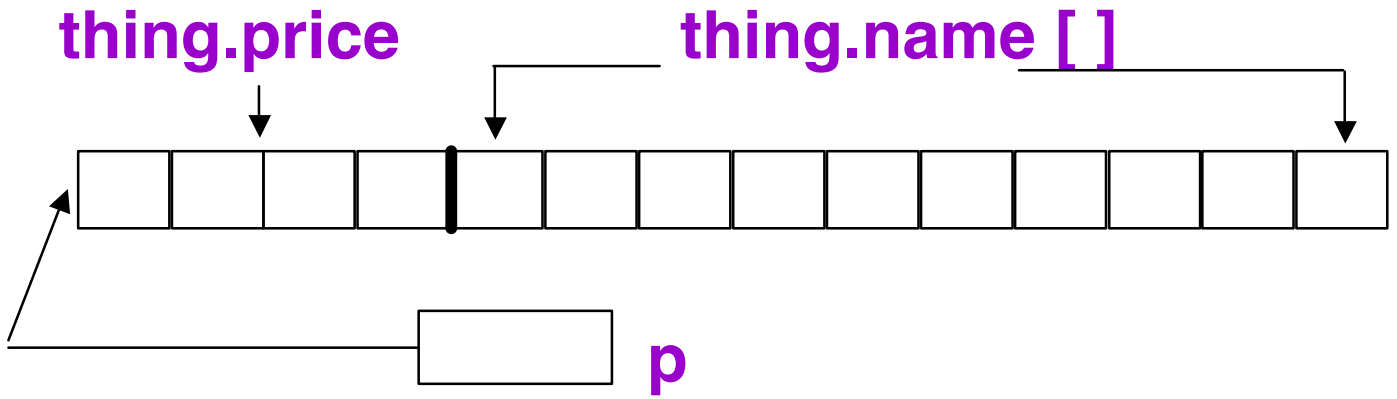
Pointers to Structures

```
struct part {
   float price ;
   char name [10] ;
} ;
struct part *p , thing;
p = &thing;
/* The following three statements are equivalent *
thing.price = 50;
(*p).price = 50;   /* () around *p is needed */
p -> price = 50;
```

Pointers to Structures



- p is set to point to the first byte of the struct variable

Pointers to Structures

```
struct part * p, *q;
p = (struct part *) malloc( sizeof(struct part) );
q = (struct part *) malloc( sizeof(struct part) );
p -> price = 199.99 ;
strcpy( p -> name, "hard disk" );
(*q) = (*p);
q = p;
free(p);
free(q); /* This statement causes a problem !!!
            Why? */
```

Pointers to Structures

- You can allocate a structure array as well:

```
{
    struct part *ptr;
    ptr = (struct part *) malloc(10 * sizeof(struct part) );
    for( i=0; i< 10; i++)
    {
            ptr[ i ].price = 10.0 * i;
            sprintf( ptr[ i ].name, "part %d", i );
    }
    ……
    free(ptr);
}
```

Pointers to Structures

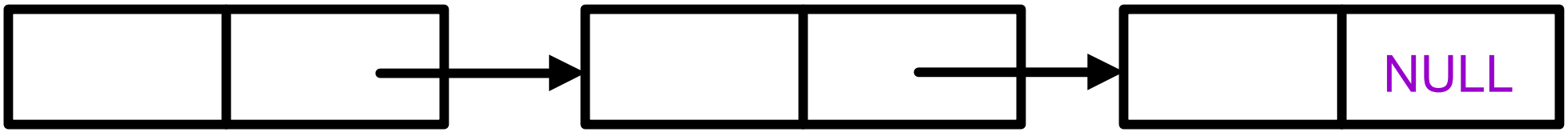
- You can use pointer arithmetic to access the elements of the array:

```
{
    struct part *ptr,  *p;
    ptr = (struct part *) malloc(10 * sizeof(struct part) );
    for( i=0, p=ptr; i< 10; i++, p++)
    {
            p -> price = 10.0 * i;
            sprintf( p -> name, "part %d", i );
    }
    ……
    free(ptr);
}
```

Pointer as Structure Member

```
struct node{
   int data;
   struct node *next;
};
struct node a,b,c;
a.next = &b;
b.next = &c;
c.next = NULL;
```

```
a.data = 1;
a.next->data = 2;
/* b.data =2 */
a.next->next->data = 3;
/* c.data = 3 */
c.next = (struct node *)
   malloc(sizeof(struct
   node));
……
```



Assignment Operator vs. memcpy

- This assign a struct to another

```
{
   struct part a,b;
   b.price = 39.99;
   b.name = "floppy";
   a = b;
}
```

- Equivalently, you can use memcpy

```
#include <string.h>
……
{
    struct part a,b;
    b.price = 39.99;
    b.name = "floppy";
    memcpy(&a,&b,sizeof(part));
}
```

Array Member vs. Pointer Member

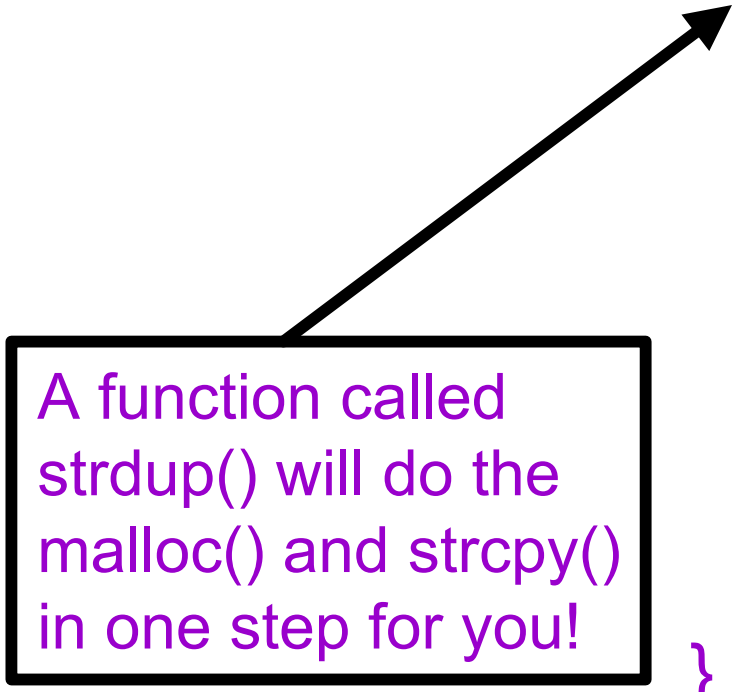
```
struct book {
   float price;
   char name[50];
};
```

```
int main()
{
    struct book a,b;
    b.price = 19.99;
    strcpy(b.name, "C handbook");
    a = b;
    strcpy(b.name, "Unix
    handbook");
    puts(a.name);
    puts(b.name);
}
```

Array Member vs. Pointer Member

```
struct book {
   float price;
   char *name;
};
```

```
int main()
{
   struct book a,b;
   b.price = 19.99;
   b.name = (char *) malloc(50);
   strcpy(b.name, "C handbook");
   a = b;
   strcpy(b.name, "Unix handbook");
   puts(a.name);
   puts(b.name);
   free(b.name);
}
```



Passing Structures to Functions (1)

- Structures are passed by value to functions
  - The parameter variable is a local variable, which will be assigned by the value of the argument passed.
  - Unlike Java.
- This means that the structure is copied if it is passed as a parameter.
  - This can be inefficient if the structure is big.
    - In this case it may be more efficient to pass a pointer to the struct.
- A struct can also be returned from a function.

Passing Structures to Functions (2)

```
struct book {
    float price;
    char abstract[5000];
};
void print_abstract( struct
    book *p_book)
{
    puts( p_book->abstract );
};
```

```
struct pairInt {
    int min, max;
};
struct pairInt min_max(int x,int y)
{
    struct pairInt pair;
    pair.min = (x > y) ? y : x;
    pair.max = (x > y) ? x : y;
    return pairInt;
}
int main(){
    struct pairInt result;
    result = min_max( 3, 5 );
    printf("%d<=%d", result.min,
    result.max);
}
```