Structures

Structures (1)

- Structures are C's way of grouping collections of data into a single manageable unit.
  - This is also the fundamental element of C upon which most of C++ is built (i.e., classes).
  - Similar to Java's classes.
- An example:
  - Defining a structure type:

```
struct coord {
        int x ;
        int y ;
};
```

  - This defines a new type struct coord. No variable is actually declared or generated.

Structures (2)

- Define struct variables:

```
struct coord {
      int x,y ;
  } first, second;
```

- Another Approach:

```
struct coord {
      int x,y ;
  };

.............
struct coord first, second;  /* declare variables */
struct coord third;
```

Structures (3)

- You can even use a typedef if your don't like having to use the word "struct"

```
typedef struct coord coordinate;

coordinate first, second;
```

- In some compilers, and all C++ compilers, you can usually simply say just:

```
coord first, second;
```

Structures (4)

- Access structure variables by the dot (.) operator
- Generic form:

  structure_var.member_name
- For example:

  first.x = 50 ;

  second.y = 100;
- These member names are like the public data members of a class in Java (or C++).
  - No equivalent to function members/methods.
- struct_var.member_name can be used anywhere a variable can be used:
  - printf ("%d , %d", second.x , second.y);
  - scanf("%d, %d", &first.x, &first.y);

Structures (5)

- You can assign structures as a unit with =

```
first = second;
```

instead of writing:

```
first.x = second.x ;
first.y = second.y ;
```

- Although the saving here is not great
  - It will reduce the likelihood of errors and
  - Is more convenient with large structures
- This is different from Java where variables are simply references to objects.

```
first = second;
```

makes first and second refer to the same object.

Structures Containing Structures

- Any “type” of thing can be a member of a structure.
- We can use the coord struct to define a rectangle

```
struct rectangle {
   struct coord topleft;
   struct coord bottomrt;
} ;
```

- This describes a rectangle by using the two points necessary:

```
struct rectangle mybox ;
```

- Initializing the points:

```
mybox.topleft.x = 0 ;
mybox.topleft.y = 10 ;
mybox.bottomrt.x = 100 ;
mybox.bottomrt.y = 200 ;
```

An Example

```
#include <stdio.h>
struct coord {
    int x;
    int y;
};
struct rectangle {
    struct coord topleft;
    struct coord bottomrt;
};
```

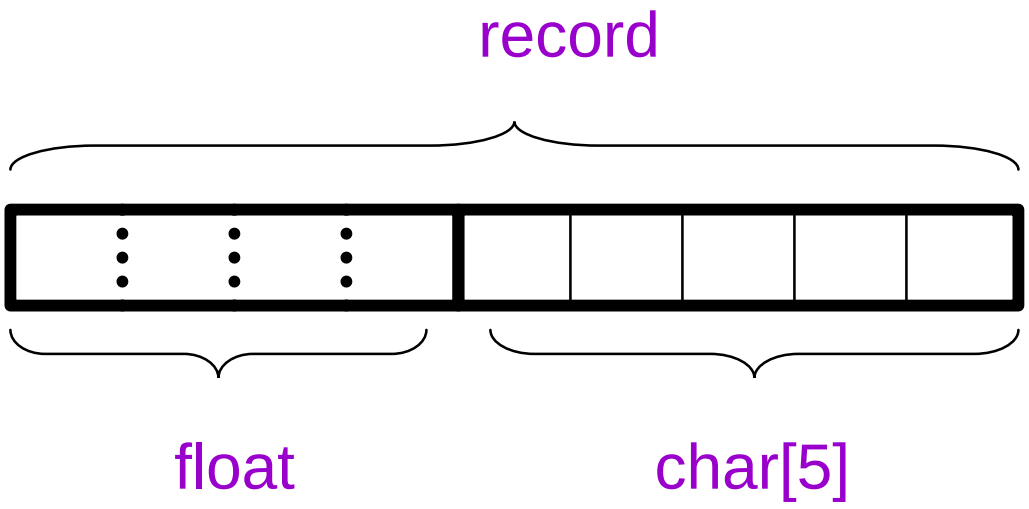
```
int main () {
    int length, width;
    long area;
    struct rectangle mybox;
    mybox.topleft.x = 0;
    mybox.topleft.y = 0;
    mybox.bottomrt.x = 100;
    mybox.bottomrt.y = 50;
    width = mybox.bottomrt.x –
            mybox.topleft.x;
    length = mybox.bottomrt.y –
            mybox.topleft.y;
    area   = width * length;
    printf ("The area is %ld units.\n",
    area);
}
```

Structures Containing Arrays

- Arrays within structures are the same as any other member element.
- For example:

```
struct record {
        float  x;
        char  y [5] ;
  } ;
```

- Logical organization:



An Example

```
#include <stdio.h>
struct data {
    float amount;
    char  fname[30];
    char lname[30];
}  rec;
int main () {
     struct data rec;
     printf ("Enter the donor's first and last names, \n");
     printf ("separated by a space: ");
     scanf  ("%s %s", rec.fname, rec.lname);
     printf ("\nEnter the donation amount: ");
     scanf  ("%f", &rec.amount);
     printf ("\nDonor %s %s gave $%.2f.\n",
                rec.fname,rec.lname,rec.amount);
}
```

Arrays of Structures

- The converse of a structure with arrays:
- Example:

```
struct  entry {
   char  fname [10] ;
   char  lname [12] ;
   char phone [8] ;
} ;
struct entry list [1000];
```

- This creates a list of 1000 identical entry(s).
- Assignments:

```
list [1] = list [6];
strcpy (list[1].phone, list[6].phone);
list[6].phone[1] = list[3].phone[4] ;
```

An Example

```
#include <stdio.h>
struct entry {
    char fname [20];
    char lname [20];
    char phone [10];
} ;
```

```
int main() {
   struct entry list[4];
   int i;
   for (i=0; i < 4; i++) {
       printf ("\nEnter first name: ");
       scanf  ("%s", list[i].fname);
       printf ("Enter last name: ");
       scanf  ("%s", list[i].lname);
       printf ("Enter phone in 123-4567 format: ");
       scanf  ("%s", list[i].phone);
   }
   printf ("\n\n");
   for (i=0; i < 4; i++) {
       printf ("Name: %s %s", list[i].fname, list[i].lname);
       printf ("\t\tPhone: %s\n", list[i].phone);
   }
}
```

Initializing Structures

- Simple example:

```
struct sale {
   char  customer [20] ;
   char  item [20] ;
   int amount ;
};

struct sale mysale = { "Acme Industries",
                       "Zorgle blaster",
                       1000 } ;
```

Initializing Structures

- Structures within structures:

```
struct  customer {
   char firm [20] ;
   char contact [25] ;
};
struct sale {
   struct customer buyer ;
   char item [20] ;
   int amount ;
} mysale =
{ { "Acme Industries", "George Adams"} ,
    "Zorgle Blaster",  1000
} ;
```

Initializing Structures

- Arrays of structures

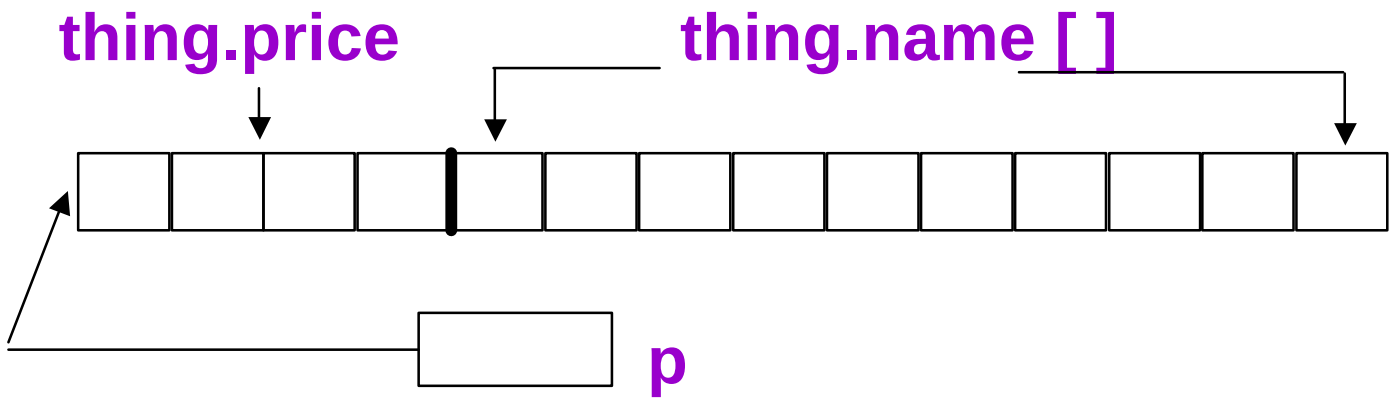
```
struct customer {
    char firm [20] ;
    char contact [25] ;
} ;
struct sale {
    struct customer buyer ;
    char item [20] ;
    int amount ;
} ;
```

```
struct sale y1990 [100] = {
    {  { "Acme Industries",
    "George Adams"} ,
    "Left-handed Idiots" ,
    1000
    },
    {  { "Wilson & Co.",
         "Ed Wilson"} ,
    "Thingamabob" , 290
    }
} ;
```

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

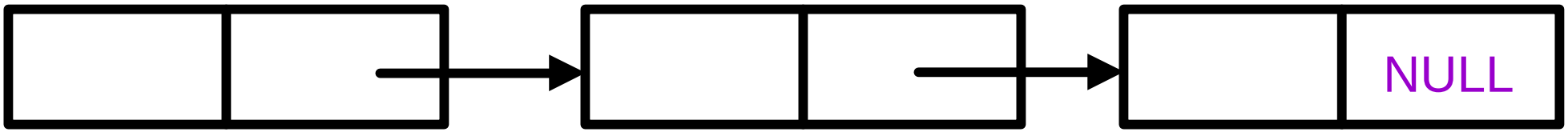
- You can use pointer arithmetic to access the elements of the array:

```
{
    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

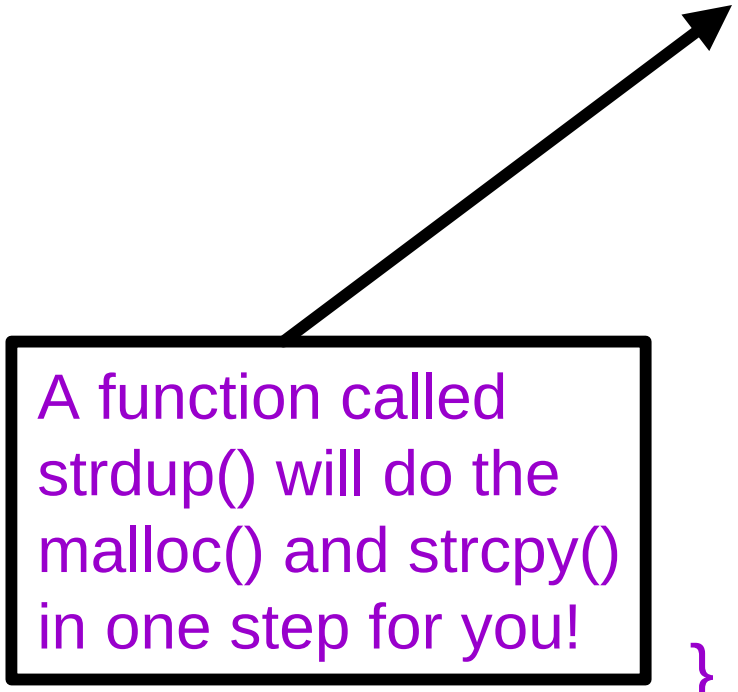
```
struct book {
   float price;
   char name[50];
};
```

```
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);
}
```