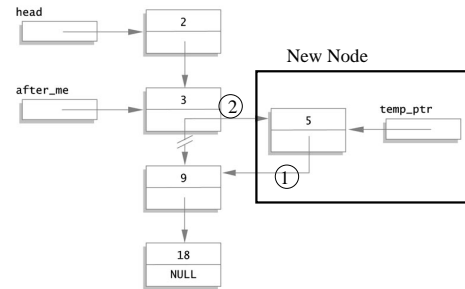
- The prototype and implementation for insert are given in Display 14.9.
- The initial setup is similar to head_insert.
- The difference is that now we want to insert not at the head but after the head.
- Display 14.8 presents a sketch of the insertion process.
- Here is C++ code:

```
// 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;
```

37

819

Display 14.8 Inserting in the Middle of a Linked List



Display 14.9 Function to Add a Node in the Middle of
a linked list (1 of 2)

```
Prototypes:
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.
```

39

Review, Monday, July 30, 2001 PIC 10 B

- Linked-lists:

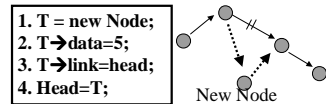
```
struct Node
{ int data;
  Node *link;
};

typedef Node* NodePtr;

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

NodePtr search(NodePtr head, int target);

void insert(NodePtr after_me, int the_number); // Just Started
//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. 40
```



Display 14.9 Function to Add a Node in the Middle of
a linked list (2 of 2)

```
Function Definitions:

//Uses cstddef:
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; } // Order of Operations
// is important!!!

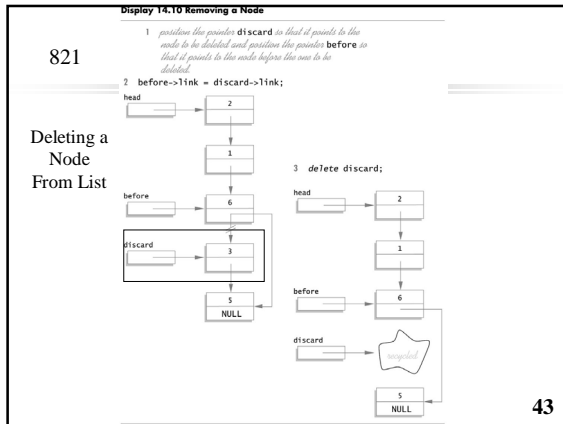
41
```

Pitfall: Using 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, hence the same linked list.

```
head2 = head1;
```
- You must remember that *there is only one* linked list, not two. If you change the linked list pointed to by head1 you are changing the linked list pointed to by head2, *since they are the same list*.
- To get a separate list you have two choices:
 - you can copy the entire list, node by node, or
 - you can overload assignment, operator =, to do whatever you want. See Chapter 11, optional subsection "Overloading the Assignment Operator", for data type: NodePtr.
- This is an example of the "aliasing" problem, where we have two variables pointing to one object.

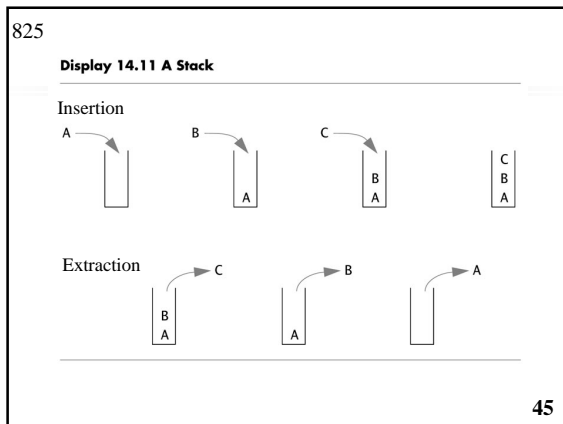
42



14.2 A Linked List Application Stacks

- A stack stores information in a way that the last item stored is the first item available when information is examined, LIFO.
- The only data in a stack that is accessible is that data on top.
- A consequence is the stack retrieves information in reverse order in which the data was stored.
- Display 14.11 diagrams the use of a stack.
- Stacks are useful in language processing tasks, and for keeping track of function calls.
- Our application will be simple example to show a use of linked lists.

44



Programming Example: A Stack ADT (1 of 2)

- The interface for the stack ADT is given in Display 14.12
- This stack stores data of type **char**.
- The basic operations on a stack are **push** and **pop**
 - Push places data on the top of the stack (insertion)
 - Pop returns the data on the top of the stack and removes it (extraction).
- Display 14.13 is a simple program illustrating use of the stack ADT.
- Display 14.14 is the implementation of our stack ADT.

46

Programming Example: A Stack ADT (2 of 2)

- The implementation of **push** is left to Self-Test Exercise 10.
- The default constructor creates an empty stack.
- The copy constructor is left to Self-Test Exercise 11.
- The **pop** member function checks for empty stack, if the stack is not empty, it returns the data from the top and removes that data. If the stack is empty, **pop** issues an error message and terminates the program.
- Each node removed by the **pop** is destroyed with a call to **delete**.
- The (total stack) destructor has only to traverse the list, calling **pop** until the stack is empty.

47

Display 14.12 Interface File for a Stack Class (1 of 2)

```
// stack.h -
// INTERFACE for class Stack, an ADT for a stack of symbols.
#ifndef STACK_H
#define STACK_H

namespace savitchstack
{
    struct StackFrame
    {
        char data;
        StackFrame *link;
    };

    typedef StackFrame* StackFramePtr;
```

48

Display 14.12 Interface File for a Stack Class (2 of 2)

```
// stack.h -- INTERFACE for class Stack, an ADT for a stack of symbols.
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 heap.
    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 stack symbol and removes that top symbol.
    bool empty() const; //Returns TRUE if the stack is empty.
    StackFramePtr getTop();
private:
    StackFramePtr top;
};
} //savitchstack
#endif //STACK_H
```

49

Display 14.13 Program Using the Stack ADT (1 of 2)

```
#include <iostream>
#include "stack.h"
using namespace std;
using namespace savitchstack;

int main()
{ Stack s;
  char next, ans;
  do
  { cout << "Enter a word: ";
    cin.get(next);
    while (next != '\n')
    { s.push(next);
      cin.get(next);
    }
  }
```

50

Display 14.13 Program Using the Stack ADT (2 of 2)

```
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;
}
```

51

Display 14.14 Implementation of the Stack ADT (1 of 2)

```
// FILE stack.cxx. IMPLEMENTATION of the driver class Stack.
//The interface for the class Stack is in the header file stack.h.
#include <iostream>
#include <cstackdef>
#include "stack.h"
using namespace std;
namespace savitchstack
{
    //Uses cstackdef:
    Stack::Stack()
    {
        top = NULL;
    }
}
```

Display 14.14 Implementation of the Stack ADT (2 of 2)

```
Stack::Stack(const Stack& a_stack) // Uses cstackdef
{ if (a_stack.getTop() == NULL) // copy constructor
    top = NULL;
else
{ StackFramePtr temp = a_stack.getTop(); // 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;
}
}
```

Display 14.14 Implementation of the Stack ADT (3 of 2)

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

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

//Uses cstackdef:
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;
}
```

54

Display 14.14 Implementation of the Stack ADT (4 of)

```
//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;
}
//savitchstack
```

```
// Standard class LIST (list.h)
#ifndef LIST_H
#define LIST_H

#include <functional>
#include <algorithm>
#include <iterator>
#include <climits>
#include "memman.h"

#ifdef __ITERATOR_NEEDED
namespace std {
template <class _Category, class _Tp, class _Distance = ptrdiff_t,
        class _Pointer = _Tp*, class _Reference = _Tp&>
struct iterator {
    typedef _Category iterator_category;
    typedef _Tp value_type;
    typedef _Distance difference_type;
    typedef _Pointer pointer;
    typedef _Reference reference;
};
};
#endif
```

```
// Standard class LIST (list.h)

// Open a new namespace, to avoid conflict with std::list
namespace list_presentation {

using namespace std;

template <class T>
class list {

protected:
    // Representation (doubly linked):

    struct list_node {
        list_node* next;
        list_node* prev;
        T data;
    };
};
```

```
// Standard class LIST (list.h)

public: // Types:
    typedef T value_type;
    typedef T& reference;
    typedef const T& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

private:
    typedef T* pointer;
    typedef list_node* link_type;
    typedef list_memory_manager<list_node>
        list_memory_manager_type;

protected:
    link_type node;
    size_type length;
    static size_type number_of_lists;
    static list_memory_manager_type manager;
```

```
// Standard class LIST (list.h)

public:
    // Iterator types (defined by nested classes and typedefs)
    // class iterator;
    // class const_iterator;
    // class reverse_iterator;
    // class const_reverse_iterator;
    #include "listiter.h"

    // Constructors:
    list();
    list(size_type n, const T& value = T());
    list(const T* first, const T* last);
    list(const list<T>& x);

    // Destructor:
    ~list();
```

```
// Standard class LIST (list.h)

// Assignment and swap:
list<T>& operator=(const list<T>& x);
void swap(list<T>& x);

// Queries:
iterator begin() { return node->next; }
iterator end() { return node; }
bool empty() const { return length == 0; }
size_type size() const { return length; }
size_type max_size() const {
    return max(size_type(1),
        size_type(UINT_MAX/sizeof(T)));
}
reference front() { return *begin(); }
reference back() { return *(--end()); }
```

```
// Standard class LIST (list.h)

// Queries for reverse traversal:
reverse_iterator rbegin() { return reverse_iterator(end()); }
reverse_iterator rend() { return reverse_iterator(begin()); }

// Insertion:
iterator insert(iterator position, const T& x);
void insert(iterator position, const T* first, const T* last);
void insert(iterator position, const_iterator first,
            const_iterator last);
void insert(iterator position, size_type n, const T& x);
void push_front(const T& x) { insert(begin(), x); }
void push_back(const T& x) { insert(end(), x); }
```

```
// Standard class LIST (list.h)

// Erasure (deletion):
void erase(iterator position);
void erase(iterator first, iterator last);
void pop_front() { erase(begin()); }
void pop_back() { iterator tmp = end(); erase(--tmp); }

protected: // Auxiliary function for implementing splice functions:
void transfer(iterator position, iterator first, iterator last);

public:
// Splicing elements or ranges from one list to another (or same) list:
void splice(iterator position, list<T>& x);
void splice(iterator position, list<T>& x, iterator i);
void splice(iterator position, list<T>& x, iterator first, iterator last);

void remove(const T& value); // Remove all occurrences of a value:
void unique(); // Remove all consecutive duplicate values:

void merge(list<T>& x); // Merge a list with the current list:
```

```
// Standard class LIST (list.h)

void reverse(); // Reverse the current list:
void sort(); // Sort the current list:
}; // end class list;

// Equality and less-than operations:
template <class T>
inline bool operator==(const list<T>& x, const list<T>& y) {
    return x.size() == y.size() && equal(x.begin(), x.end(), y.begin()); }

template <class T>
inline bool operator<(const list<T>& x, const list<T>& y) {
    return lexicographical_compare(x.begin(), x.end(), y.begin(),
                                   y.end()); }

#include "list.C"
}; // close namespace list_presentation
#endif
```

830

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 is indicated by setting the pointer member variable equal to NULL.

64