CMSC 216
Introduction to Computer Systems
Lecture 10
Dynamic Memory Allocation & Linked Lists

Larry Herman & Alan Sussman

Notes

- Project 2 late deadline tomorrow
  - questions?
- Project 3 posted today
  - submit server open and public tests posted soon
  - we'll talk about it on Thursday
- Exam #1 next Thursday, March 8
  - practice exam posted by end of week, with answers later
- Read Reek, Chapter 12: Linked Lists

The two new string functions

- These functions are often used with pointer arithmetic
- What does this do?

```
void mystery(char *t, const char *s, char ch) {
  char *p;
  s = strchr(s, ch);
  if (s != NULL) {
    p = strchr(s + 1, ch);
    if (p != NULL) {
      strncpy(t, s + 1, p - s - 1);
      *(t + (p - s - 1))= '\0';
    }
  }
}
```

Chapter 11, Reek

DYNAMIC MEMORY ALLOCATION

Don't dereference freed pointers

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int x;
  int y;
} Point;

int main() {
  Point p = {3, 4}, *p1, *p2;
  p1 = malloc(sizeof(Point));  /* no NULL check!  Baaad... */
  p1->x = 5;
  p1->y = 9;
  printf("%d %d\n", p1->x, p1->y);
  free(p1);  /* now p1 points to invalid memory */
  p2 = malloc(sizeof(Point));
  *p2 = p;
  printf("%d %d\n", p1->x, p1->y);  /* Printing out p1, not p2... */
  return 0;
}
```


Pointer aliases

- We can have two pointers pointing to the same address
- But what happens when we free one of them?

```
int *p, *q;
p = malloc(sizeof(int));
q = p;
free(p);
p = NULL;
*q = 42; /* what is the value of q? */
```

- `q` is called a dangling pointer

Common errors

- Forgetting to check the return value from `malloc()` for `NULL`
- Not initializing the memory `malloc()` returns
- Not allocating enough space:

```
char string[] = "Inconceivable!";
char *p = malloc(strlen(string));
strcpy(p, string);  /* Oops... */
```


Common errors, cont.

- Attempting to free non-heap memory:

```
int i, *p;
p = &i;
free(p);  /* ??? Undefined operation */
```

- Freeing something that's not the **beginning** of a dynamically allocated block:

```
int *p = calloc(10, sizeof(int));
free(p + 3);
```

- Freeing the same block more than once:

```
int *p = malloc(sizeof(int)), *q;
q = p;
free(p);
free(q);
```

Common errors, cont.

- Dereferencing a garbage pointer

```
int *p;
*p = 42;
```

- Dereferencing a **NULL** pointer

```
int *p;
p = NULL;
*p = 23;
```

- Dereferencing a dangling pointer, again

```
int *p = malloc(...);
free(p);
...
*p = 27;
```


Common errors, cont.

- Referring to data past the end of allocated space
  - Remember the string example?
- Using freed memory – dangling pointer again

```
List_node *ptr =
  malloc(sizeof(List_node));
...
free(ptr);
...
ptr = ptr->next;  /* Uh oh... */
```


Common errors, cont.

- What happens to the first **Stack** object in Java?

```
Stack s = new Stack();
...
s = new Stack();
```

- Losing a pointer to dynamically allocated memory causes memory leaks
  - not an immediately fatal error
  - can just be a waste of resources, but left to run for a while, it can keep your program from being able to do anything

Reallocation

- If you need more space than you originally allocated, you could handle this yourself:
  - allocate a larger array
  - copy everything over
  - free original array
- But, there's a way to do this without nearly as much work on your part

Reallocation, cont.

```
void *realloc(void *p, size_t new_size);
```

- checks current size of the block **p** points to
  - if >= **new_size**, can perform reallocation in place, and returns **p**
  - if < **new_size**, new block of size **new_size** is allocated
    - if allocation fails, returns **NULL**
    - otherwise, copies items from **p** over, free **p**, and returns pointer to new block

Use of `realloc()`

- Often, you'll see this:

```
ptr = realloc(ptr, size * 2);
```

- But what happens if the allocation fails?
- You can only do this if you know for certain you're going to exit the program on an allocation failure

Chapter 12

USING STRUCTURES AND POINTERS

Linked lists

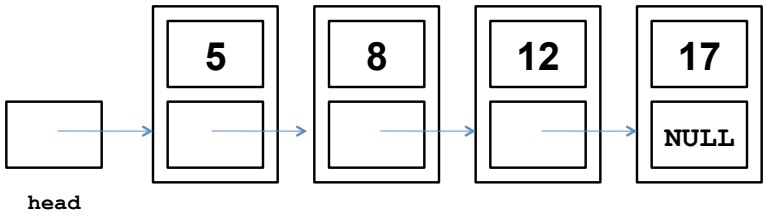
- Much like Java's objects with references to objects of that class, C structures can have pointers to structures of the same type

```
typedef struct node {
  int val;
  struct node *next;
} Node;

Node *head;
```

- Both **next** and **head** are of the same type

Linked list in memory

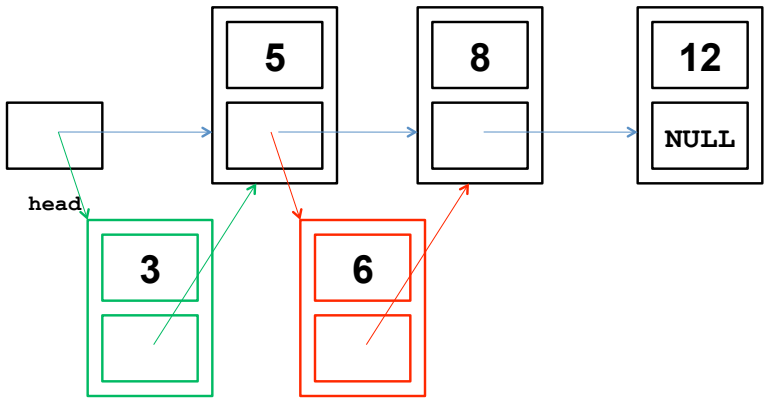


Searching through a linked list

```
Node *find(Node *start, int target) {
  while (start != NULL) {
    if (start->val == target)
      return start;
    start = start->next;
  }
  return NULL;
}

int find(Node *start, int target) {
  while (start != NULL && start->val != target)
    start = start->next;
  return start != NULL;
}
```


Inserting into an ordered linked list



Tracing insertion

```
int insert(Node *start, int new_value) {
  Node *current = start, *prev = NULL, *new_item;

  while (current != NULL && current->val < new_value) {
    prev = current;
    current = current->next;
  }
  new_item = malloc(sizeof(*new_item));
  if (new_item == NULL)
    return 0;

  else {
    new_item->val = new_value;
    new_item->next = current;
    if (prev == NULL)
      start = new_item;
    else
      prev->next = new_item;
    return 1;
  }
}
```

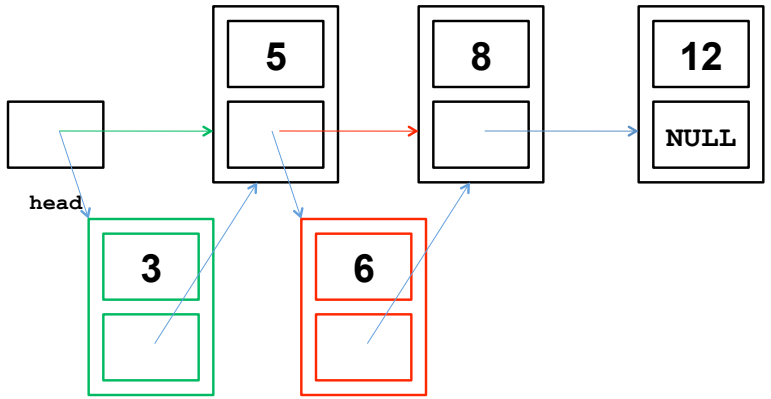
Tracing insertion - fixed

```
int insert(Node **start, int new_value) {
  Node *current, *prev = NULL, *new_item;

  if (start == NULL) return 0;

  current = *start;
  while (current != NULL && current->val < new_value) {
    prev = current;
    current = current->next;
  }
  new_item = malloc(sizeof(*new_item));
  if (new_item == NULL)
    return 0;
  new_item->val = new_value;
  new_item->next = current;
  if (prev == NULL)
    *start = new_item;
  else
    prev->next = new_item;
  return 1;
}
```


Deleting from a linked list



Tracing deletion

```
int delete(Node **start, int value) {
  Node *prev = NULL, *current = *start;

  if (start == NULL)
    return 0;

  current = *start;
  while (current != NULL && current->val!= value) {
    prev = current;
    current = current->next;
  }
  if (current == NULL)
    return 0;  /* not found */

  if (prev != NULL)
    prev->next = current->next;
  else
    *start = current->next;  /* deleted first item */
  free(current);
  return 1;
}
```