CSE-105

Lecture - 19

Structures, Unions, Typedefs & Bit Fields

Course Instructor : Md.Shamsujjoha

Structures and Unions

- Essential for building up “interesting” data structures — e.g.,
    - Data structures of multiple values of different kinds
    - Data structures of indeterminate size
- Essential for solving “interesting” problems
    - Most of the “real” problems in the *C* world

Definition — *Structure*

- A collection of one or more variables, typically of different types, grouped together under a single name for convenient handling

- Known as **`struct`** in *C* and *C*++

struct

- Defines a new *type*
  - A new kind of data type that compiler regards as a unit.

```
struct motor {
  float volts;     //voltage of the motor
  float amps;      //amperage of the motor
  int phases;      //# of phases of the motor
  float rpm;       //rotational speed of motor
};        //struct motor
```

struct

- Defines a new *type*

Name of the type

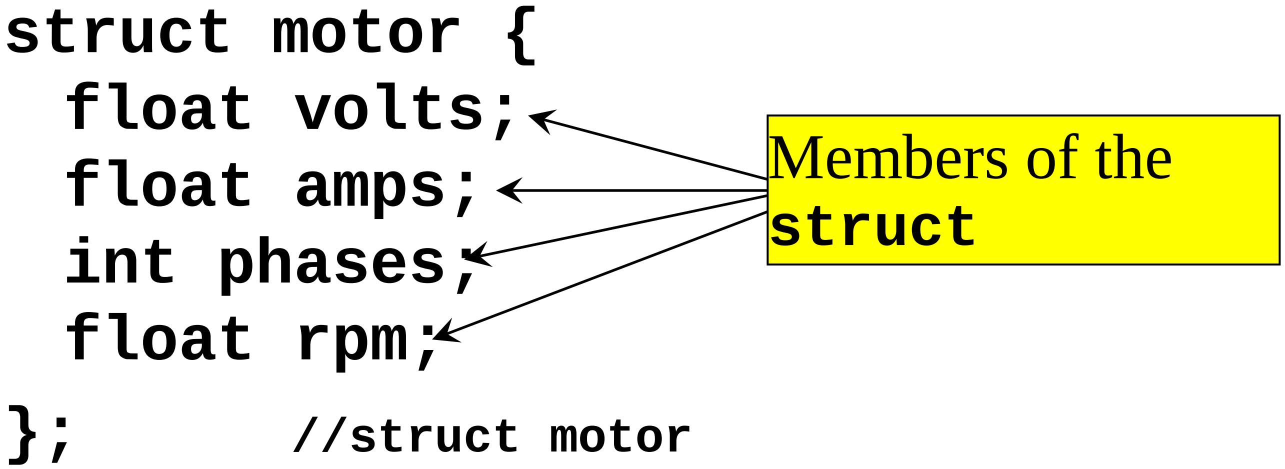
tag

Note:– name of type is optional if you are just declaring a single **struct**

```
struct motor {
  float volts;       //voltage of the motor
  float amps;        //amperage of the motor
  int phases;        //# of phases of the motor
  float rpm;         //rotational speed of motor
};         //struct motor
```

struct

- Defines a new *type*



```
struct motor {
  float volts;
  float amps;
  int phases;
  float rpm;
};        //struct motor
```

Members of the **struct**

Declaring `struct` variables

```
struct motor p, q, r;
```

- Declares and sets aside storage for three variables – `p`, `q`, and `r` – each of type `struct motor`

```
struct motor M[25];
```

- Declares a 25-element array of `struct motor`; allocates 25 units of storage, each one big enough to hold the data of one `motor`

```
struct motor *m;
```

- Declares a pointer to an object of type `struct motor`

Structures

```
struct ADate {
   int  month;
   int  day;
   int  year;
};

struct ADate date;

date.month = 9;
date.day = 1;
date.year = 2005;
```

To display the screen locations stored in the structure Adate,

printf("%d, %d, %d", **date.month, date.day, date.year**);

What are the Advantage ??

```
struct ADate {
   int  month;
   int  day;
   int  year;
};

struct ADate date1, date2;

date1.month = 9;
date1.day = 1;
date1.year = 2005;

date2 = date1 ;

date2.month = date1.month;
date2.day = date1.day;
date2.year = date1.year;
```

More Examples

-
  ```
  struct SSN {
              int first_three;
              char dash1;
              int second_two;
             char dash2;
             int last_four;
          };
  struct  SSN  customer_ssn ;
  ```

-
  ```
  struct date {
             char month[2];
             char day[2];
             char year[4];
             } current_date ;
  ```

-
  ```
  struct time {
                 int hours;
                int minutes;
                int seconds;
               }  time_of_birth = { 8, 45, 0 };
  ```

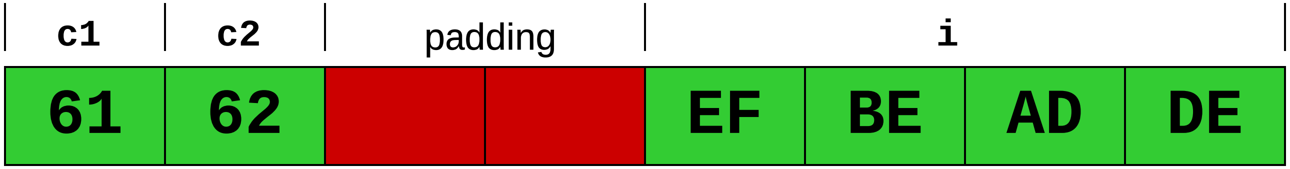
Structure Representation & Size

- `sizeof(struct …)` =
- sum of `sizeof(`field`)`
- + alignment padding

  Processor- and compiler-specific

```
struct CharCharInt {
   char  c1;
   char  c2;
   int   i;
} foo;

foo.c1 = ’a’;
foo.c2 = ’b’;
foo.i  = 0xDEADBEEF;
```



c1	c2	padding		i			
61	62	🟥	🟥	EF	BE	AD	DE

x86 uses “little-endian” representation

Accessing Members of a **`struct`**
`Repeat`

```
struct motor {
float volts;
float amps;
int phases; otor
float rpm;
};
```

- Let

```
struct motor p;
struct motor q[10];
```

- Then

`p.volts` — is the voltage
`p.amps` — is the amperage
`p.phases` — is the number of phases
`p.rpm` — is the rotational speed

`q[i].volts` — is the voltage of the **`i`**th motor
`q[i].rpm` — is the speed of the **`i`**th motor

Accessing Members of a **`struct`** (continued)

- Let

  `struct motor *p;`

- Then

  Why the parentheses?

  `(*p).volts` — is the voltage of the **`motor`** pointed to by **`p`**

  `(*p).phases` — is the number of phases of the **`motor`** pointed to by **`p`**

Accessing Members of a `struct` (continued)

- Let

```
struct motor *p;
```

- Then

Because '.' operator has higher precedence than unary '*'

`(*p).volts` — is the voltage of the **`motor`** pointed to by **`p`**

`(*p).phases` — is the number of phases of the **`motor`** pointed to by **`p`**

Accessing Members of a `struct` (continued)

- Let

  ```
  struct motor *p;
  ```

- Then

  `(*p).volts` — is the voltage of the `motor` pointed to by `p`

  `(*p).phases` — is the number of phases of the `motor` pointed to by `p`

Reason:– you really want the expression

```
m.volt * m.amps
```

to mean what you think it should mean!

Accessing Members of a **`struct`** (continued)

- The **`(*p).member`** notation is a nuisance
  - Clumsy to type; need to match **`( )`**
  - Too many keystrokes

- This construct is so widely used that a special notation was invented, i.e.,
  - **`p->member`**, where **`p`** is a pointer to the structure

Previous Example Becomes …

- Let

  `struct motor *p;`

- Then

  `p -> volts` — is the voltage of the **`motor`** pointed to by **`p`**

  `p -> phases` — is the number of phases of the **`motor`** pointed to by **`p`**

Operations on `struct`

- Copy/assign

```
struct motor p, q;
p = q;
```

- Get address

```
struct motor p;
struct motor *s
s = &p;
```

- Access members

```
p.volts;
s -> amps;
```

Initialization of a `struct`

- Let

```
struct motor {
          float volts;
          float amps;
          int phases;
          float rpm;
};              //struct motor
```

- Then

```
struct motor m = {208, 20, 3, 1800};
```

initializes the `struct`

Why **`structs`** AGAIN???

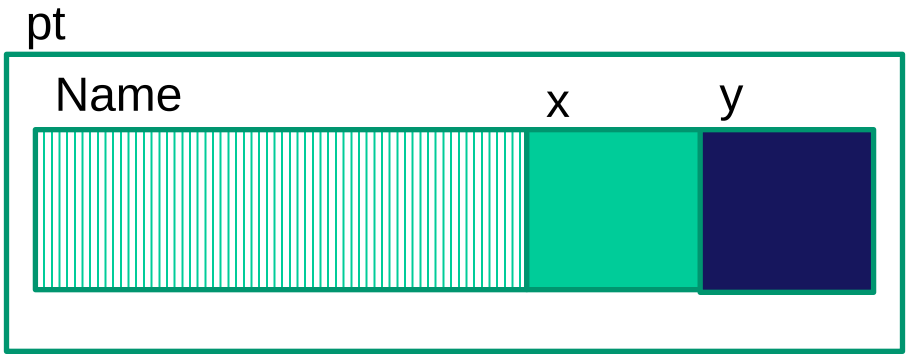
- Open-ended data structures
  - E.g., structures that may grow during processing
  - Avoids the need for **`realloc()`** and a lot of copying

- Self-referential data structures
  - Lists, trees, etc.

Nesting Structures

```
struct Point {
        char name[30];
        int x;
        int y;
};
```

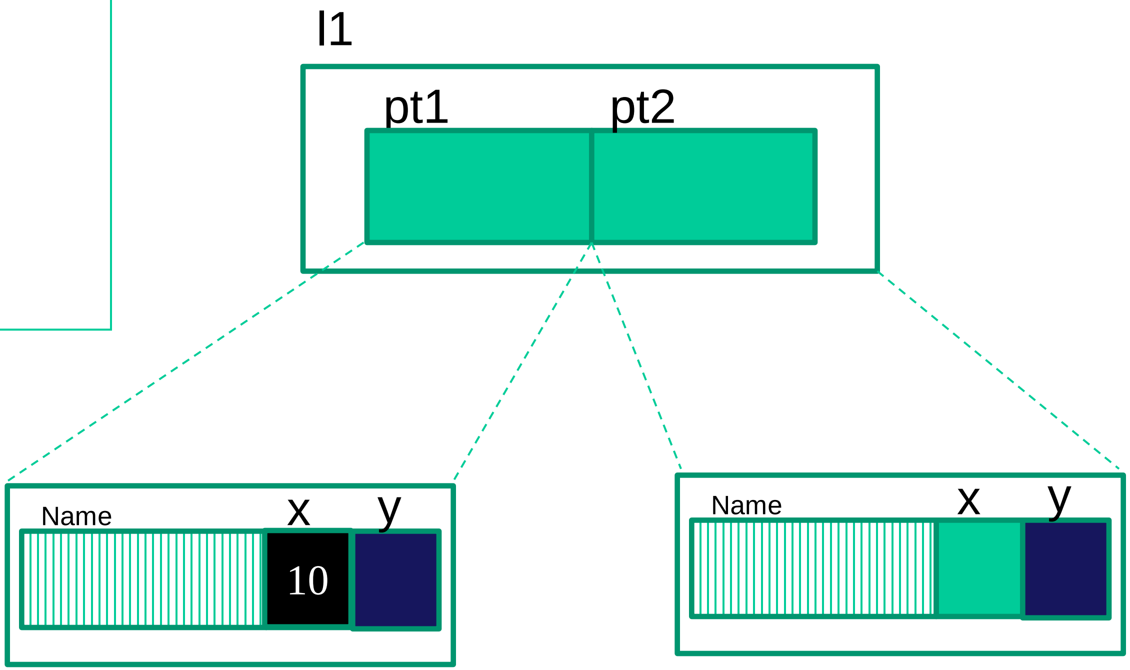
struct Point pt;



```
struct Line {
        struct Point pt1;
        struct Point pt2;
};
struct Line l1;
```

Nesting Structures

```
struct Point {
        char name[30];
        int x;
        int y;
};
```



```
struct Line {
        struct Point pt1;
        struct Point pt2;
};
struct Line l1;
```

To Access the Elements

l1.pt1.x=10;

Array of Structures

```
struct Point {
        char name[30];
        int x;
        int y;
};
```

- Array of Structures act like any other array.

```
struct Point pt[3];
```

```
pt[0].name = “A”;
pt[0].x = 0;
pt[0].y = 1;
```

```
pt[1].name = “B”;
pt[1].x = 4;
pt[1].y = 1;
```

```
pt[2].name = “mid”;
pt[2].x = (pt[0].x + pt[1].x)/2;
pt[2].y = (pt[0].y + pt[1].y)/2;
```

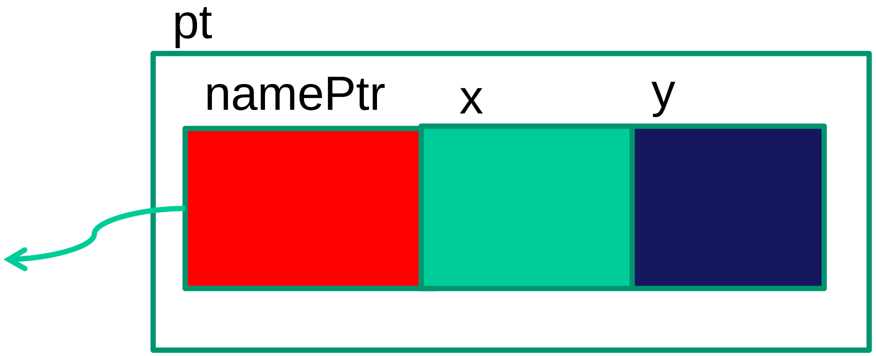
- Memory occupied: the dimensions of the array multiply by sizeof(struct tag)
  - (Remember) sizeof() is compile time function

Pointers in Structures

- A structure can have a pointer as its member

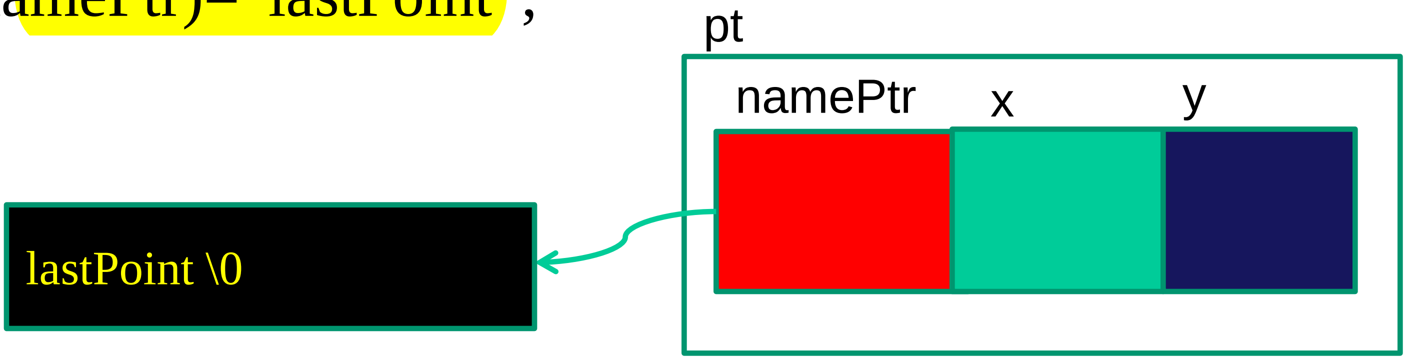
```
struct Point {
                char *namePtr;
                int x;
                int y;
};
```

struct Point pt;



pt.namePtr=(char *) malloc(20*sizeof(char));

*(pt.namePtr)=“lastPoint”;



Pointer to Structures

- A pointer to a structure can be defined

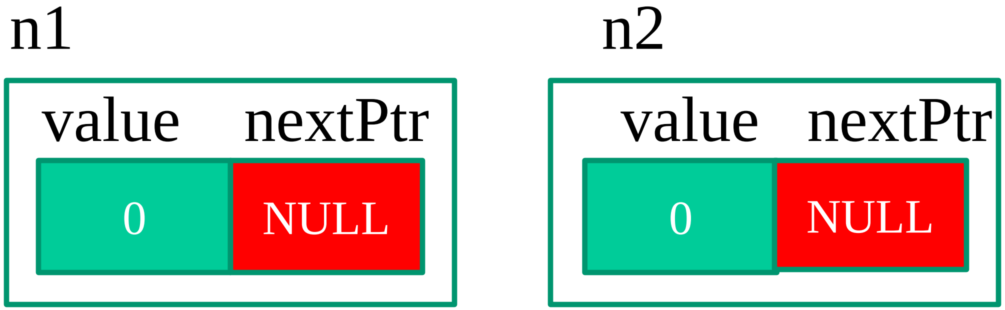
```
struct Point p1, *ptr;
ptr=&p1;
```

p1.x=10 ≡ ptr→x =10 ≡ (*ptr).x=10 ≡ (&p1)→x = 10

Self referencing Structures

- Useful in data structures like trees, linked lists.
- It is illegal for a structure to contain an instance of itself.
  - Solution: Have a pointer to another instance.

```
struct lnode {                          /* the linked list node */
          int value;
          struct lnode *nextPtr; /* pointer to next node */
} n1,n2;
```



n1

value nextPtr

0 NULL

n2

value nextPtr

0 NULL

Example

```
struct item {
  char *s;
  struct item *next;
}
```

Yes! This is legal!

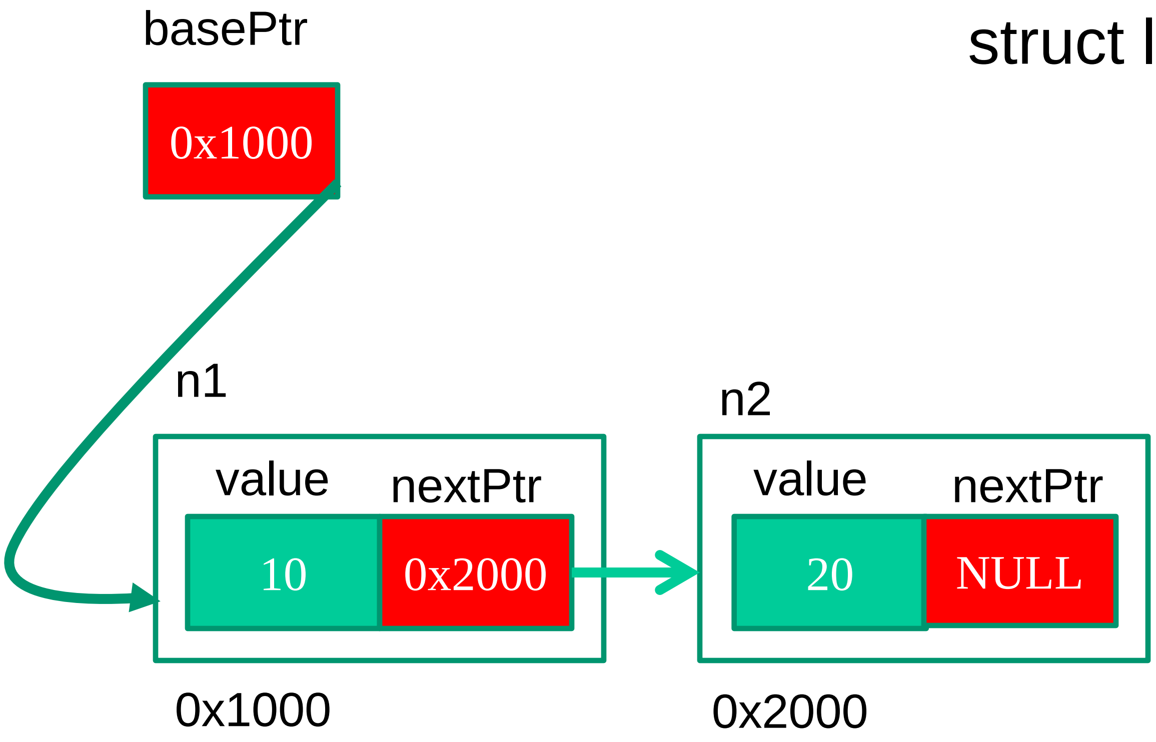
- I.e., an **`item`** can point to another **`item`**
- … which can point to another **`item`**
- … which can point to yet another **`item`**
- … etc.

Thereby forming a *list* of **`items`**

Self referencing Structures

```
struct lnode {
            int value;
            struct lnode *nextPtr;
            } n1,n2;
```

```
n1.value=10;
n1.nextPtr=&n2;
n2.value=20;
n2.nextPtr=NULL;
struct lnode *basePtr=&n1;
```



Typedef

- Use **typedef** for creating new data type names

-
  ```
  typedef int length;
  ```

this the name length a synonym (alias) for int. Afterwards, you can do:

```
length x = 4;
```

- In context of structs, you can do:

```
struct Point {
int x;
int y;
};
typedef struct Point myPoint;
myPoint p1;
struct Point p2;
p1.x=10;
```

```
typedef struct Point *pointPtr;
pointPtr p1;
struct Point p2;
p2.x=20;
p1.x=10; ??
p1→x=10; ??
p1=&p2;
p1→x=10; ??
p1=(pointPtr) malloc(sizeof(struct Point));
p1→x=10; ??
```

```
typedef struct lnode {
          .
          .
} myNode;
myNode n1, *ptr;
```

```
typedef struct {
          .
          .
} myNode;
myNode n1, *ptr;
```

typedef (continued)

- **typedef** may be used to rename *any* type
  - Convenience in naming
  - Clarifies purpose of the type
  - Cleaner, more readable code
  - Portability across platforms
- E.g.,
  - **typedef char *String;**
- E.g.,
  - **typedef int size_t;**
  - **typedef long int32;**
  - **typedef long long int64;**

Very common in C and C++
Esp. for portable code!

These three may change from platform to platform

Defined once in a **.h** file!

Unions

- A union is a memory location that is shared by two or more different types of variables.

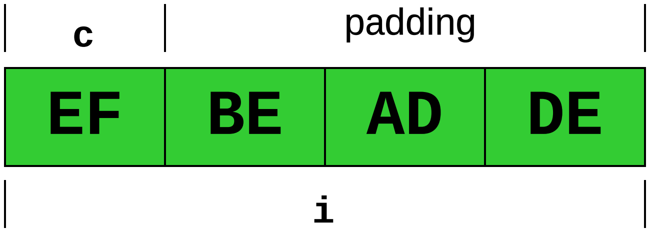
```
union u_tag {
        int ival;
        float fval;
        char cval;
} u;
```

- Each of ival, fval, cval have the same location in memory.
- `sizeof(union …)` = maximum of `sizeof(`field`)`
- Usage is similar to that of structs: `u.ival or u.cval`
- Up to programmer to determine how to interpret a union (i.e. which member to access) and used for low-level programming

Example

```
union AnElt {
   int   i;
   char  c;
} elt1, elt2;

elt1.i = 4;
elt2.c = ’a’;
elt2.i = 0xDEADBEEF;
```



c

padding

EF BE AD DE

i

Unions

- Storage
  - size of union is the size of its largest member
  - avoid unions with widely varying member sizes;

    for the larger data types, consider using pointers instead
- Initialization
  - Union may only be initialized to a value appropriate for the type of its first member

Bit-fields

- When storage is high cost affair, we need to use memory efficiently (e.g in embedded systems)

```
struct {
            unsigned pin1 : 1;
            unsigned pin2 : 2;
            unsigned pin3 : 1;
} flags;
```

- Here each of the element takes a bit of memory (1 bit)
- The number following the colons represent the field length in bits.

Looking Ahead

- The rest of this course is about *data structures* that you will typically encounter in *C* programs in your professional lives

- All of them involve **`structs`**.

Another note about **structs**

- The following is *not* legal:–

```
struct motor {
      float volts;
      float amps;
      float rpm;
      unsigned int phases;
  };  //struct motor
```

```
motor m;
motor *p;
```

You must write

```
   struct motor m;
   struct motor *p;
```

Revisit note about **`structs`** and pointers

- The following *is* legal:–

```
/* in a .c or .h file */
typedef struct _item Item;
Item *p, *q;

…    /* In another file */
struct _item {
  char *info;
  Item *nextItem;
};
```

Questions?