Recursive structures
(aka self-referential structures)

• What is the meaning of

```c
struct rec { int i; struct rec r; }
```

• A structure can not refer itself directly.
• The only way to create a recursive structure is to use pointers

```c
struct node {
    char * word;
    int count;
    struct node *left,*right;
}
```

• The tag is useful here

Run recurrence.c to see the size of the struct
Unions

- `typedef union {int units; float kgs;} amount;`

- Unions can hold different type of values at different times
- Definition similar to a structure but
  - storage is shared between members
  - only one field present at a time
  - programmers must keep track of what it is stored
- Useful for defining values that range over different types
  - Critically, the memory allocated for these types is shared
- Memory layout
  - All members have offset zero from the base
  - Size is big enough to hold the widest member
  - The alignment is appropriate for all the types in the union
Union operations

• The same operations as the ones on structures
  – Assignment,
  – Copying as a unit
  – Taking the address
  – Accessing a member
• Can be initialized with a value of the type of its first member
• Example

```c
typedef union { int units; float kgs; } amount;

typedef struct {
    char item[15]; float price;
    int type; amount qty;
} product;
```
Safety

• C provides no guarantees that unions are correctly accessed (see union.c *)

```c
void print(amount x){printf("%d\n", x.units);}  
int main () {  
    product p[10];  
    memcpy(p[0].item, "toys", strlen("toys")+1);  
p[0].price = 2.0;  
p[0].type = 2;  
p[0].qty.kgs = 3.0;  
print(p[0].qty);  /* see the garbled print */
}
```
void check(int len, product* store) {
    for (int i=0; i<len; i++) {
        printf("%s\n", store->item);
        switch (store->type) {
        case 1:
            printf("%d units\n", store->qty.units);
            store++;
            break;
        case 2:
            printf("%f kgs\n", store->qty.kgs);
            store++;
            break;
        }
    }
}
Function Pointers

• C permits functions to be treated like any other data object
  – A function pointer can be
    • supplied as an argument
    • returned as a result
    • stored in any array
    • compared

• We’ll use this for Lab 6 (cypher)

• Main caveat:
  – *function_pointer does not have an lvalue
Example: Operating on a List

- Operating on a list of integers...

```c
union VAL {int i_val;
    float f_val;}
struct list {
    union VAL v;
    struct list * next;
};

typedef struct list List;
```
Creating lists...

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

List *makeList(int n) {
    List *l, *l1 = NULL; int i;
    for (i = 0; i < n; i++) {
        l = malloc(sizeof(List));
        l->v.i_val = 0;
        l->next = l1;
        l1 = l;
    }
    return l;
}
```

Given a number n, build a list of length n init to 0; (picture)
#include <stdio.h>
#include <stdlib.h>

List *makeList(int n) {
    List *l, *l1 = NULL; int i;
    for (i = 0; i < n; i++) {
        l = malloc(sizeof(List));
        l->v.f_val = 1.0;
        l->next = l1; l1 = l;
    }
    return l;
}

Given a number n, build a list of length n init to 1 (picture)
Creating lists... (3)

• We can imagine many different ways of populating a list
  – The overall control structure remains the same
  – Only the computation responsible for producing the next element changes

• How can we abstract the definition to reuse the same control structure for the different kinds of lists we might want?
Function pointers

• Supply a pointer to the function that computes values

```c
void init_i(union VAL *v){
    v->i_val = 0;
}
```

```c
void init_f(union VAL *v) {
    v->f_val = 1.0;
}
```
Abstraction revisited

```c
List *makeGenList (int n, void (*f)(union VAL *))
{
    List * l, *ll = NULL; int i;
    for (i = 0; i < n; i++) {
        l = (List*) malloc(sizeof(List));
        (*f>(&l->v));
        l->next = ll; ll = l;
    }
    return l;
}

makeGenList(10,init_i);
makeGenList(10,init_f);
```

Expects a function pointer that points to a function which yields an int, and which expects an int argument

Applies (invokes) the function pointed to by f with argument n
Lab 6

• Use function pointers
• Use bit-wise operations
• We now give some examples of bit-wise operations
Bitwise operations

- Boolean algebra has basic operations for manipulating sets
  - \& bit-wise AND: set intersection
  - | bit-wise OR: set union
  - ^ bit-wise exclusive OR: which members are different?
  - ~ set complement

- A set of n elements can be represented as a vector of n bits
  - \{0, 1, 4\} can be represented 00001011

- Any integral type can be treated as a bit vector
  - preferably use unsigned values

\[
\begin{array}{c}
01101001 \ & 01101001 \ & 01101001 \\
& 01010101 \ & | \ & 01010101 \ & ^ \ & 01010101 \ & ~ \ & 01010101 \\
01000001 \ & 01111101 \ & 00111100 \ & 10101010
\end{array}
\]
XOR as a cypher operation

• In data encryption algorithms, the bit wise XOR is often used as an elementary coding function
• Let $D$ be an $n$-bit piece of data, $K$ be an $n$-bit key
• We can have $E = D \oplus K$ be the encrypted version of $D$.
• Notice that to decrypt such a simple encrypted data, we can simply do $E \oplus K$ to retrieve $D$
Bit shifting and rotation

• Bit shifting is useful for reading a particular bit in a word
  – E.g. to read the k-th bit of a word x (counting from the least significant bit position 0), we right shift x by k bit position, obtain y.
  – We then do y & 1 to get z. If z == 1, then the k-th bit of x is 1. Otherwise it is 0. Why?

• Another elementary cypher operation could be bit rotation
An example of bit rotation

/* Purpose: showing result of bit rotation */
#include <stdio.h>

main() {
    char a = '\x0f';
    char b, c;
    
    unsigned char ua = '\x0f';
    unsigned char ub, uc;
    
    printf("a is \t %#X\n", a);
    b = a >> 2;
    printf("b is \t %#X\n", b);
    c = a << 6;
    printf("c is \t %#X\n", c);
    a = b | c;
    printf("a is \t %#X\n", a);
    printf("ua is \t %#X\n", ua);
    ub = ua >> 2;
    printf("ub is \t %#X\n", ub);
    uc = ua << 6;
    printf("uc is \t %#X\n", uc);
    ua = ub | uc;
    printf("ua is \t %#X\n", ua);
    ua = '\x0f';
    ua = ua >> 2 | ua << 6;
    printf("rotation of ua is \t%#X\n", (unsigned char) ua >> 2 | ua << 6);
    printf("rotation of ua is \t%#X\n", ua);
    /*
    printf("rotation of ua is \t%#X\n", (unsigned char) ua >> 2 | ua << 6);
    printf("rotation of ua is \t%#X\n", ua);
    */
    printf("rotation of ua is \t%#X\n", (unsigned char) ua >
        > 2 | ua << 6);
}