Self-Referential Structures

Illegal - infinite

```
struct SELF_REF {
        int      a;
        struct  SELF_REF  b;
        int      c;
} ;
```

Watch out

```
typedef struct {
        int      a;
        struct  SELF_REF  *b;
        int      c;
}  SELF_REF ;
```

Correction

```
struct SELF_REF {
        int      a;
        struct  SELF_REF  *b;
        int      c;
} ;
```

Correction

```
typedef struct SELF_REF_TAG {
        int      a;
        struct  SELF_REF_TAG  *b;
        int      c;
}  SELF_REF ;
```

Incomplete Declarations

- Structures that are mutually dependent
- As with self referential structures, at least one of the structures must refer to the other only through pointers
- So, which one gets declared first???

```
struct B;

struct A  {
      struct  B    *partner;
      /* etc */
} ;

struct B  {
      struct  A    *partner;
      /* etc */
} ;
```

- **Declares an identifier to be a structure tag**
- **Use this tag in declarations where the size of the structure is not needed (pointer!)**
- **Needed in the member list of A**

- **Doesn't have to be a pointer**

Initializing Structures

- Missing values cause the remaining members to get default initialization… whatever that might be!

```
typedef struct  {
    int        a;
    char      b;
    float      c;
}  Simple;

struct INIT_EX  {
    int          a;
    short      b[10];
    Simple  c;
}  x = {  10,
          { 1, 2, 3, 4, 5 } ,
          { 25, 'x', 1.9 }
} ;
```

What goes here (hint in blue below)?

```
struct INIT_EX   y = { 0 , {10,  20, 30, 40, 50,
                              60, 70, 80, 90, 100 },
                            { 1000, 'a', 3.14 }
                        } ;
```

Name all the variables and their initial values:

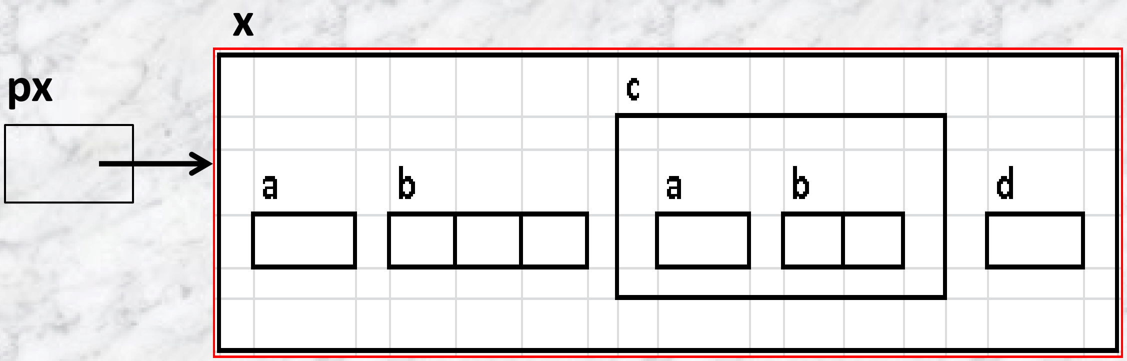
y.a = 0;
y.b[0] = 10; y.b[1] = 20; y.b[2] = 30; etc
y.c.a = 1000;
y.c.b = ‘a’;
y.c.c = 3.14;

Structure memory (again)

- What does memory look like?

```
typedef struct  {
    int        a;
    short    b[2];
} Ex2;

typedef struct EX  {
    int         a;
    char      b[3];
    Ex2        c;
    struct    EX   *d;
} Ex;
```



Given the following declaration, fill in the above memory locations:

Ex x = { 10, “Hi”, { 5 , { -1, 25 } } , 0 };

Ex *px = &x;

Structures as Function arguments

- Legal to pass a structure to a function similar to any other variable but often inefficient

```
/* electronic cash register individual
transaction receipt */
#define PRODUCT_SIZE   20;
typedef struct  {
    char     product[PRODUCT_SIZE];
    int       qty;
    float     unit_price;
    float     total_amount;
}  Transaction;
```

- Function call:
  - print_receipt(current_trans);
  - Copy by value copies 32 bytes to the stack which can then be discarded later
- Instead…
  - (Transaction *trans)
  - trans->product   // fyi: (*trans).product
  - trans->qty
  - trans->unit_price
  - trans->total_amount
  - print_receipt(&current_trans);
  - void print_receipt(Transaction *trans)

```
void print_receipt (Transaction trans)  {
  printf(“%s\n, trans.product);
  printf(%d @ %.2f total %.2f\n”, trans.qty, trans.unit_price, trans.total_amount);
}
```

Dynamic Memory Allocation (again?!)

- Dynamic allocation allows a program to create space for a structure whose size isn't known until runtime.
  - memory is more explicitly (but more flexibly) managed, typically, by allocating it from the *heap*, an area of memory structured for this purpose.
- The ***malloc*** and ***calloc*** functions both allocate memory and return a **void** pointer to it; NULL is returned if the requested allocation could not be performed (in stdlib.h)… MUST check for this!
  - malloc
    - Argument: # of bytes needed
    - Leaves the memory uninitialized
  - calloc
    - Arguments: number of elements AND the size of each element
    - Initializes the memory to zero before returning
- The ***free*** function
  - You may not pass a pointer to this function that was not obtained from an earlier call to malloc/calloc.
  - Memory must not be accessed after it has been freed.
- Memory Leaks
  - Memory that has been dynamically allocated but has not been freed and is no longer in use.
  - Negative because it increases the size of the program and lead to problems.

DMA Example

Set each element of the newly allocated integer array of five elements to zero instead of declaring int_array[5]

```
int  *pi_save, *pi;
pi = malloc(20);

if (pi == NULL)
{
    printf("Out of memory!\n");
    exit(1);
}

for (int x = 0;  x < 5; x +=1)
    *pi++ = 0;

// print
```

QUESTIONS

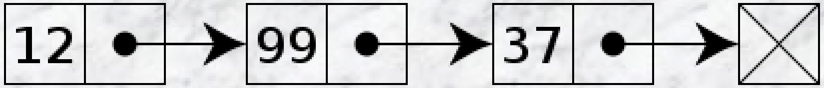
1. What are the values in the new memory before initializing to zero?
2. Where is pi pointing to after the for loop?
3. What does the print loop look like?
4. How update to use calloc?
5. How free the memory?

(see dma1.c)

Linked List Node structure

- A linked list is…a data structure consisting of a group of nodes which together represent a sequence
- Simply, each node is composed of a data and a reference (in other words, a *link*) to the next node in the sequence
- Allows for efficient insertion or removal of elements from any position in the sequence (vs an array).
- Data items need not be stored contiguously in memory
- Major Disadvantage:
  - does not allow random access to the data or any form of efficient indexing

```
/* Node Structure */
struct node {
    int data;
    struct node *next; }
```



A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

DMA structure (linked list)

```
struct myRecord {
    char firstName[20];
    char lastName[25];
    int employeeID;
    struct myRecord * nextRecord;
};

// point to first structure in the list
struct myRecord  *headPtr;

headPtr = (struct myRecord *) malloc(sizeof(myRecord));
// when allocate another structure,
// the pointer returned should be assigned to the first record’s pointer
```

(see linklst1.c)