Week 9 - Wednesday

CS222

Last time

- What did we talk about last time?
- structs

Questions?

Project 4

Quotes

C combines the power and performance of assembly language with the flexibility and ease-of-use of assembly language.

Anonymous

typedef

Naming types

- You might have noticed that there are all these odd types floating around
  - `time_t`
  - `size_t`
- On some systems, you will even see aliases for your basic types
  - `FLOAT`
  - `INT32`
- How do people create new names for existing types?

typedef

- The **typedef** command allows you to make an alias for an existing type
- You type **typedef**, the type you want to alias, and then the new name

```
typedef int SUPER_INT;

SUPER_INT value = 3; //has type int
```

- Don't overuse **typedef**
- It is useful for types like **time_t** which can have different meanings in different systems

typedef with structs

- The **typedef** command is commonly used with structs
  - Often it is built into the struct declaration process
- It allows the programmer to leave off the stupid **struct** keyword when declaring variables

```
typedef struct _wombat
{
        char name[100];
        double weight;
} wombat;
```

- The type defined is actually **struct _wombat**
- We can refer to that type as **wombat**

```
wombat martin;
```

Even more confusing!

- You can actually **typedef** the name of the struct to be the same without the struct part

```
typedef struct wombat
{
      char name[100];
      double weight;
} wombat;
```

- Or, if you don't need the name of the struct inside itself, you can **typedef** an anonymous struct

```
typedef struct
{
      char name[100];
      double weight;
} wombat;
```

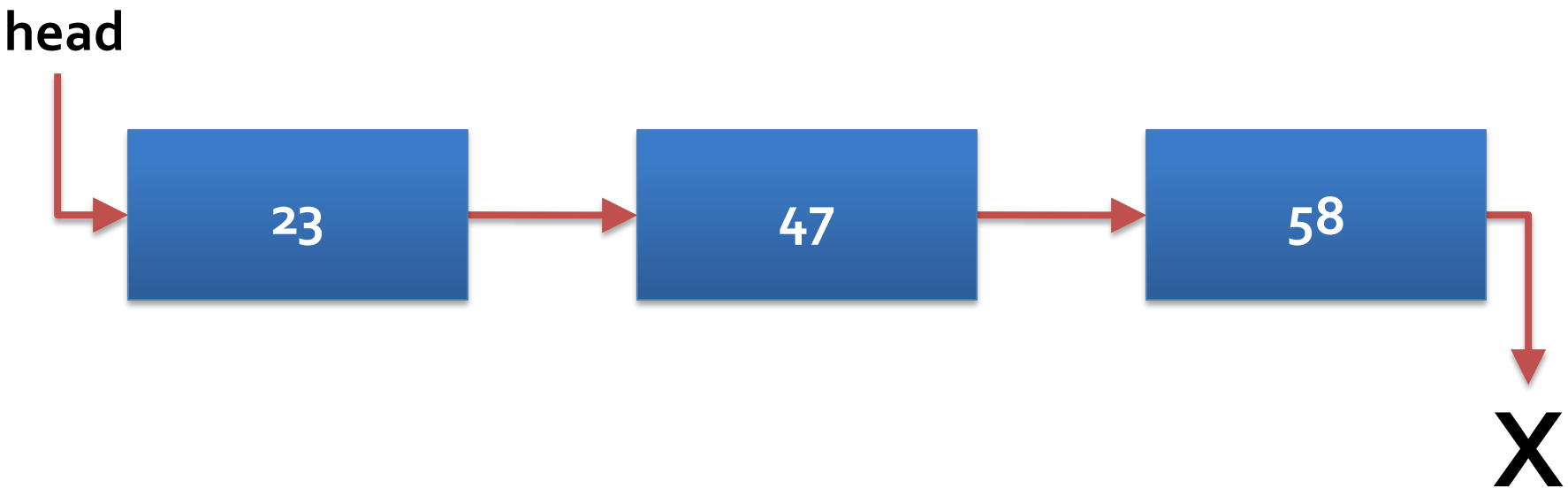
Linked lists

Linked lists

- Since you have all taken CS122 (and many have taken CS221), you all know the power of the linked list
- A linked list is a dynamic data structure with the following features:
  - Insertion, add, and delete can be O(1) time
  - Search is O(***n***) time
  - They are ideally suited for a merge sort
  - They are a pain to program

Singly linked list

- Node consists of data and a single next pointer
- Advantages: fast and easy to implement
- Disadvantages: forward movement only



Linked lists in C

- Since C doesn't have classes, we can't make a self-contained linked list
- But we can create nodes and a set of operations to use on them
- Clearly, we will need a struct to make the node
  - It will contain data
  - It will contain a pointer to the next node in the list
- Doubly-linked lists are possible too

An example node struct

- We'll use this definition for our node for singly linked lists

```
typedef struct _node
{
      int data;
      struct _node* next;
} node;
```

- Somewhere, we will have the following variable to hold the beginning of the list

```
node* head = NULL;
```

Add to front

- Let's define a function that takes a pointer to a (possibly empty) linked list and adds a value to the front
- There are two possible ways to do it
  - Return the new head of the list

```
node* add(node* head, int value);
```

  - Take a pointer to a pointer and change it directly

```
void add(node** headPointer, int value);
```

Find

- Let's define a function that takes a pointer to a (possibly empty) linked list and a value and returns the **node** containing the value
  - Or **NULL** if there is no such **node**

```
node* find(node* head, int value);
```

Sum values

- Let's define a function that takes a pointer to a (possibly empty) linked list and returns the sum of the values inside
  - An empty list has a sum of **0**

```
int sum(node* head);
```

Remove

- Let's define a function that takes a pointer to a (possibly empty) linked list and deletes the first occurrence of a given value
  - List is unchanged if the value isn't found
- There are two possible ways to do it
  - Return the new head of the list

```
node* remove(node* head, int value);
```

  - Take a pointer to a pointer and change it directly

```
void remove(node** headPointer, int value);
```

Insert in sorted order

- Let's define a function that takes a pointer to a (possibly empty) linked list and adds a value in sorted order (assuming that the list is already sorted)
- There are two possible ways to do it
  - Return the new head of the list

```
node* add(node* head, int value);
```

  - Take a pointer to a pointer and change it directly

```
void add(node** headPointer, int value);
```

Upcoming

Next time…

- Deeper coverage of linked lists
- Lab 9

Reminders

- Finish Project 4
  - **Due Friday by midnight!**
- Keep reading K&R chapter 6