CIS 190: C/C++ Programming

Lecture 5
Linked Lists
Outline

• (from last class) Memory and Functions
• Linked Lists & Arrays
• Anatomy of a Linked List
  – Details On Handling headPtr
• Using Linked Lists
  – Creation
  – Traversal
  – Inserting a Node
  – Deleting a Node
• Homework 4B
Memory and Functions

• how do different types of variables get passed to and returned from functions?

• passing by value

• passing by reference
  – implicit: arrays, strings
  – explicit: pointers
Memory and Functions

• some simple examples:

```c
int Add(int x, int y);
    int answer = Add(1, 2);
void PrintMenu(void);
    PrintMenu();
int GetAsciiValue(char c);
    int ascii = GetAsciiValue ('m');
```

• all passed by value
Memory and Functions

• passing arrays to functions
  ```c
  void TimesTwo(int array[], int size);
  
  int arr [ARR_SIZE];
  /* set values of arr */
  
  TimesTwo(arr, ARR_SIZE);
  ```

• arrays of any type are passed by reference
  – changes made in-function persist
Memory and Functions

• passing arrays to functions

```c
void TimesTwo(int array[], int size);
void TimesTwo(int * array, int size);
```

• both of these behave the same way
  – they take a pointer to:
    • the beginning of an array
    • an int – that we (can) treat like an array
Memory and Functions

• passing strings to functions

    void PrintName(char name[]);
    void PrintName(char *name);

    char myName [NAME_SIZE] = "Alice";
    PrintName(myName);

• strings are arrays (of characters)
  – implicitly passed by reference
Memory and Functions

• passing pointers to int to functions

```c
void Square(int *n);
int x = 9;
Square(&x);
```

• pass address of an integer (in this case, x)
Memory and Functions

• passing int pointers to function

```c
void Square(int *n);

int x = 9;
int *xPtr = &x;
Square(??);
```

• pass ???
Memory and Functions

- passing int pointers to function

```c
void Square(int *n);

int x = 9;
int *xPtr = &x;
Square(xPtr);
```

- pass xPtr, which is an address to an integer (x)
Memory and Functions

• returning pointers from functions

    CAR* MakeCar(void) {
        CAR temp;

        return &temp;
    }

• temp is on the stack – so what happens?
Memory and Functions

• returning pointers from functions

```c
CAR* MakeCar(void) {
    CAR temp;

    return &temp;
}
```

• temp is on the stack – so it will be returned to the process when MakeCar() returns!
Memory and Functions

• returning pointers from functions

```c
CAR* MakeCar(void) {
    CAR* temp;
    temp = (CAR*) malloc (sizeof(CAR));
    return temp;  }
```

• temp is on the heap – so what happens?
Memory and Functions

• returning pointers from functions

```c
CAR* MakeCar(void) {
    CAR* temp;
    temp = (CAR*) malloc (sizeof(CAR));
    return temp;
}
```

• temp is on the heap – so it belongs to you and will remain on the heap until you free() it
Outline

• (from last class) Memory and Functions
• Linked Lists & Arrays
• Anatomy of a Linked List
  – Details On Handling headPtr
• Using Linked Lists
  – Creation
  – Traversal
  – Inserting a Node
  – Deleting a Node
• Homework 4B
What is a Linked List?
What is a Linked List?

• data structure
• dynamic
  – allow easy insertion and deletion

• uses nodes that contain
  – data
  – pointer to next node in the list
    • this is called singly linked, and is what we’ll be using
Question

• What are some disadvantages of arrays?

• not dynamic
  – size is fixed once created
  – you can resize it, but you have to do it by hand
  – same for insertion & deletion; it’s possible, but it’s difficult and there’s no built-in function for it

• require contiguous block of memory
Question

• Can we fix these with linked lists? How?

• not dynamic
  – linked lists can change size constantly
  – can add nodes anywhere in a linked lists
  – elements can be removed with no empty spaces

• require contiguous block of memory
  – only one node needs to be held contiguously
Question

• Are there any disadvantages to linked lists?

• harder to search/access than arrayz
• need to manage size/counter for size
• harder to manage memory
  – in-list cycles, segfaults, etc.
• pointer to next node takes up more memory
Outline

• (from last class) Memory and Functions
• Linked Lists & Arrays
• Anatomy of a Linked List
  – Details On Handling headPtr
• Using Linked Lists
  – Creation
  – Traversal
  – Inserting a Node
  – Deleting a Node
• Homework 4B
Linked List Anatomy
Nodes

• a “node” is one element of a linked list

• nodes consist of two parts:

• typically represented as structs
Nodes

- a “node” is one element of a linked list
- nodes consist of two parts:
  - data stored in node
- typically represented as structs
Nodes

• a “node” is one element of a linked list

• nodes consist of two parts:
  – data stored in node
  – pointer to next node in list

• typically represented as structs
Node Definition

• nodes are typically represented as structs

```c
typedef struct node {
    int data;
    NODEPTR next;
} NODE;
```
Node Definition

• nodes are typically represented as structs

```c
typedef struct node * NODEPTR;
typedef struct node {
    int data;
    NODEPTR next;
} NODE;
```

• typedef NODEPTR beforehand so that it can be used in the definition of the NODE structure
Linked List Anatomy

headPtr

data

next

data

next

data

next

data

next

NULL
List Head

• points to the first NODE in the list
  – if there is no list, points to NULL

• headPtr does not contain any data of its own
  – only a pointer to a NODE
Linked Lists

headPtr

NODEPTR

NODE

data

next

NODE

data

next

NODE

data

next

NODE

data

next

NODE

NULL
Linked Lists

```
data
next
NODE @ 0xFFC4

data
next
NODE @ 0xFFC8

data
next
NODE @ 0xFFDC

data
next
NODE @ 0xFFEE

headPtr
NODEPTR @ 0xFFC0

NULL
```
Linked Lists
Linked Lists

NODEPTR @ 0xFFC0

0xFFFFC4

NODE @ 0xFFFFC4

0xFFFFC8

data = 5;

0xFFFFDC

NODE @ 0xFFFFC8

0xFFFFDC

data = 3;

0xFFFFEE

NODE @ 0xFFFFDC

0xFFFFEE

data = 8;

NULL

NODE @ 0xFFFFEE

NULL
Outline

• (from last class) Memory and Functions
• Linked Lists & Arrays
• Anatomy of a Linked List
  – Details On Handling headPtr
• Using Linked Lists
  – Creation
  – Traversal
  – Inserting a Node
  – Deleting a Node
• Homework 4B
More About headPtr

- headPtr is a pointer to a NODE
  - it has a place where it’s stored in memory
  - and it has where it points to in memory
More About headPtr

• headPtr is a pointer to a NODE
  – it has a place where it’s stored in memory
  – and it has where it points to in memory

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC4</td>
<td>0xDC44</td>
</tr>
</tbody>
</table>

address: where it’s stored in memory
value: where it points to in memory
Passing Pointers

• when we pass a pointer by value, we pass where it points to in memory

• so we can change the value(s) stored in the memory location to which it points
  – but we can’t alter the pointer itself
void SquareNum (int *intPtr) {
    (*intPtr) = (*intPtr) * (*intPtr);
}

Passing Pointers by Value Example
Passing Pointers by Value Example

```c
void SquareNum (int *intPtr) {
    (*intPtr) = (*intPtr) * (*intPtr);
}

int x = 4;
int *xPtr = &x;
```
void SquareNum (int *intPtr) {
    (*intPtr) = (*intPtr) * (*intPtr);
}

int x = 4;
int *xPtr = &x;
SquareNum (xPtr);
/* value of x is now 16 */
Passing Pointers

• when we pass a pointer by reference, we are passing where it is stored in memory

• so we can change both
  – where it points to in memory
  
  and

  – the values that are stored there
void Reassign (int **ptr,
               int *newAddress) {
    *ptr = newAddress;
}
void Reassign (int **ptr,
             int *newAddress) {
    *ptr = newAddress;
}

int x = 3, y = 5;
int *intPtr = &x;
void Reassign (int **ptr, int *newAddress) {
    *ptr = newAddress;
}

int x = 3, y = 5;
int *intPtr = &x;
ReassignPointer (&intPtr, &y);
/* intPtr now points to y */
headPtr Naming Conventions

• two variable names for headPtr inside functions

• when we pass headPtr by value
  – we pass where it points to in memory
  – NODEPTR head = address of first node

• when we pass headPtr by reference
  – we pass where it’s stored in memory
  – NODEPTR *headPtr = where headPtr is stored
Outline

• (from last class) Memory and Functions
• Linked Lists & Arrays
• Anatomy of a Linked List
  – Details On Handling headPtr
• Using Linked Lists
  – Creation
  – Traversal
  – Inserting a Node
  – Deleting a Node
• Homework 4B
Building a Linked List from Scratch
Building a Linked List from Scratch

1. declare a headPtr, and set equal to NULL
Building a Linked List from Scratch

1. declare a headPtr, and set equal to NULL
2. allocate space for a node and set to a temporary variable
Building a Linked List from Scratch

1. declare a headPtr, and set equal to NULL
2. allocate space for a node and set to a temporary variable
3. initialize node’s data
Building a Linked List from Scratch

1. declare a headPtr, and set equal to NULL
2. allocate space for a node and set to a temporary variable
3. initialize node’s data
4. insert node in list
Building a Linked List from Scratch

• insert another node
Building a Linked List from Scratch

- insert another node
- and another, etc.
Outline

• (from last class) Memory and Functions
• Linked Lists & Arrays
• Anatomy of a Linked List
  – Details On Handling headPtr
• Using Linked Lists
  – Creation
  – Traversal
  – Inserting a Node
  – Deleting a Node
• Homework 4B
Creating a Node

 NODEPTR CreateNode (void)

1. create and allocate memory for a node

   newNode = (NODEPTR) malloc (sizeof(NODE));
Creating a Node

**NODEPTR CreateNode (void)**

1. create and allocate memory for a node
   ```c
   newNode = (NODEPTR) malloc (sizeof(NODE));
   ```

   – cast as NODEPTR, but space for NODE – why?
Creating a Node

NODEPTR CreateNode (void)

1. create and allocate memory for a node
   
   newNode = (NODEPTR) malloc (sizeof(NODE));

   – cast as NODEPTR, but space for NODE – why?

2. ensure that memory was allocated

3. initialize data
Setting a Node’s Data

```c
void SetData (NODEPTR temp, int data)
```

• temp is a pointer, but it points to a struct
  – use arrow notation to access elements
    • or dot star notation

```c
temp->data = data;
(*temp).data = data;
```
When “data” Itself is a Struct

typedef struct node {
    CIS_CLASS class;
    NODEPTR next;
} NODE;

CIS_CLASS STRUCT ("data")

NODE STRUCT

{ int classNum; char room[ROOM_STR]; char title [TITLE_STR];
  next
}
Setting Data when “data” is a Struct

```c
void SetData (NODEPTR temp, int classNum, 
             char room  [ROOM_STR], 
             char title [TITLE_STR])

    temp->class.classNum = classNum;
    strcpy(temp->class.room,  room);
    strcpy(temp->class.title, title);
```

- the **class** struct is not a pointer, so we can use just dot notation
Outline

• (from last class) Memory and Functions
• Linked Lists & Arrays
• Anatomy of a Linked List
  – Details On Handling headPtr
• Using Linked Lists
  – Creation
  – Traversal
  – Inserting a Node
  – Deleting a Node
• Homework 4B
Traversing a Linked List

• used for many linked list operations

• check to see if list is empty

• use two temporary pointers to keep track of current and previous nodes (prev and curr)
  • move through list, setting prev to curr and curr to the next element of the list
    – continue until you hit the end (or conditions met)
Special Cases with Linked Lists

• always a separate rule when dealing with the first element in the list (where headPtr points)
  – and a separate rule for when the list is empty

• laid out in the code available online, but keep it in mind whenever working with linked lists
  – make sure you understand the code before you start using it in your programs
Traversing a Linked List – Step 1

- set `curr` to `*headPtr`, the first NODE in the list
Traversing a Linked List – Step 2

- check if `curr == NULL` (end of list)
  - if it doesn’t, continue
Traversing a Linked List – Step 2

- check if `curr == NULL` (end of list)
  - if it doesn’t, continue

or if your conditions have been met! but you
must always check that `curr != NULL` first
Traversing a Linked List – Step 3

• set $prev = curr$
Traversing a Linked List – Step 4

- \texttt{set curr = curr->next}
Traversing a Linked List – Step 4

- set curr = curr->next
Traversing a Linked List – Step 5

- continue to repeat steps 2-4 until you reach the end

headPtr
NODEPTR

curr
prev

NODE

data

next

NODE

data

next

NODE

data

next

NULL
Traversing a Linked List – Step 5...

- check if `curr == NULL` (end of list)
  - if it doesn’t, continue
Traversing a Linked List – Step 5...

* set $\text{prev} = \text{curr}$
Traversing a Linked List – Step 5...

- set curr = curr->next
Traversing a Linked List – Step 5...

- set curr = curr->next
Traversing a Linked List – Step 5...

- check if \texttt{curr == NULL} (end of list)
  - if it doesn’t, continue
Traversing a Linked List – Step 5...

- set `$prev = curr$`
Traversing a Linked List – Step 5...

- set curr = curr->next
Traversing a Linked List – Step 5...

- set \( \text{curr} = \text{curr} \rightarrow \text{next} \)
Traversing a Linked List – Step 5...

- check if `curr == NULL` (end of list)
Traversing a Linked List – Step 5...

- check if `curr == NULL`
  (end of list)

- it does!
- we’ve reached the end of the list
Printing the Entire Linked List

```c
void PrintList (NODEPTR head)
```

- check to see if list is empty
  - if so, print out a message
- if not, traverse the linked list
  - print out the data of each node
Outline

• (from last class) Memory and Functions
• Linked Lists & Arrays
• Anatomy of a Linked List
  – Details On Handling headPtr
• Using Linked Lists
  – Creation
  – Traversal
  – Inserting a Node
  – Deleting a Node
• Homework 4B
Inserting a Node

```c
void Insert (NODEPTR *headPtr,
            NODEPTR temp)
```

- check if list is empty
  - if so, temp becomes the first node
- if list is not empty
  - traverse the list and insert temp at the end
Inserting a Node

- check if `curr == NULL` (end of list)
Inserting a Node

• check if \texttt{curr == NULL} (end of list)

• insert the new node by changing where \texttt{prev->next} points to
Inserting a Node

- check if `curr == NULL` (end of list)
- insert the new node by changing where `prev->next` points to
  - address of new node
Inserting a Node

- check if `curr == NULL` (end of list)
- insert the new node by changing where `prev->next` points to – address of new node
- new node is successfully inserted at end of the list!
Inserting a Node in the Middle

int Insert (NODEPTR *headPtr, NODEPTR temp, int where)

• traverse list until you come to place to insert
  – CAUTION: don’t go past the end of the list!
• insert temp at appropriate spot
  – CAUTION: don’t “lose” any pointers!
• return an integer to convey success/failure
Inserting a Node – Step 1

- traverse the list until you find where you want to insert temp
Inserting a Node – Step 2

- first have `temp->next` point to what will be the node following it in the list (`curr`)

\[ temp->next = curr; \]
Inserting a Node – Step 3

- then you can have `prev->next` point to `temp` as the new next node in the list

```c
prev->next = temp;
```
Inserting a Node – Done

- `temp` is now stored in the list between `prev` and `curr`
Inserting a Node – Done

- \textit{temp} is now stored in the list between \texttt{prev} and \texttt{curr}
- return a successful code (insert worked)
Outline

• (from last class) Memory and Functions
• Linked Lists & Arrays
• Anatomy of a Linked List
  – Details On Handling headPtr
• Using Linked Lists
  – Creation
  – Traversal
  – Inserting a Node
  – Deleting a Node
• Homework 4B
Deleting a Node

```c
int Delete (NODEPTR *headPtr,
           int target)
```

• code is similar to insert
• pass in a way to find the node you want to delete
  – traverse list until you find the correct node:
    ```c
    curr->data == target
    ```
• return an integer to convey success/failure
Deleting a Node – Step 1

• traverse the list, searching until `curr->data` matches `target`
Deleting a Node – Step 1

- traverse the list, searching until `curr->data` matches target

but don’t forget, you **must** always check that `curr != NULL` first
Deleting a Node – Step 2

- "remove" `curr` from the list by setting `prev->next` to `curr->next`
Deleting a Node – Step 3

- free the memory used by curr and set pointers to NULL
Deleting a Node – Step 3

- free the memory used by `curr` and set pointers to NULL

```c
curr->next = NULL;
```

100
Deleting a Node – Step 3

- free the memory used by `curr` and set pointers to NULL

```
curr->next = NULL;
free(curr);
```
Deleting a Node – Step 3

- free the memory used by \texttt{curr} and set pointers to NULL

\texttt{curr->next = NULL; free(curr);}
Deleting a Node – Step 3

- free the memory used by `curr` and set pointers to NULL

```c
curr->next = NULL;
free(curr);
curr = NULL;
```
Deleting a Node – Step 3

- free the memory used by `curr` and set pointers to NULL

```c
curr->next = NULL;
free(curr);
curr = NULL;
```
Outline

• (from last class) Memory and Functions
• Linked Lists & Arrays
• Anatomy of a Linked List
  – Details On Handling headPtr
• Using Linked Lists
  – Creation
  – Traversal
  – Inserting a Node
  – Deleting a Node
• Homework 4B
Homework 4B

• Karaoke

• heavy on pointers and memory management
• think before you attack

• start early
• test often (don’t forget edge cases)
• use a debugger when needed
Linked List Code for HW4B

• code for all of these functions is available on the Lectures page

• comments explain each step

• you can use this code in your Homework 4B, or as the basis for similar functions