

- Pointer variables
- Operations on pointer variables
- Linked lists
- Operations on linked lists
- Variations on simple linked lists
  - doubly linked lists
  - circular linked lists



## **Pointer Variables**

- Declaring a variable creates space for it
  - in a region of process memory called *stack*
  - each memory cell has an address
    - memory can be considered to be linearly addressed starting from 0 to MAX

int var = 268;

var



• Use pointers to refer to variables *indirectly* by *pointing at them* 



- A pointer contains the location, or address in memory, of a memory cell
- Declaration of an integer pointer variable p
  - static allocation; initially undefined, but not NULL
     int var = 268;

int \*p;





• Can assign address of any variable (including another pointer variable) to the pointer variable



Indirect updates through pointer variables





- & : address-of operator
- \* : used for "de-reference" a pointer
  - expression \*p represents the memory cell to which p points
- Pointer variables are also variables!
  - need space in memory
  - can have pointer variables pointing to other pointer variables

```
int a, *p, **pp;
p = &a;
pp = &p;
```



## Pointer Variable – Types

- All pointer variables hold integer addresses, but have types
  - very important during pointer arithmetic

char c, \*cp = &c;

ip ++; // increments value in 'ip' by 4/8
cp ++; // increments value in 'cp' by 1
pp = &a; // Is this valid ?

 Multiple/divide with pointer variables generally is not meaningful



## **New Operator**

- All declared variables, arrays are statically assigned space (on the stack) by the compiler
- Can also allocate space dynamically at runtime

   use the new operator

```
int *p = new int;
```

```
double *dp = new double(4.5);
```

my\_class \*instance = new my\_class();

- if the operator new cannot allocate memory, it throws the exception std::bad\_alloc (in the <new> header)
  - very uncommon



## **Delete Operator**

- Memory available to a program is limited
  - return dynamically allocated memory to the system if no longer needed
  - use the *delete* operator

int 
$$*p = new int(268);$$

cout << "Integer is: " << \*p; delete p;



## **De-allocating Memory**

- *delete* leaves the variable contents undefined
  - a pointer to a deallocated memory (\*p) cell is possible and *dangerous*
  - deallocated memory can be reassigned after another call to *new*
  - so, indirect reference through 'p' after delete refers to undefined memory
  - called the *dangling pointer* error
  - p = NULL; // safeguard



## Memory Leak

- A memory leak is another common problem when using pointers and dynamic memory
  - happens when allocated memory can no longer be reached
  - so, cannot be de-allocated!
  - wastes memory resources, eventually system will run out of memory
  - int i, \*ip;
  - ip = new int(268);
  - ip = &i; // memory leak!

### **Pointer Examples**





EECS 268 Programming II



### **Best Practices**

- Memory allocated using *new* should be deallocated using *delete*
  - destructor is a good place to deallocate memory
  - implicitly called once object goes out of scope
  - can also be called explicitly when object no longer needed
- Do not call *delete* again to de-allocate same memory

   usually happened unintentionally!
- Do not call delete on a pointer
  - that is not initialized or is NULL,
  - that is pointing to a variable not allocated using *new*



 Use "new" operator to allocate array dynamically *int arraySize = 50;*

double \*anArray = new double[arraySize ];

- delete[] to release array memory delete[] anArray;
- The size of a dynamically allocated array can be increased

double \*oldArray = anArray;

anArray = new double[2\*arraySize];



## **Arrays and Pointers**

- Array name is a pointer to array's first element
- Pointer variable assigned to an array name can be used just like an array

```
int arr[100], *ip;
```

```
ip = arr;
```

```
for(i=0 ; i<100 ; i++)
```

ip[i] = arr[i]+1; // ip and arr are aliased

 ip[i], arr[i], \*(ip+i) all point to the same location.



## Linked List?

- Options for implementing an ADT List
  - Array has a fixed size
    - Data must be shifted during insertions and deletions
  - Linked list is able to grow in size as needed
    - Does not require the shifting of items during insertions and deletions



Figure 4-1 (a) A linked list of integers; (b) insertion; (c) deletion



## **Pointer-Based Linked Lists**

- A node in a linked list is usually a struct struct Node
  - { int item
    - Node \*next;
  - }; // end Node



item next

• The head pointer points to the first node in a



Figure 4-7 A head pointer to a list



## Pointer-Based Linked Lists

- If head is *NULL*, the linked list is empty
- A node is dynamically allocated
   Node \*p; // pointer to node
   p = new Node; // allocate node

isplaying the Contents of a Linked List

- Reference a node member with the -> operator
  - p->item
- Visits each node in the linked list

   pointer variable cur keeps track of current node

### Displaying the Contents of a Linked List



#### Figure 4-9

The effect of the assignment *cur = cur->next* 



• Deleting an interior node

prev->next = cur->next;



Figure 4-10 Deleting a node from a linked list

### Deleting the First Node from a Linked List

• Deleting the first node head = head->next;





Inserting a Node into a Specified Position of a Linked List

• To insert a node between two nodes

newPtr->next = cur;

prev->next = newPtr;



# Serting a Node at the Beginning of a Linked List

To insert a node at the beginning of a linked list

newPtr->next = head; head = newPtr; head 3 6 100 cur Figure 4-13 Inserting at the beginning of a linked list prev newPtr



Inserting a Node into a Specified Position of a Linked List

 Finding the point of insertion or deletion for a sorted linked list of objects

Node \*prev, \*cur;

for (prev = NULL, cur = head;
 (cur != NULL) && (newValue > cur->item);
 prev = cur, cur = cur->next);



A Pointer-Based Implementation of the ADT List

- Public methods
  - isEmpty
  - -getLength
  - -insert
  - remove
  - retrieve
- Private method
  - -find

- Private data members
  - -head
  - size
- Local variables to methods
  - -cur
  - -prev



- **Default constructor initializes** size and head
- A destructor is required for de-allocating dynamically allocated memory
  - else, we will have a memory leak!

```
List::~List()
{
    while (!isEmpty())
        remove(1);
} // end destructor
```

## Constructors and Destructors

- Copy constructor creates a deep copy
  - copies size, head, and the linked list
  - the copy of head points to the copied linked list
- In contrast, a shallow copy
  - copies size and head
  - the copy of head points to the original linked list
- If you omit a copy constructor, the compiler generates one
  - but it is only sufficient for implementations that use statically allocated arrays



*Figure 4-18* Copies of the linked list in Figure 4-17; (a) a shallow copy; (b) a deep copy

Comparing Array-Based and Pointer-Based Implementations

- Size
  - increasing the size of a resizable array can waste storage and time
  - linked list grows and shrinks as necessary
- Storage requirements
  - array-based implementation requires less memory than a pointer-based one for each item in the ADT

Comparing Array-Based and Pointer-Based Implementations

- Retrieval
  - the time to access the ith item
    - Array-based: Constant (independent of i)
    - Pointer-based: Depends on i
- Insertion and deletion
  - Array-based: Requires shifting of data
  - Pointer-based: Requires a traversal



- A method with access to a linked list's head pointer has access to the entire list
- Pass the head pointer to a method as a reference argument
  - Enables method to change value of the head

"Actual argument"



## Processing Linked Lists Recursively

- Recursive strategy to display a list
  - write the first item in the list
  - write the rest of the list (a smaller problem)
- Recursive strategies to display a list backward
  - write the list minus its first item backward
  - write the first item in the list
- Recursive view of a sorted linked list
  - The linked list to which head points is a sorted list if
    - head is NULL or
    - head->next is NULL or
    - head->item < head->next->item, and head->next points to a sorted linked list



## **Objects as Linked List Data**

Data in a node of a linked list can be an instance of a class

typedef ClassName ItemType;
struct Node

- { ItemType item; Node \*next;
- }; //end struct
  Node \*head;



## **Consts and References**

 "Const" keyword is often used in C++ const int val = 100; const int \*ptr = &val;

```
const int * const ptr = &val;
```

void List::method() const;

- Reference variables
  - used for passing arguments to methods by *reference*
  - changes made within the method reflected in caller





- Last node points to the first node
- Every node has a successor
- No node in a circular linked list contains NULL



list

## Variations: Circular Linked Lists

- Access to last node requires a traversal
- Make external pointer point to last instead of first node

- to access both first and last nodes without a traversal



Figure 4-26 A circular linked list with an external pointer to the last node

EECS 268 Programming II

## Variations: Dummy Head Nodes

- Dummy head node
  - always present, even when the linked list is empty
  - insertion and deletion algorithms initialize prev to point to the dummy head node, rather than to NULL
    - eliminates special case for head node



Figure 4-27 A dummy head node



- Each node points to both its predecessor and its successor
- Circular doubly linked list with dummy head node
  - precede pointer of the dummy head node points to the last node
  - next pointer of the last node points to the dummy head node



• To delete the node to which cur points

(cur->precede) ->next = cur->next; (cur->next) ->precede = cur->precede;

• To insert a new node pointed to by newPtr before the node pointed to by cur

newPtr->next = cur; newPtr->precede = cur->precede; cur->precede = newPtr; newPtr->precede->next = newPtr;





(b) listHead



Figure 4-29 (a) A circular doubly linked list with a dummy head node

(b) An empty list with a dummy head node

## he C++ Standard Template Library

- The STL contains class templates for some common ADTs, including the list class
- The STL provides support for predefined ADTs through three basic items
  - Containers
    - Objects that hold other objects
  - Algorithms
    - That act on containers
  - Iterators
    - Provide a way to cycle through the contents of a container



## Summary

- The C++ new and delete operators enable memory to be dynamically allocated and recycled
- Using static 'arrays' Vs dynamic 'lists'
- A class that allocates memory dynamically needs an explicit copy constructor and destructor
  - compiler provides shallow copy constructor by default
- In a doubly linked list, each node points to both its successor and predecessor