Week 9 - Monday

COMP 2400

Last time

- What did we talk about last time?
- Linked lists

Questions?

Project 4

Quotes

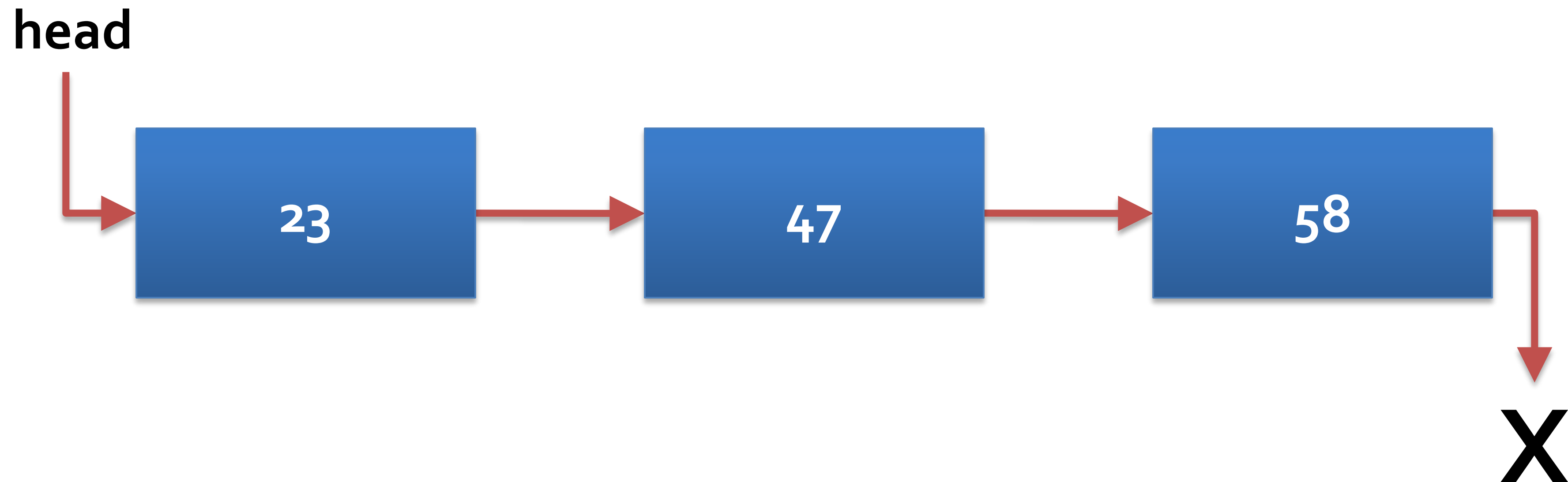
C combines the power and performance of assembly language with the flexibility and ease-of-use of assembly language.

Anonymous

Linked lists

Singly linked list

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



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;
```

Empty

- Let's write a method that will remove all the nodes from a singly linked list
  - Don't forget to free all the nodes!

```
void empty(node* head);
```

- With this implementation, the user will have to set **head** to **NULL** manually

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);
```

Enums

`enum`

- There are situations where you'd like to have a set of named constants
- In many cases, you'd like those constants to be different from each other
- What if there were a way to create such a list of constants easily?
- Enter **`enum`**!

Using enum

- To create these constants, type enum and then the names of your constants in braces

```
enum { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY };
```

- Then in your code, you can use these values (which are stored as integers)

```
int day = FRIDAY;
if (day == SUNDAY)
    printf ("My 'I don't have to run' day\n");
```

Creating enum types

- You can also create named enum types

```
enum Color { BLACK, BLUE, GREEN, ORANGE, PURPLE, RED, WHITE,
YELLOW };
```

- Then you can declare variables of these types

```
enum Color color;
color = YELLOW;
```

- Naturally, because they are constants, it is traditional to name enum values in ALL CAPS

typedef + enum

- If you want to declare **enum** types (and there isn't much reason to, since C treats them exactly like **int** values), you can use **typedef** to avoid typing **enum** all the time

```
typedef enum { C, C_PLUS_PLUS, C_SHARP, JAVA, JAVASCSRIPT,
LISP, ML, OBJECTIVE_C, PERL, PHP, PYTHON, RUBY, VISUAL_BASIC }
Language;

Language language1 = C;
Language language2 = JAVA;
```

enum values

- **enum** values by default start at 0 and increase by one with each new constant

```
enum { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY };
```

- In this case, the constants have the following numbering
  - **SUNDAY**: 0
  - **MONDAY** : 1
  - **TUESDAY** : 2
  - **WEDNESDAY** : 3
  - **THURSDAY** : 4
  - **FRIDAY** : 5
  - **SATURDAY** : 6

Specifying values

- You can even specify the values in the **`enum`**

```
enum { ANIMAL = 7, MINERAL = 9, VEGETABLE = 11 };
```

- If you assign values, it is possible to make two or more of the constants have the same value (usually bad)
- A common reason that values are assigned is so that you can do bitwise combinations of values

```
enum { PEPPERONI = 1, SAUSAGE = 2, BACON = 4, MUSHROOMS = 8,
PEPPER = 16, ONIONS = 32, OLIVES = 64, EXTRA_CHEESE = 128 };

int toppings = PEPPERONI | ONIONS | MUSHROOMS;
```

A classic enum

- Before C90, there was no **`bool`** type
- Then, a common uses of **`enum`** was to specify a Boolean type

```
typedef enum { FALSE, TRUE } BOOLEAN;

BOOLEAN value = TRUE;
BOOLEAN flag = FALSE;
```

- It's not a perfect system, since you can assign values other than **`0`** and **`1`** to a **`BOOLEAN`**
- Likewise, other values are also true in C

Bit Fields

Saving space

- The next topics we'll discuss today are primarily about saving space
- They don't make code safer, easier to read, or more time efficient
- At C's inception, memory was scarce and expensive
- These days, memory is plentiful and cheap

What if you wanted to record bits?

- The smallest addressable chunk of memory in C is a byte
  - Stored in a **`char`**
- If you want to record several individual bit values, what do you do?
- You can use bitwise operations (**`&`**, **`|`**, **`<<`**, **`>>`**, **`~`**) to manipulate bits
  - But it's tedious!

Bit fields in a struct

- You can define a struct and define how many bits wide each element is
  - It only works for integral types, and it makes the most sense for **`unsigned int`**
  - Give the number of bits it uses after a colon
  - The bits can't be larger than the size the type would normally have
  - You can have unnamed fields for padding purposes

```
typedef struct _toppings
{
        unsigned pepperoni : 1;
        unsigned sausage   : 1;
        unsigned onions    : 1;
        unsigned peppers   : 1;
        unsigned mushrooms : 1;
        unsigned sauce     : 1;
        unsigned cheese    : 2; //goes from no cheese to triple cheese
} toppings;
```

Code example

- You could specify a pizza this way

```
toppings choices;
memset(&choices, 0, sizeof(toppings));
//sets the garbage to all zeroes
choices.pepperoni = 1;
choices.onions = 1;
choices.sauce = 1;
choices.cheese = 2; //double cheese
order(&choices);
```

Struct size and padding

- Structs are always padded out to multiples of 4 or even 8 bytes, depending on architecture
  - Unless you use compiler specific statements to change byte packing
- After the last bit field, there will be empty space up to the nearest 4 byte boundary
- You can mix bit field members and non-bit field members in a struct
  - Whenever you switch, it will pad out to 4 bytes
  - You can also have **0** bit fields which also pad out to 4 bytes

Padding example

```
struct kitchen
{
   unsigned light     : 1;
   unsigned toaster   : 1;
   int count;   // 4 bytes
   unsigned outlets   : 4;
   unsigned           : 4;
   unsigned clock     : 1;
   unsigned           : 0;
   unsigned flag      : 1;
};
```

16 bytes

Data	Bits
light	1
toaster	1
padding	30
count	32
outlets	4
unnamed	4
clock	1
unnamed	0
padding	23
flag	1
padding	31

Upcoming

Next time…

- Finish bit fields
- Unions
- Trees
- Users and time

Reminders

- Finish Project 4
  - **Due Friday by midnight!**
- Keep reading K&R chapter 6
- Read LPI chapters 8 and 10
- **Exam 2 is next Monday**