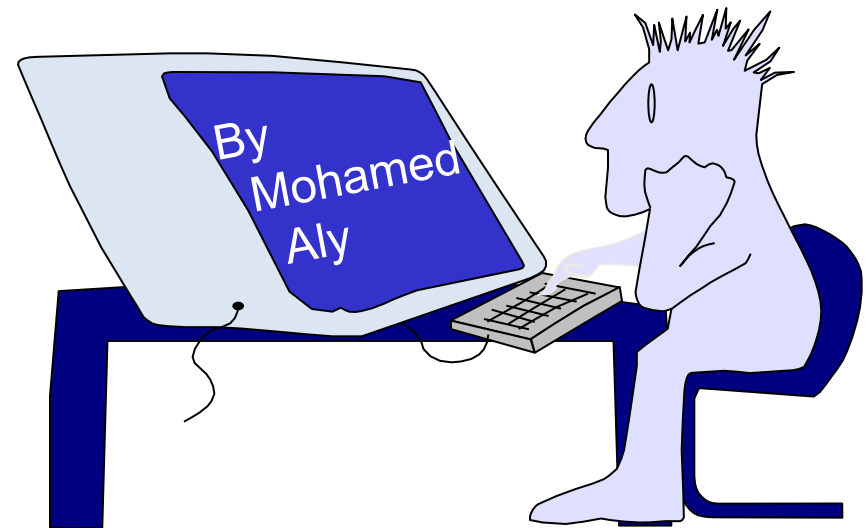
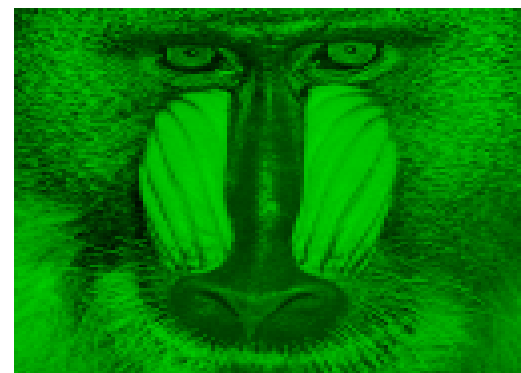
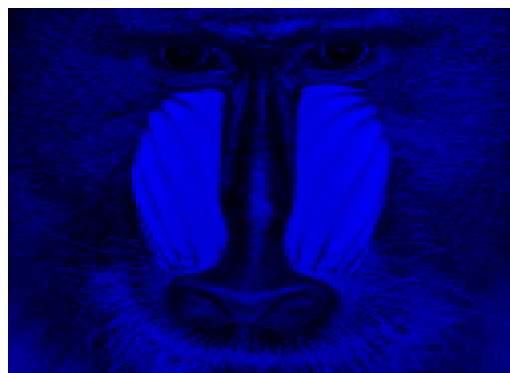
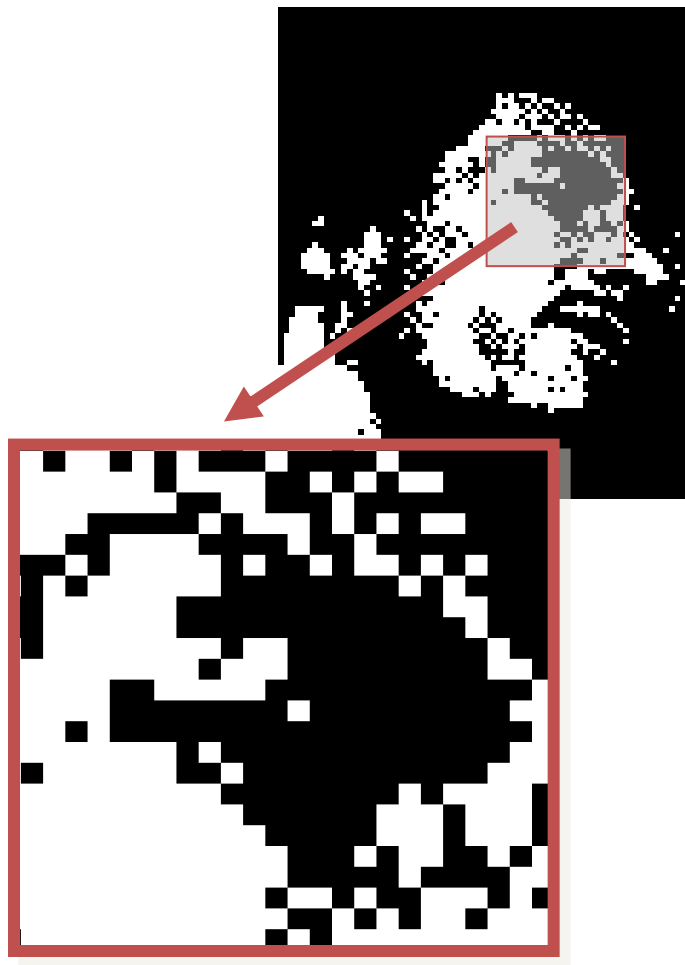


# *Embedded C*

## *C Programming Part 5*



# *Structure*



## *Structure(Cont.)*

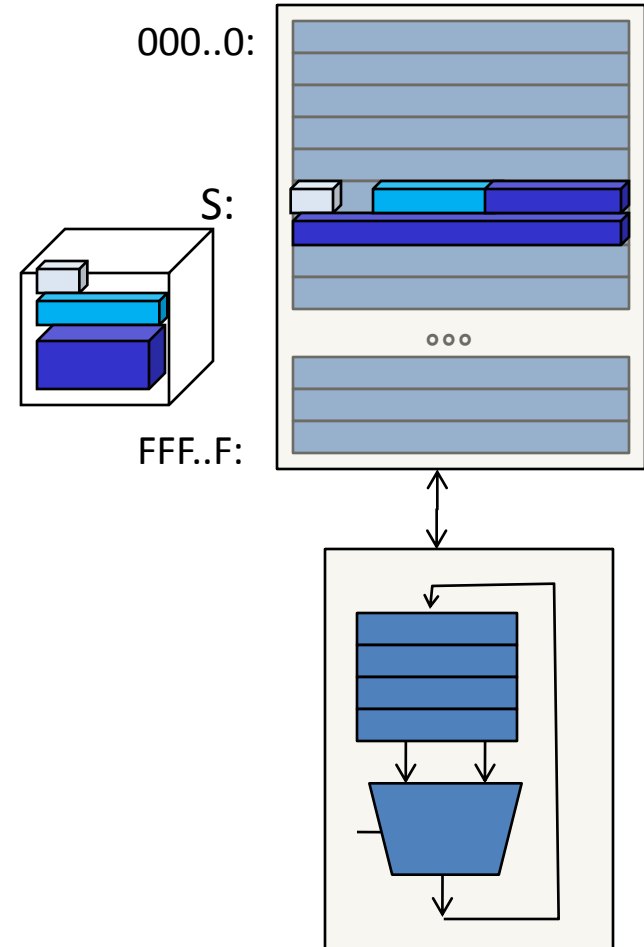
- A **struct** is a data structure composed from simpler data types

```
struct point
{
    int x;
    int y;
};
void PrintPoint(struct point p)
{
    printf("(%d,%d)", p.x, p.y);
}
int main(void)
{
    struct point p1 = {0,10};
    PrintPoint(p1);
}
```

## Structure(Cont.)

# Where do structures reside?

- **struct** are stored in memory
- The variable (i.e., name) is associated with the location (i.e., address) of the collection
- Elements are stored at fixed offsets
  - Can locate each of the elements
- Can operate on the named member object just like an object of that type



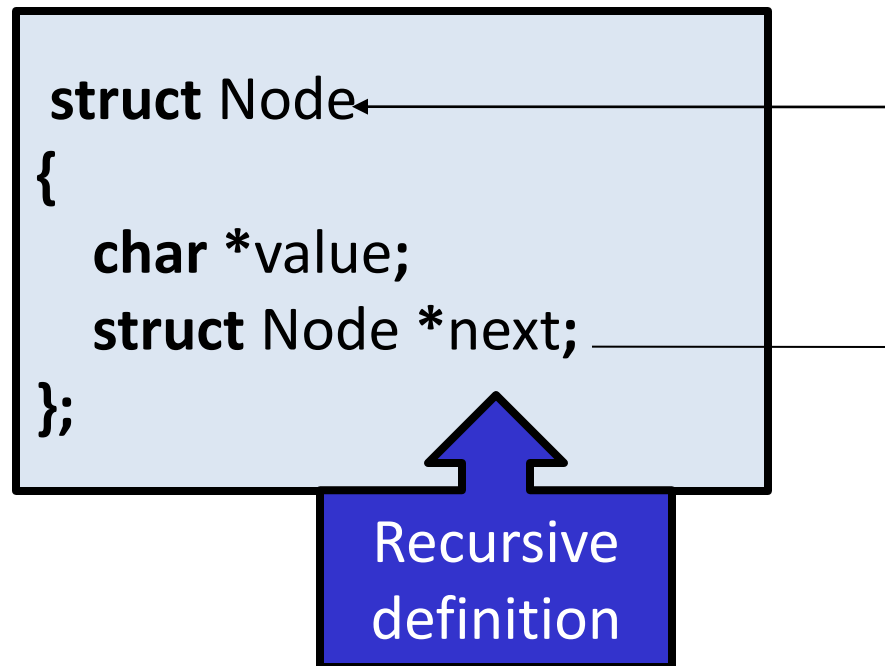
## Structure(Cont.)

- Comparing of structures are not allowed
- Usually, more efficient to pass a pointer to the **struct**
- The C arrow operator (->) dereferences and extracts a structure field with a single operator
- The following are equivalent:

```
struct point *p;  
/* code to assign to pointer */  
printf("x is %d\n", (*p).x);  
printf("x is %d\n", p->x);
```

## Structure(Cont.)

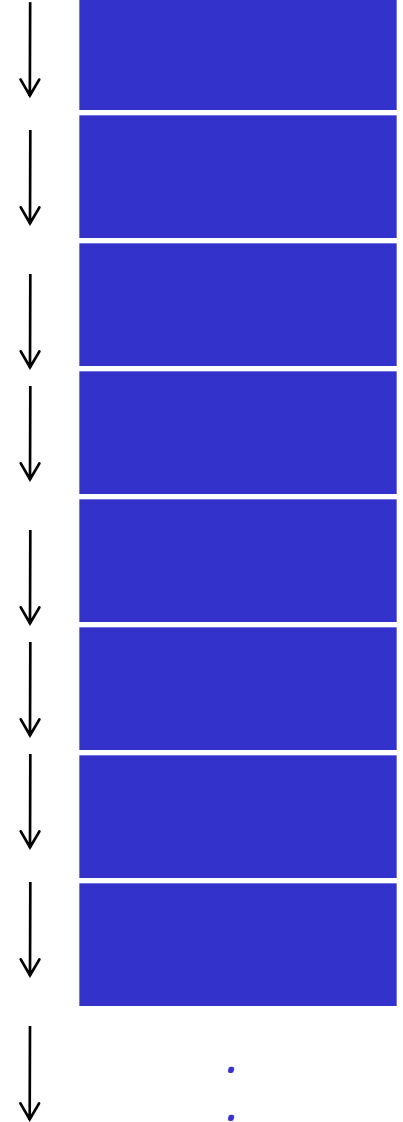
- A pointer to structure can be defined inside itself which is called **recursive definition**(Self-referential)
- note that we can not define structure inside itself, why?



# Structure(Cont.)

## Example

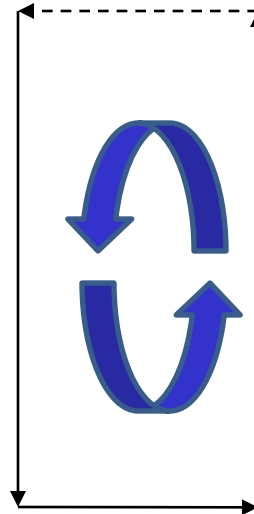
Memory



Memory  
overflow

```
Struct processor
{
  Alu a;
  memory m;
  bus b;
  peripheral p;
  Register * r;
  Processor coprocessor;
  Int num_reg;
};
```

```
Void main(int)
{
  Processor my_processor;
}
```



## Structure(Cont.)

### *How big are structs?*

- Recall C operator sizeof() which gives size in bytes (of type or variable)

```
struct p  
{  
    char x;  
    int y;  
};
```

- How big is sizeof(p)? 5 bytes? 8 bytes?
- Compiler may word align integer y



## Structure(Cont.)

### Bit fields

- Bit field is member of a structure whose size (in bits) has been specified
- Bit field enables better memory utilization but what about performance
- Bit field does not enable us to access individual bits
- To define bit fields, Follow **unsigned** or **int** member with a colon (:) and an integer constant representing the width of the field

**struct Example**

```
{  
    unsigned a : 13;  
    unsigned b : 3;  
    unsigned c : 4;  
};
```

**struct Example2**

```
{  
    unsigned a : 13;  
    unsigned b : 3;  
    unsigned   : 4;  
};
```

- Unnamed bit field used as padding in the structure and nothing may be stored in these bits
- Unnamed bit field with zero width aligns next bit field to a new storage unit boundary
- Attempting to take the address of a bit field may not be applicable

# Unions

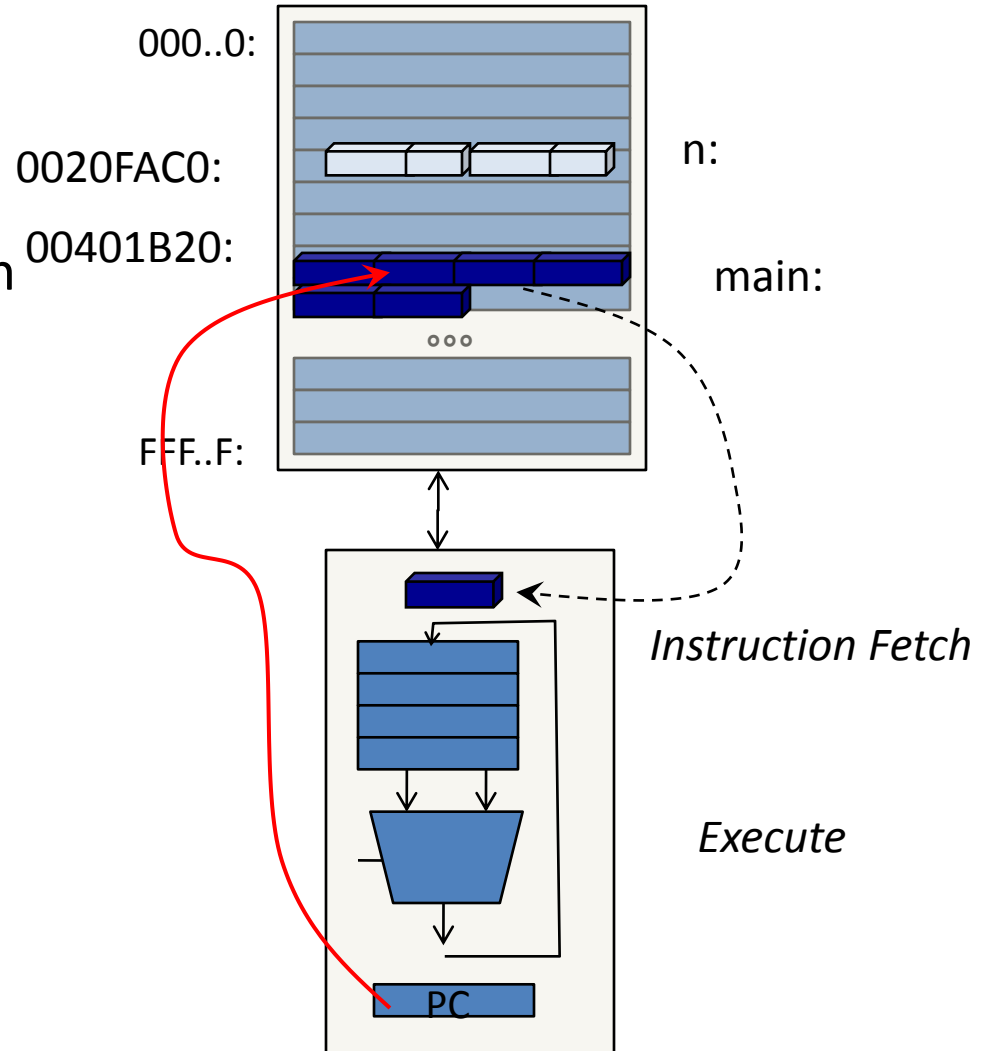
- Memory that contains a variety of objects over time
- Conserves storage but what about performance

```
union Number  
{  
    int x;  
    float y;  
};  
union Number value;
```

- The amount of storage required to store a union is implementation dependent but will always be at least as large as the largest member of the union
- Valid union operations
  - Assignment to union of same type: =
  - Taking address: &
  - Accessing union members: .
  - Accessing members using pointers: ->
  - Comparing unions is not allowed

# Where does the program itself reside?

- In memory, just like the data
- Processor contains a special register – PC
  - Program counter
  - Address of the instruction to execute (i.e. ptr)
- Instruction Execution Cycle
  - Instruction fetch
  - Decode
  - Operand fetch
  - Execute
  - Result Store
  - Update PC



# *Pointers to functions*

- Function name is starting address of code that defines function
- Similar to how array name is address of first element
- What is a pointer to a function?
  - A pointer just like any other
  - Data pointed at by the pointer is actually machine code for the function pointed at.
- How is it declared?

```
int (*f)(int start,int stop);
```

- This declares a variable f which is a pointer to a function that returns an int and takes two ints as parameters
- Now, just like any other pointer, the declaration does no allocation
- So, in this case, f points at nothing and any attempt to dereference it will have very spectacular side effects!
- You cannot dynamically allocate memory for function pointers

## *Pointers to functions(Cont.)*

- You can only set pointer-to-function variables equal to pointers to existing functions.
- How do you do that?

```
int SimpleAdd(int arg1,int arg2)  
{  
    return arg1 + arg2;  
}  
int main()  
{  
    int (*f)(int start,int stop);  
    f = SimpleAdd;  
    return 0;  
}
```

## *Pointers to functions(Cont.)*

- We can call it!
- How do you call a function when you have a pointer to it

```
int SimpleAdd(int arg1,int arg2)
{
    return arg1 + arg2;
}
int main()
{
    int (*f)(int start,int stop);
    f = SimpleAdd;
    int x = (*f)(3,4);
    int y = f(3,4);
    printf( "x is %d ",x);
    return 0;
}
```

## *Pointers to functions(Cont.)*

- OK, interesting concept. But what use is it?
- Most frequently used to allow a programmer to pass a function to another function
- Suppose I am writing a function which contains variables which need to be acted on
- Suppose that I want to be able to have multiple ways to act on those variables
- A function pointer as a parameter is a good solution

# *Pointers to functions(Cont.)*

## *Examples*

`char *argv;`

Argv : pointer to char

`Int (*daytab)[13];`

daytab: pointer to array[13] of int

`int *daytab[13];`

daytab: array[13] of pointer to int

`void *comp();`

comp: function returning pointer to void

`void (*comp)();`

comp: pointer to function returning void



## *Pointers to functions(Cont.)*

### *Examples(Cont.)*

```
char (*(*x())[ ] )();
```

x: function returning pointer to .....

```
char (*a[ ] )();
```

a: array[ ] of pointer to .....

```
char b();
```

b: function returning char

x: function returning pointer to array[ ] of pointer to function returning char

## *Pointers to functions(Cont.)*

### *Examples(Cont.)*

```
char (*(*x[3])())[5];
```

x: array[3] of pointer to .....

```
char (*a())[5];
```

a: function returning pointer to .....

```
char b[5];
```

b: array[5] of char

x: array[3] of pointer to function returning pointer to  
array[5] of char

# ***Global variable***

- So far we have talked about several different ways to allocate memory for data:
  1. Declaration of a local variable
  2. “Dynamic” allocation at runtime by calling allocation function (alloc).
- One more possibility exists...
  3. Data declared outside of any procedure (i.e., before main).
    - Similar to #1 above, but has “global” scope.

```
int myGlobal;  
main()  
{  
}
```

*Thanks*

*Embedded C*