CMSC 313
Introduction to Computer Systems
Lecture 11
Linked Lists &
the C Preprocessor

Alan Sussman
als@cs.umd.edu

Administrivia

- Project 3 spec and public tests posted, due Oct. 20
  - questions?
- Exam #1 returned Wednesday
  - questions answered then, or in my office hours

Chapter 11, Reek

DYNAMIC MEMORY ALLOCATION

Operating on blocks of memory

- Much as we have functions like **`strcpy()`** that operate on strings of characters, we can copy whole blocks of memory (or strings of bytes, if you like)

```
void *memcpy(void *dst, void *src, size_t n);
void *memmove(void *dst, void *src, size_t n);
```

- Both copy **`n`** bytes from the memory starting at **`src`** to the memory starting at **`dst`**, and return the **`dst`** pointer
- **`memcpy()`** cannot be used on overlapping arrays, though; but it's faster than **`memmove()`**
- Both prototypes contained in **`string.h`**

A possible **memcpy()** implementation

```
#include <stdio.h>
/* not #including <string.h> because of conflict */

void *memcpy(void *dst, void *src, size_t n) {
  char *dp = dst;
  char *sp = src;
  while (dp - (char *) dst < n)
    *dp++ = *sp++;
  return dst;
}

int main() {
  int i, j = 5;
  memcpy(&i, &j, sizeof(int));
  printf("%d\n", i);
  return 0;
}
```


Chapter 12

USING STRUCTURES AND POINTERS

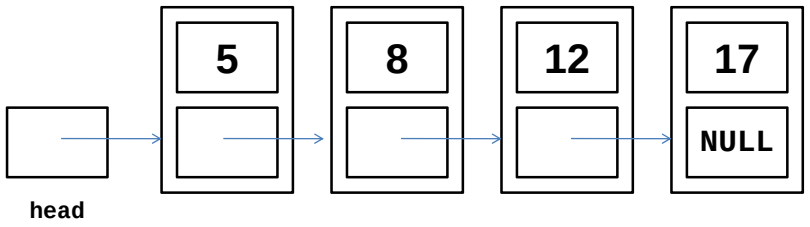
Linked lists

- 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

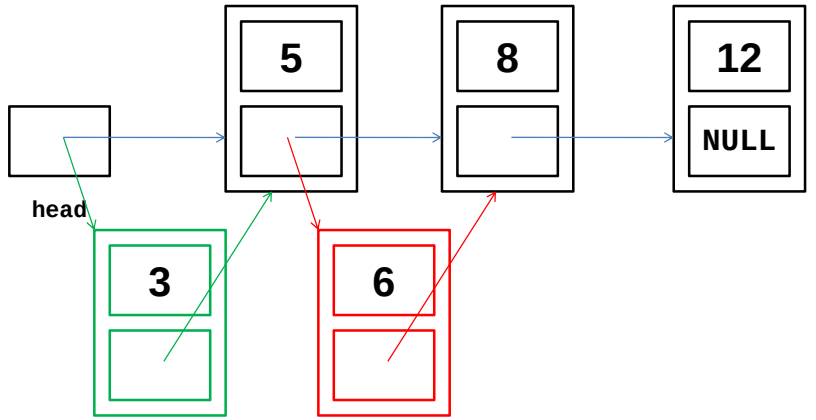
Linked list in memory



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



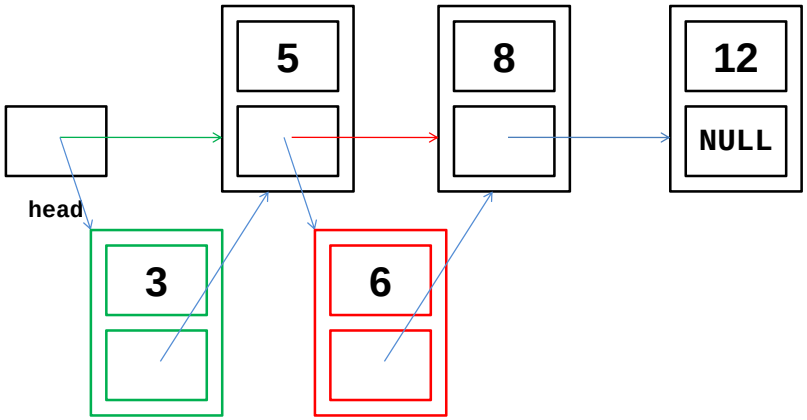
Tracing insertion

```
int insert(Node *head, int new_value) {
  Node *current = head, *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;
  new_item->val = new_value;
  new_item->next = current;
  if (prev == NULL)
    head = new_item;
  else
    prev->next = new_item;
  return 1;
}
```


Tracing insertion - fixed

```
int insert(Node **head, int new_value) {
  Node *current = *head, *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;
  new_item->val = new_value;
  new_item->next = current;
  if (prev == NULL)
    *head = new_item;
  else
    prev->next = new_item;
  return 1;
}
```

Deleting from a linked list



Tracing deletion

```
int delete(Node **head, int value) {
  Node *prev = NULL, *current = *head;
  while (current != NULL && current->data != value) {
    prev = current;
    current = current->next;
  }
  if (current == NULL)
    return 0;  /* not found */
  if (prev != NULL)
    prev->next = current->next;
  else
    *head = current->next;  /* deleted first item */
  free(current);
  return 1;
}
```


Doubly linked lists

- We can have references in both directions
- Allows use of only one pointer when performing some operations

```
typedef struct dl_node {
  struct dl_node *prev;
  struct dl_node *next;
  int val;
} DL_node;
```


Deletion from a doubly linked list

```
int delete(DL_node **root, int target) {
  DL_node *current;
  for (current = *root; current != NULL; current = current->next)
    if (current->val == target)
      break;
  if (current == NULL)
    return 0;
  if (current == *root)
    *root = current->next;
  else
    current->prev->next = current->next;
  if (current->next != NULL)
    current->next->prev = current->prev;
  free(current);
  return 1;
}
```

Chapter 14, Reek

THE PREPROCESSOR

Compiling a program

- Source files (**`*.c`**) compiled into object files (**`*.o`**)
  1. Source file is first preprocessed
  2. Preprocessed file is then compiled to assembly
     1. scanner/parser
     2. type checker
     3. code generator
     4. optimizer
  3. Assembler converts assembly code to object code
- Object files are converted into an executable
  - Done by the linker, which resolves symbols
    - match function use to its definition
    - global variable declaration and external use

Preprocessor predefined symbols

- **`__FILE__`** : a string, the filename in which the symbol is found
- **`__LINE__`** : an integer, the line on which the symbol is found
- **`__DATE__`** : a string, date of compilation
- **`__TIME__`** : a string, time of compilation
- **`__STDC__`** : 1 or undefined, whether or not compiler is ANSI-compliant
- Note that each is preceded and followed by two underscores
- Examples of their values are available in table 14.1 (pg. 383) of Reek

The **`#define`** directive

- Syntax: **`#define`** *name lots of text*
- Substitutes lots of text for name throughout the source file
- Most commonly used for constants
- By convention, we use all capital letters for our constants - makes it easier for humans to recognize them

Macros

- **`#define`** can also be used to define macros, where parameters are substituted as well as straight text substitution
- Syntax: **`#define`** *name(parameter-list) text*
- Example:

```
#define SUM(a, b) a + b
...
x = SUM(2, 6);
/* becomes "x = 2 + 6;" */
```


Macros, cont.

- We can also use a backslash at the end of a line to extend a macro onto a second line:

```
#define MIN(a, b) a < b \
  ? a : b
```


Typical uses for macros

- **`#define SQUARE(x)  ((x) * (x))`**

```
s = SQUARE(5);
```

becomes

```
s = ((5) * (5));

s = SQUARE(a + 1);
```

becomes

```
s = ((a + 1) * (a + 1));
```