Warning: These notes are not complete, it is a Skelton that will be modified/add-to in the class. If you want to us them for studying, either attend the class or get the completed notes from someone who did

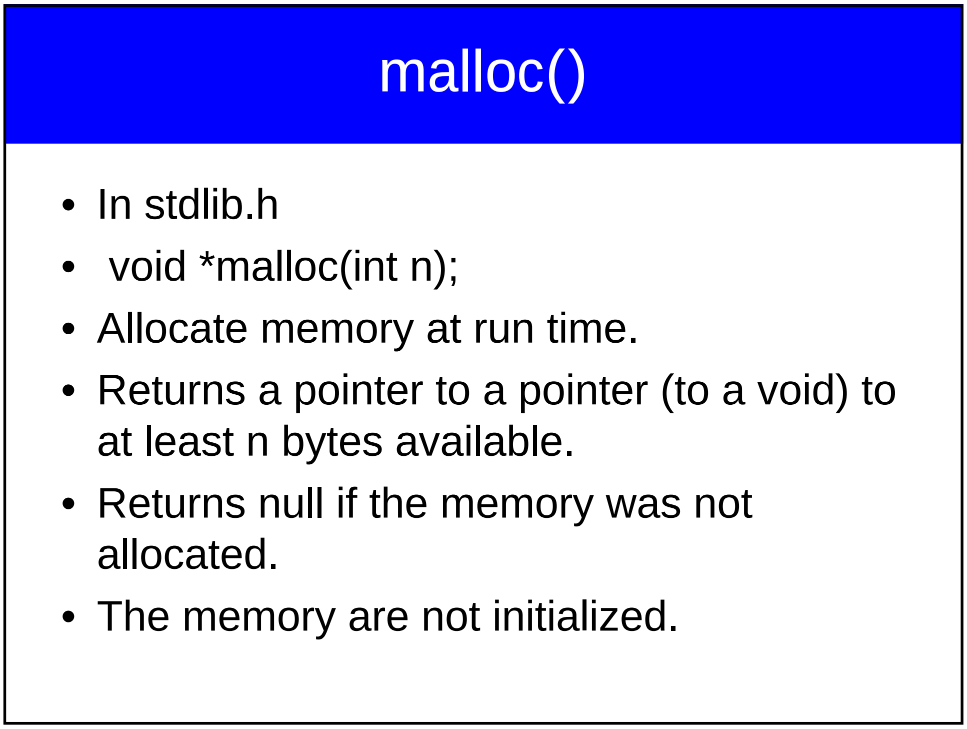
CSE2301

Dynamic Memory Allocation and Structs

These slides are based on slides by Prof. Wolfgang Sturzlinger at York University

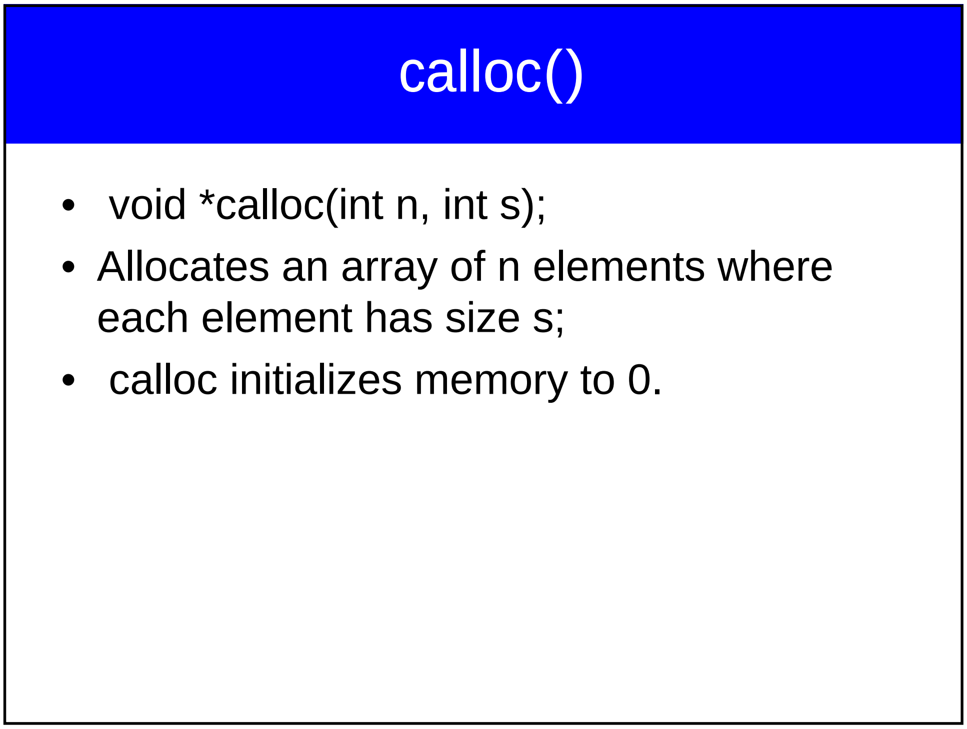
Dynamic memory Allocation

- How to allocate memory during run time.
- int x=10;
- int myarray[x]; That is not allowed in C



malloc()

- In stdlib.h
- void *malloc(int n);
- Allocate memory at run time.
- Returns a pointer to a pointer (to a void) to at least n bytes available.
- Returns null if the memory was not allocated.
- The memory are not initialized.



calloc()

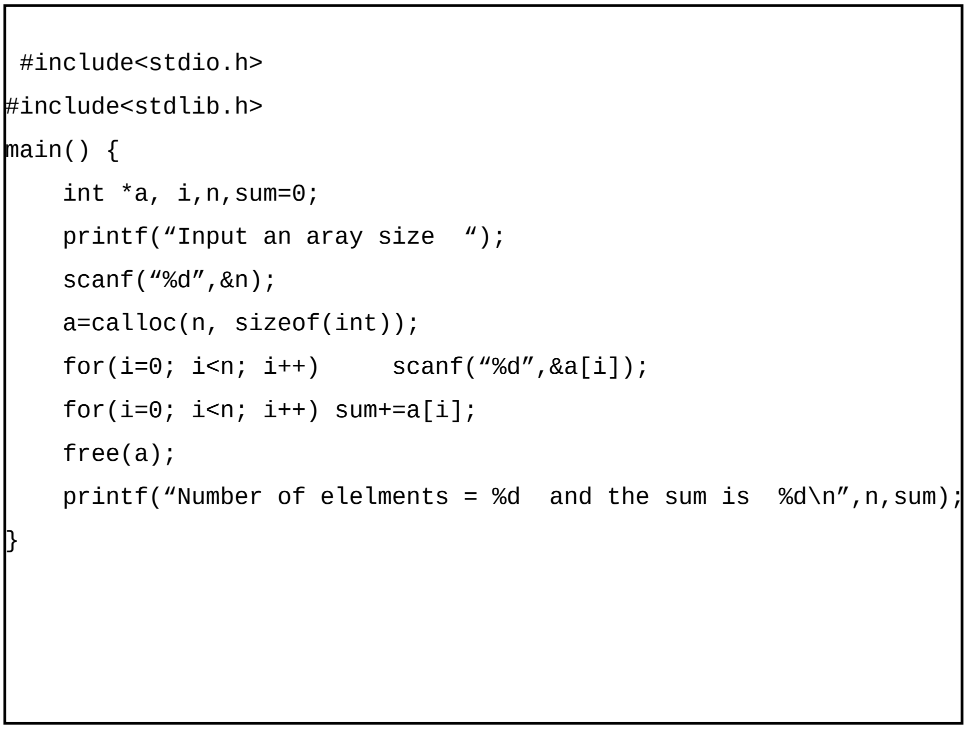
- void *calloc(int n, int s);
- Allocates an array of n elements where each element has size s;
- calloc initializes memory to 0.

realloc()

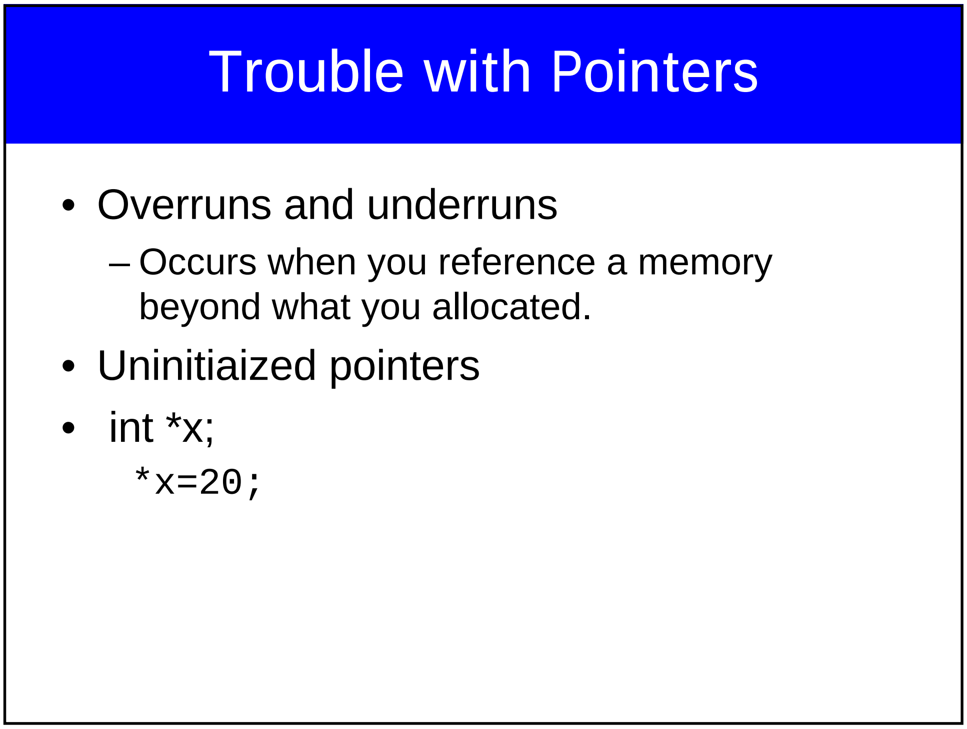
- What if we want our array to grow (or shrink)?
- void * realloc(void *ptr, int n);
- Resizes a previously allocated block of memory.
- ptr must have been returned from either calloc, malloc, or realloc.
- Array me be moved it it could not be extended in its current location.

free()

- void free(void *ptr)
- Releases the memory we previously allocated.
- ptr must have been returned by malloc, alloc, or realloc.



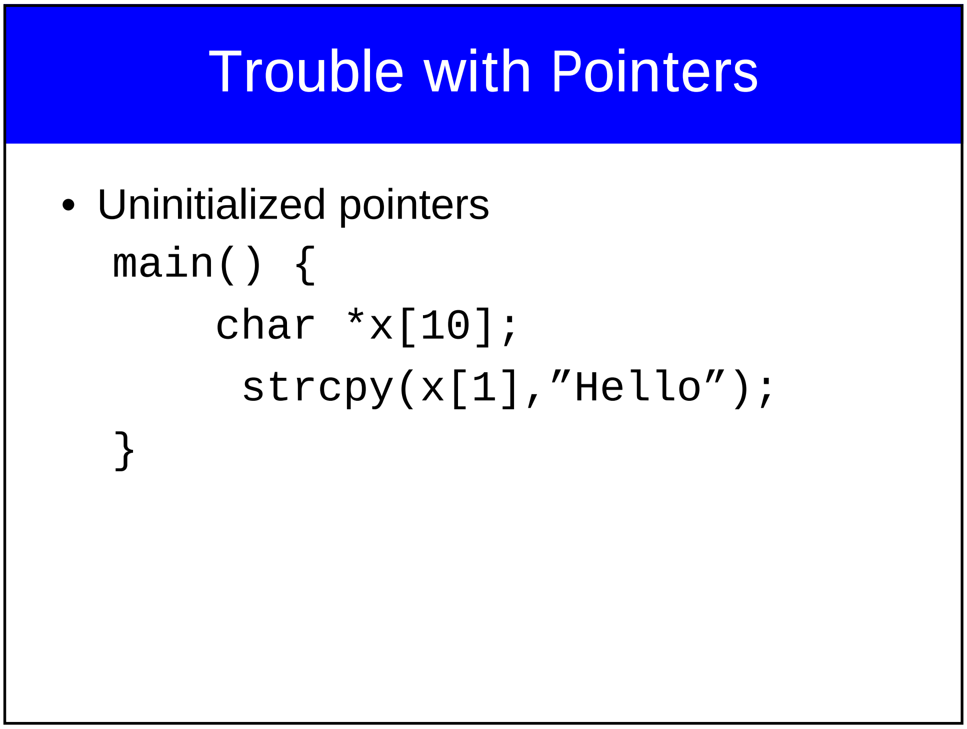
```
 #include<stdio.h>
#include<stdlib.h>
main() {
    int *a, i,n,sum=0;
    printf(“Input an aray size  “);
    scanf(“%d”,&n);
    a=calloc(n, sizeof(int));
    for(i=0; i<n; i++)     scanf(“%d”,&a[i]);
    for(i=0; i<n; i++) sum+=a[i];
    free(a);
    printf(“Number of elelments = %d  and the sum is  %d\n”,n,sum);
}
```



Trouble with Pointers

- Overruns and underruns
  - Occurs when you reference a memory beyond what you allocated.
- Uninitiaized pointers
- int *x;

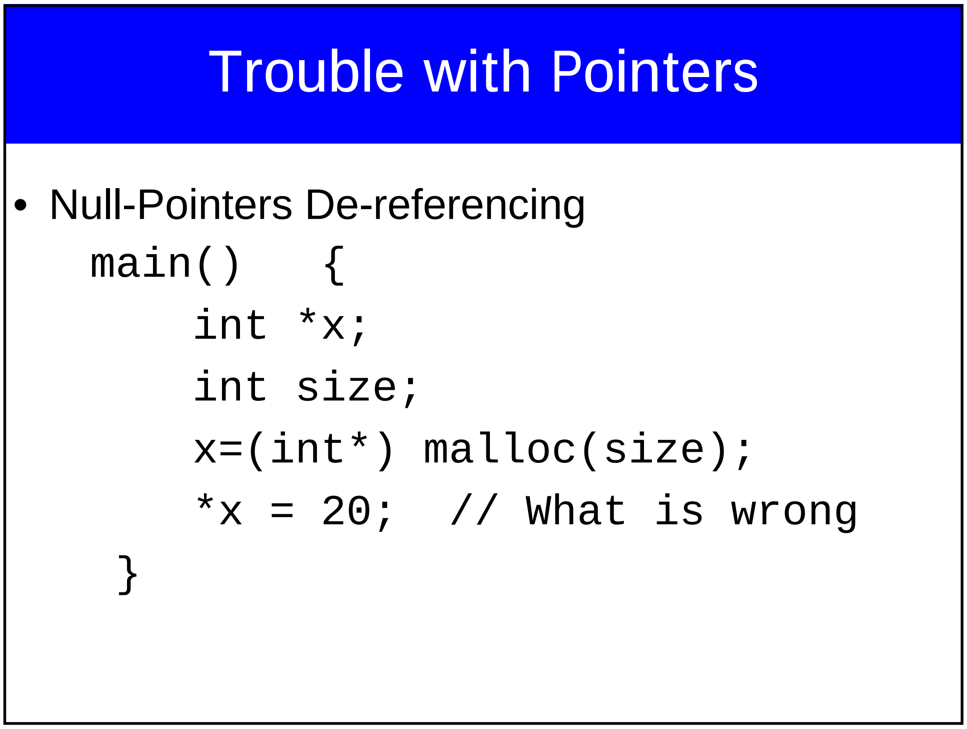
```
  *x=20;
```



Trouble with Pointers

- Uninitialized pointers

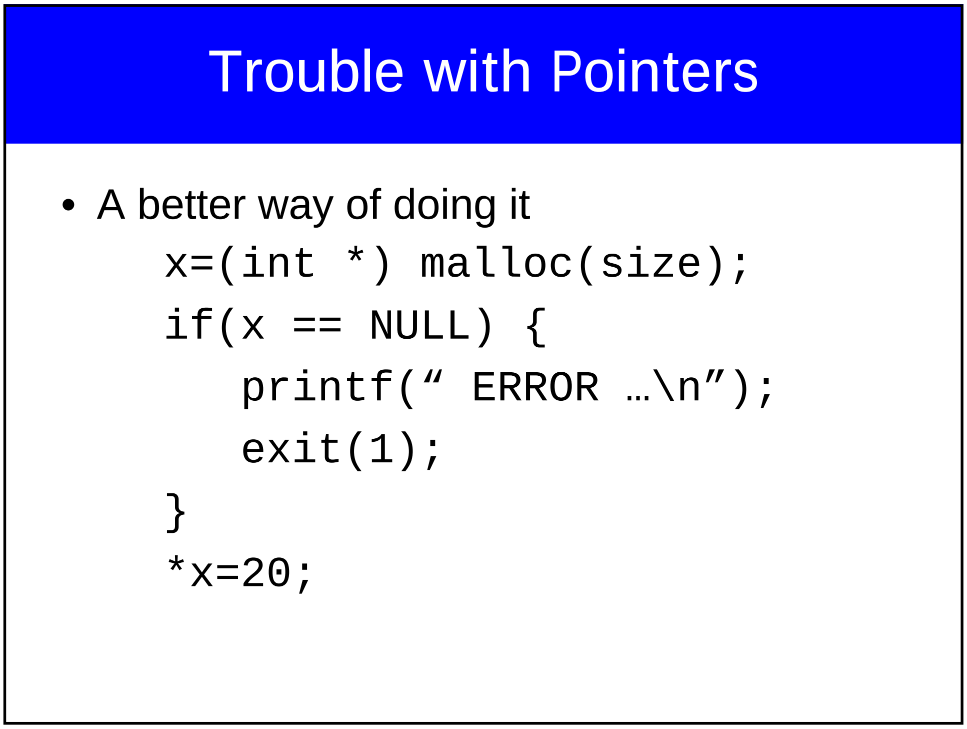
```
main() {
    char *x[10];
     strcpy(x[1],”Hello”);
}
```



Trouble with Pointers

- Null-Pointers De-referencing

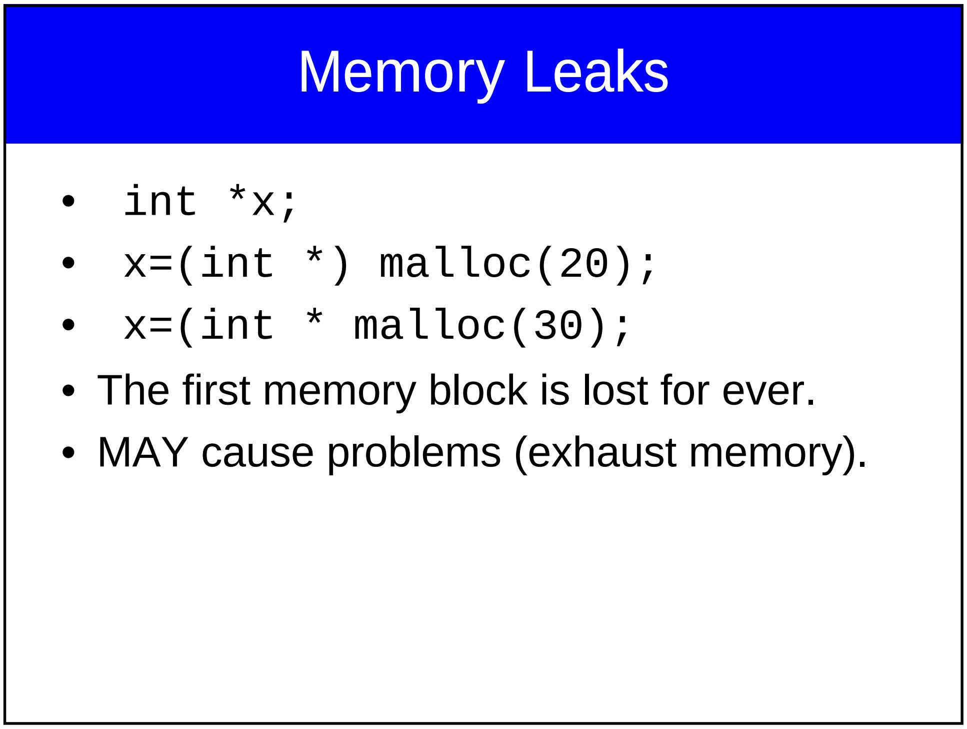
```
main()   {
    int *x;
    int size;
    x=(int*) malloc(size);
    *x = 20;  // What is wrong
 }
```



Trouble with Pointers

- A better way of doing it

```
x=(int *) malloc(size);
if(x == NULL) {
   printf(“ ERROR …\n”);
   exit(1);
}
*x=20;
```



Memory Leaks

- `int *x;`
- `x=(int *) malloc(20);`
- `x=(int * malloc(30);`
- The first memory block is lost for ever.
- MAY cause problems (exhaust memory).

Trouble with Pointers

- Inappropriately use freed memory
- char *x;
- x=(char *) malloc(50);
- free(x);
- x[0]=‘A’;

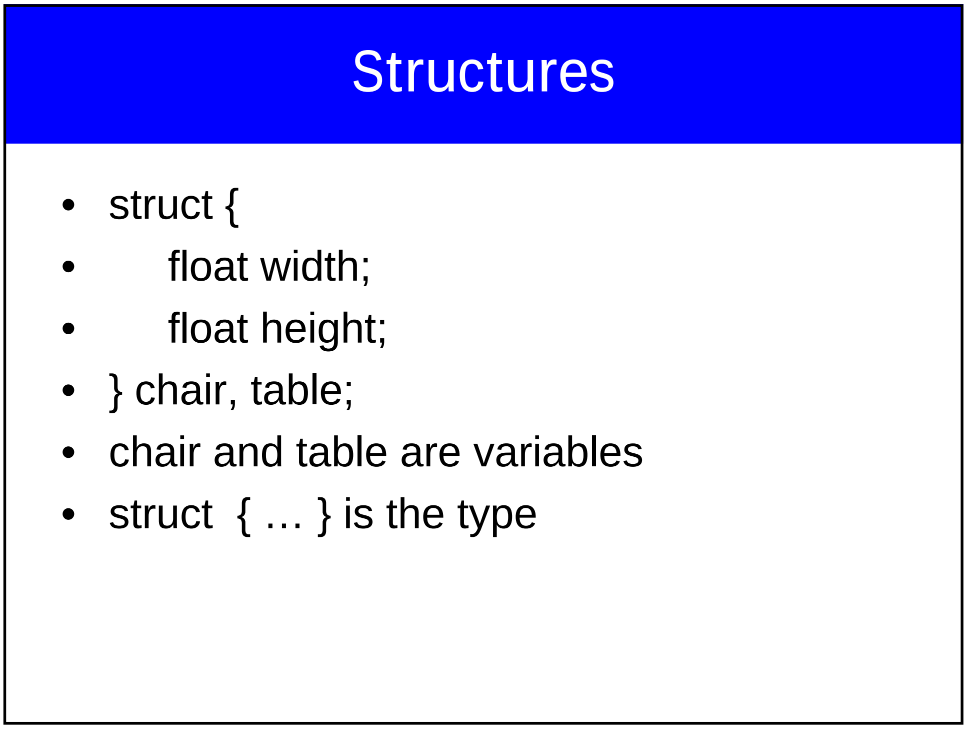
Does work on my system

Trouble with Pointers

- Inappropriately freed memory
- `char *x=NULL;`
- `free(x);`

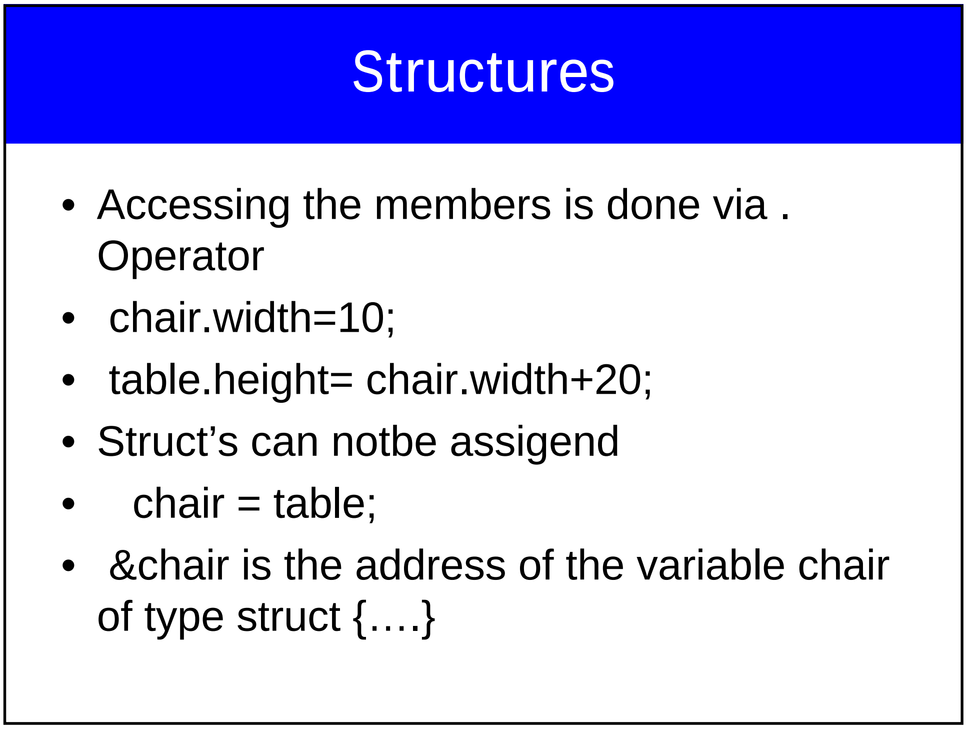
- `x=malloc(50);`
- `free(x+1);`

- `free(x)`
- `free(x)`



Structures

- struct {
- float width;
- float height;
- } chair, table;
- chair and table are variables
- struct { … } is the type



Structures

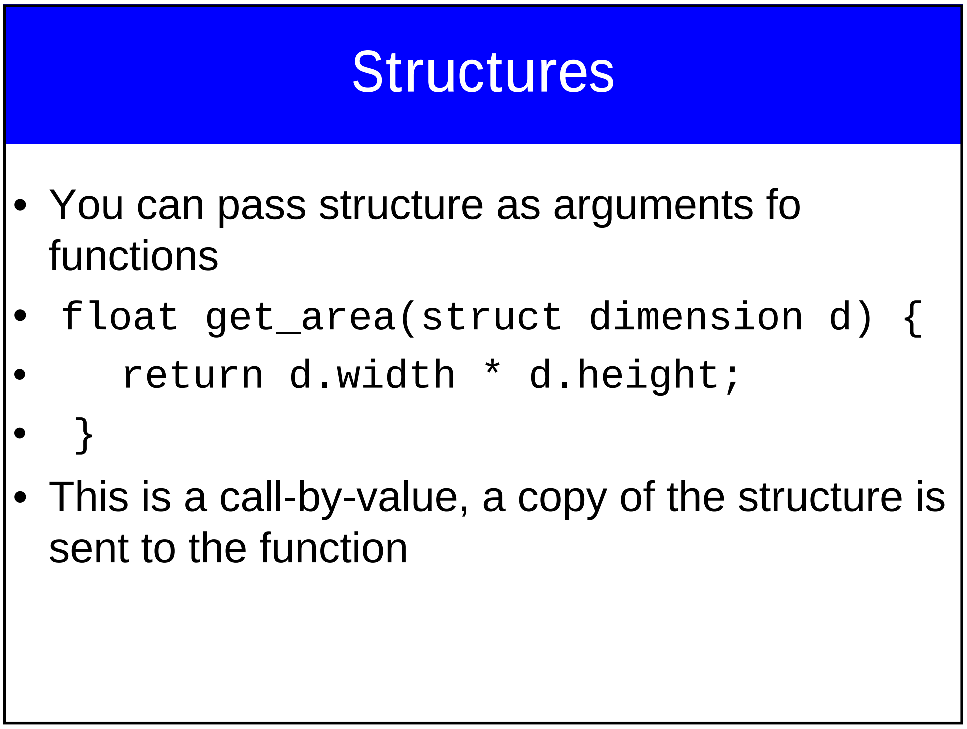
- Accessing the members is done via . Operator
- chair.width=10;
- table.height= chair.width+20;
- Struct’s can notbe assigend
- chair = table;
- &chair is the address of the variable chair of type struct {….}

Structures

- struct dimensions {
-    float width;
-    int height;
- };
- Now, `struct dimension` is a valid type
- `struct dimension a,chair,table;`

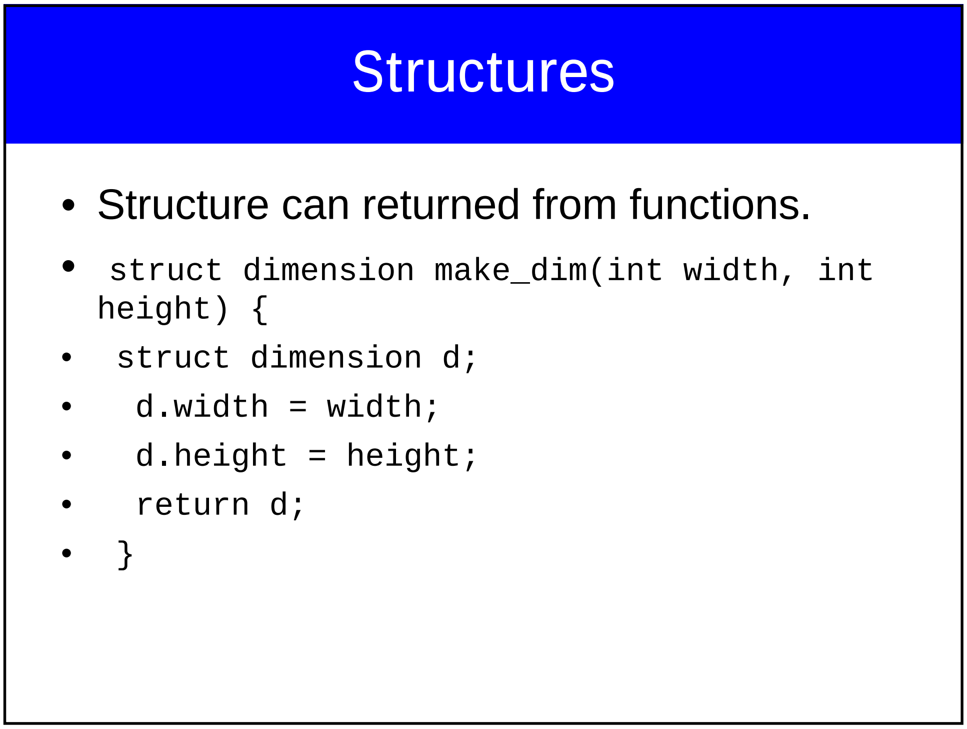
Structures

- Struct names have their own namespace separate from variables and functions;
- Truct member names have their own namespace.
- `struct dimension dimension;` √
- `struct dimesnsion {`
- `      float width;` √
- `      float height;`
- `} height;`



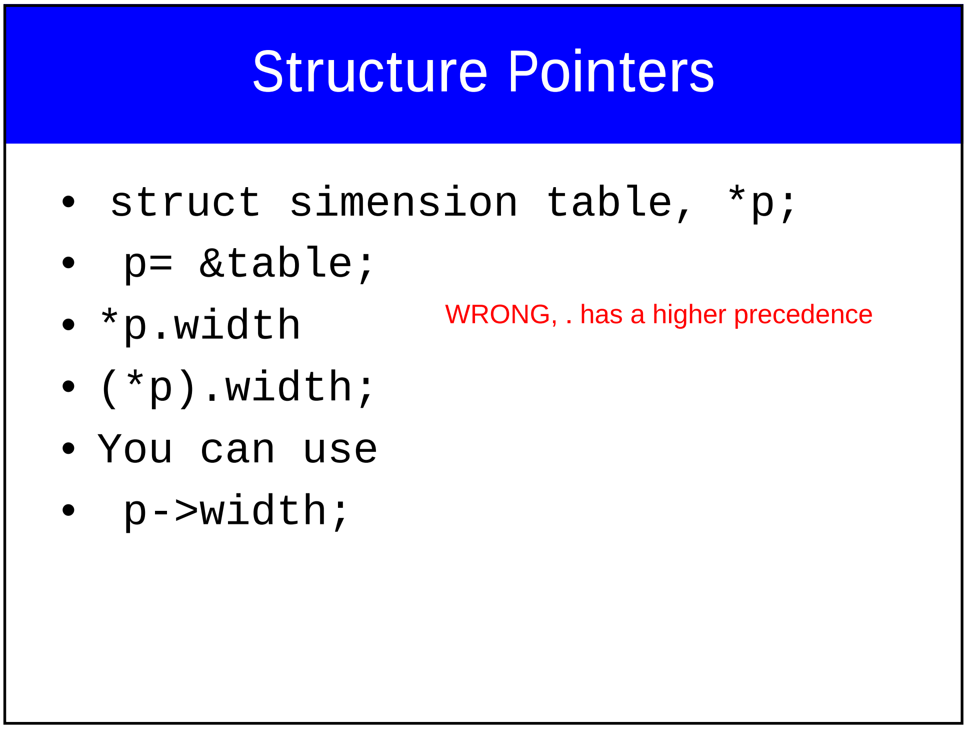
Structures

- You can pass structure as arguments fo functions
- `float get_area(struct dimension d) {`
- `   return d.width * d.height;`
- ` }`
- This is a call-by-value, a copy of the structure is sent to the function



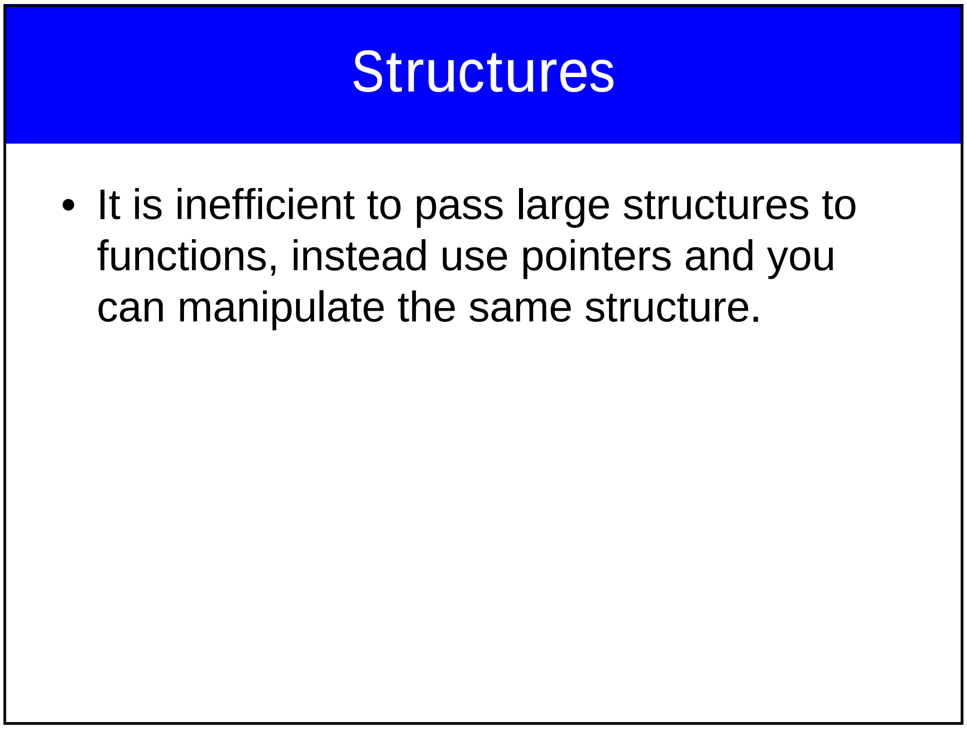
Structures

- Structure can returned from functions.
- `struct dimension make_dim(int width, int height) {`
- `struct dimension d;`
- `  d.width = width;`
- `  d.height = height;`
- `  return d;`
- `}`



Structure Pointers

- struct simension table, *p;
- p= &table;
- *p.width WRONG, . has a higher precedence
- (*p).width;
- You can use
- p->width;



Structures

- It is inefficient to pass large structures to functions, instead use pointers and you can manipulate the same structure.

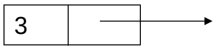
Example

- #include <stdio.h>
- main() {
- struct {
-             int len;
-             int height;
- } tmp, *p=&tmp;
- tmp.len=10;
- tmp.height=20;
- printf(" 111   %d \n",++(p->len));
- printf(" 222   %d \n",++p->len);
- }

```
111   11
222   12
```

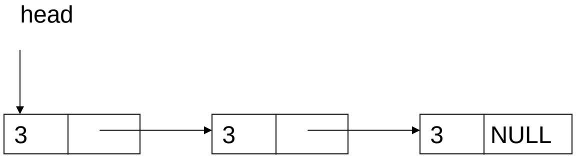
Linked List

- struct list {
-    int data;
-    struct list *next;
- };
- It is O.K. to use a pointer to a struct that is declared but not defined



Linked List

- Poniter head points to the first element
- Last element pointer is NULL



Linked List

```
#include <stdio.h>
#include <stdlib.h>
main() {
     struct list{
             int len;
             struct list *next;
     } *head,*p,*last;
     head=(struct list *)malloc(sizeof(list));
     head->len=1;
     head->next=NULL;
     last=head;
     int i;
     scanf("%d",&i);
```

```
while(i>=0) {
    scanf("%d",&i);
    p = (struct list *)malloc(sizeof(list));
    p->len=i;
    p->next=NULL;
    last->next=p;
    last=p;
}
printf("Enter the number you want to
search for   ");
scanf("%d",&i);
for(p=head; p!=NULL; p=p->next)
  if(p->len == i)
        printf("FOUND \n");

}
```

Delete a node

- void deleteit(dim **p, int i) {
- // DOES NOT WORK 1st element
- dim **p1,*temp;
- p1=p;
-     while((*p1)->num != i) p1=&(*p1)->next;
-     temp = *p1;
-     *p1 = (*p1)->next;
- free(temp);
- }

p is a pointer to head (pointer to 1st element in list

Debug it

Array of Structures

- struct dimension {
-     float width;
-     float height;
- };
- struct dimension chairs[2];
- struct dimension *tables;
- tables = (struct dimension*) malloc (20*sizeoff(struct dimension));

Initializing Structures

- struct dimension sofa={2.0, 3.0};

- struct dimension chairs[] = {
- {1.4, 2.0},
- {0.3, 1.0},
- {2.3, 2.0} };

Nested Structures

- struct point {int x, int y;};
- struct line {
- struct point a;
- struct point b;
- } myline;
- myline.a.x=0;
- myline.a.y=5;

Structs

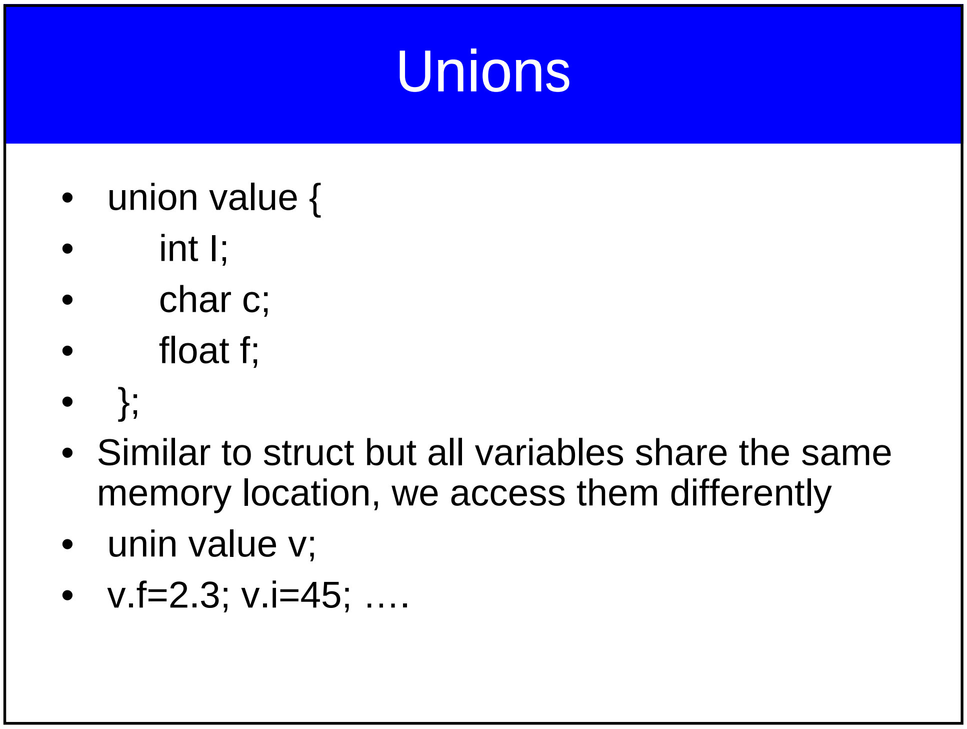
- struct {float w,h;} chair;
- struct dim {float w,h;} chair1;
- struct dim {float w,h;};
-     struct dim chair2;
- typedef struct {float w,h;} dim;
- dim x,y;

typedef

- We can define a new type and use it later

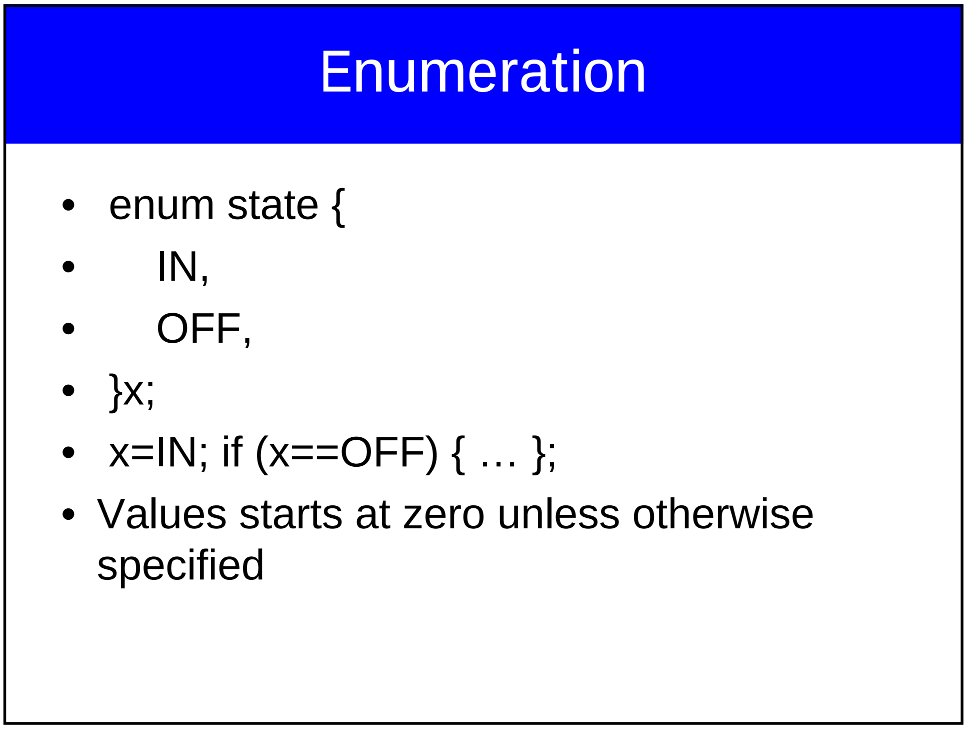
```
typedef struct {
     int x,y;
     float z;
  } newtype;
newtype a1,b1,c1,x;
```

- Now, newtype is a type in C just like int and float



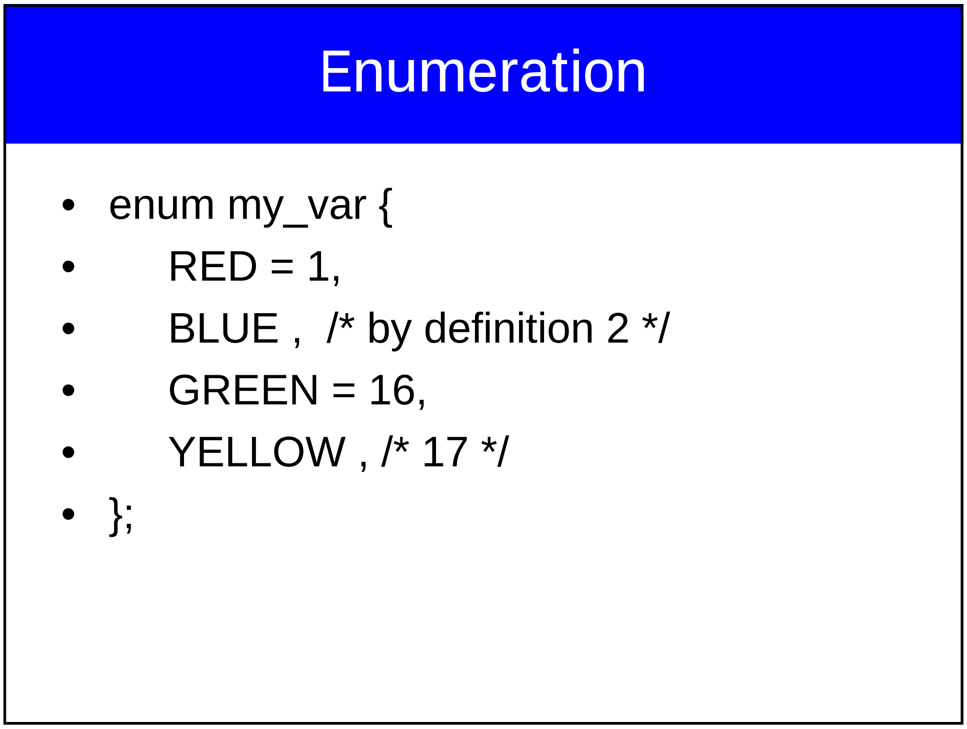
Unions

- union value {
- int I;
- char c;
- float f;
- };
- Similar to struct but all variables share the same memory location, we access them differently
- unin value v;
- v.f=2.3; v.i=45; ….



Enumeration

- enum state {
- IN,
- OFF,
- }x;
- x=IN; if (x==OFF) { … };
- Values starts at zero unless otherwise specified



Enumeration

- enum my_var {
- RED = 1,
- BLUE , /* by definition 2 */
- GREEN = 16,
- YELLOW , /* 17 */
- };