Week 12 - Monday

CS222

Last time

- What did we talk about last time?
- Binary file I/O
- Lab 11

Questions?

Project 5

Quotes

Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?" Improve the code and then document it to make it even clearer.

Steve McConnell
Author of *Code Complete*

Saving space

- The topics we will 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

Bit Fields

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 toaster *padding*	1 1 30
count	32
outlets *unnamed* clock *unnamed* *padding*	4 4 1 0 23
flag *padding*	1 31

An alternative to bitwise operations

- You can also use a pointer to a struct with bit fields to read bit values out of other types

```
typedef struct
{
     unsigned LSB     : 1;
     unsigned         : 30;
     unsigned MSB     : 1;
} bits;
```

- Which bit is which is dependent on endianness

```
bits* bitsPointer;
int number = 1;
float value = 3.7;
bitsPointer = (bits*)&number;
printf("LSB: %d\nMSB: %d\n", bitsPointer->LSB,
bitsPointer->MSB);
```

Unfortunately…

- Bit fields are compiler and machine dependent
- How those bits are ordered and packed is not specified by the C standard
- In practice, they usually work
  - Most machines are little endian these days
- In theory, endianness and packing problems can interfere

Unions

Unions

- What if you wanted a data type that could hold any of three different things
  - But it would only hold one at a time…
- Yeah, you probably wouldn't want that
- But, back in the day when space was important, maybe you would have
- This is exactly the problem that unions were designed to solve

Declaring unions

- Unions look like structs
  - Put the keyword **`union`** in place of **`struct`**

```
union Congressperson
{
      int district;    //representatives
      char state[15]; //senators
};
```

- There isn't a separate district and a state
  - There's only space for the larger one
  - In this case, 15 bytes (rounded up to 16) is the larger one

Example use

- We can store into either one

```
union Congressperson representative;
union Congressperson senator;
representative.district = 1;
strcpy(senator.state, "Wisconsin");
printf("District: %d\n", senator.district);
//whoa, what's the int value of Wisconsin?
```

- But… the other one becomes unpredictable

What's in the union?

- How can you tell what's in the union?
  - You can't!
- You need to keep separate information that says what's in the union
- Anonymous (unnamed) unions inside of structs are common

```
struct Congressperson
{
      int senator;         //which one?
      union
      {
           int district;   //representatives
           char state[15]; //senators
      };
};
```

Operands and operators

- We could use such a struct to store terms in an algebraic expression
- Terms are of the following types
  - Operands are double values
  - Operators are char values: **+**, **-**, *****, and **/**

```
typedef enum { OPERATOR, OPERAND } Type;
typedef struct
{
     Type type;
     union
     {
           double operand;
           char operator;
     };
} Term;
```

Stack

- A stack is a simple (but useful) data structure that has three basic operations:
  - **Push** Put an item on the top of the stack
  - **Pop** Remove an item from the top of the stack
  - **Top** Return the item currently on the top of the stack
- This kind of data structure is sometimes referred to as an **Abstract Data Type** (ADT)
- We don't actually care how the ADT works, as long as it supports certain basic operations

Stack of double values

- We can implement a stack of double values in a very similar way to the contact list from a couple of labs ago

```
typedef struct
{
     double* values;
     int size;
     int capacity;
} Stack;
```

Stack initialization

- Initializing the stack isn't hard
  - We give it an initial capacity (perhaps 5)
  - We allocate enough space to hold that capacity
  - We set the size to 0

```
Stack stack;
stack.capacity = 5;
stack.values = (double*)
     malloc(sizeof(double)*stack.capacity );
stack.size = 0;
```

Push, pop, and top

- We can write simple methods that will do the operations of the stack ADT

```
void push(Stack* stack, double value);
```

```
double pop(Stack* stack);
```

```
double top(Stack* stack);
```

Postfix notation

- You might recall postfix notation from CS221
  - It is an unambiguous way of writing mathematical expressions
- Whenever you see an operand, put it on the stack
- Whenever you see an operator, pop the last two things off the stack, perform the operation, then put the result back on the stack
- The last thing should be the result
- Example: **`5 6 + 3 –`** gives (5 + 6) – 3 = 8

Evaluate postfix

- Finally, we have enough machinery to evaluate an array of postfix terms
- Write the following function that does the evaluation:

```
double evaluate(Term terms[], int size);
```

- We'll have to see if each term is an operator or an operand and interact appropriate with the stack

Quiz

Upcoming

Next time…

- Low-level file I/O

Reminders

- Work on Project 5
  - Due on Friday
- Read LPI Chapters 4 and 5