Sunbeam Institute of Information Technology
Pune and Karad

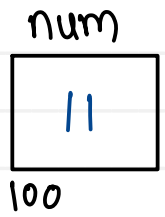
Module - Embedded C Programming

Trainer - Devendra Dhande

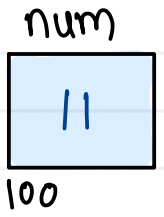
Email – devendra.dhande@sunbeaminfo.com

const qualifier

```
int num = 11;
```




```
const int num = 11;
   or
int const num = 11;
```



num is constant

```
num = 2.
num++
num+=2.
```

} error

```
int *ptr = &num;
```

ptr is not constant

```
ptr = 2.
ptr++
*ptr = 2.
```

```
const int num = 11;
   or
int const num = 11;
```

num is constant

```
num = 2.
num++
num+=2.
```

} error

const qualifier

```
const int *ptr = &num;    int num = 11;
```

Wrt
ptr is not constant
num is constant

```
ptr = 2.
ptr++
*ptr = 2.   <- error
```

num is not constant

```
num = 2.
num++
num+=2.
```

```
int *const ptr = &num;    int num = 11;
```

Wrt
ptr is constant
num is not constant

```
ptr = 2.   } <- error
ptr++
*ptr = 2.
```

num is not constant

```
num = 2.
num++
num+=2.
```

```
const int *const ptr = &num;     int num = 11;
```

wrt
ptr is constant
num is not constant

```
ptr = 2.
ptr++
*ptr = 2.
```
← error

num is not constant

```
num = 2.
num++
num += 2.
```

void pointer

- void pointer is called as generic pointer because it stores address of any type of variable

- while assigning address to the void pointer typecasting is not required

```
int num = 10;
void *ptr = &num;
```

- while dereferencing void pointer, type casting is required, because void pointer don't have any scale factor.

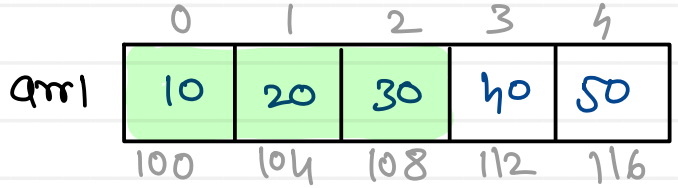
```
*(int *)ptr;
```

```
#define NULL (void *)0
```

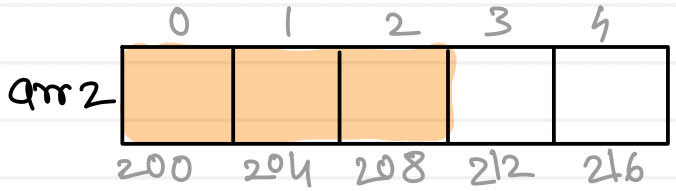
- NULL is used to initialize any type of pointer.

memcpy() and memmove()

```
int arr1[5] = {10, 20, 30, 40, 50};
```

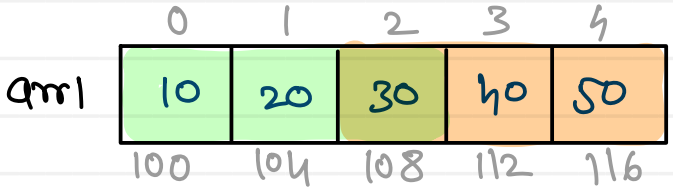



```
int arr2[5]
```




```
memcpy(arr2, arr1, 12);
```

```
int arr1[5] = {10, 20, 30, 40, 50};
```



overlapping address

```
memcpy(arr1+2, arr1, 12)
```

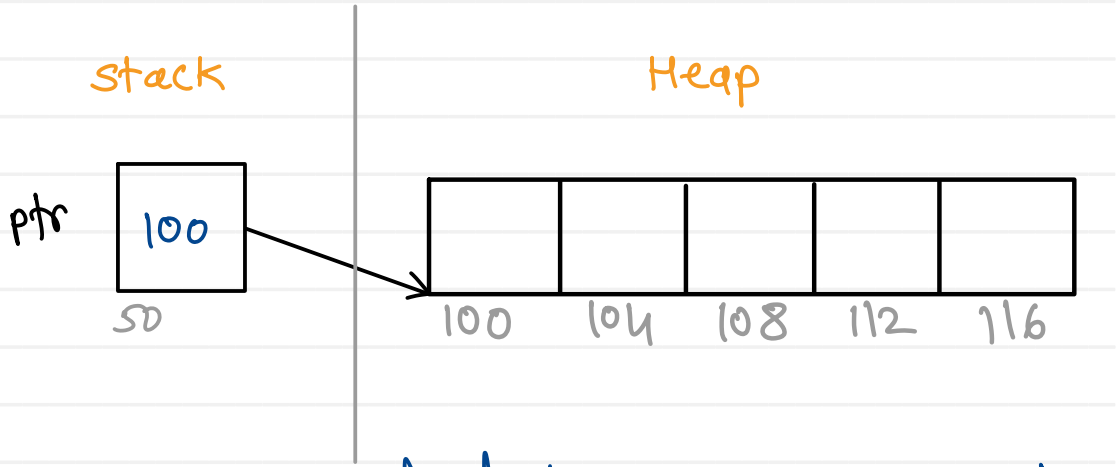
108 100

↓↓

```
memmove(arr1+2, arr1, 12)
```

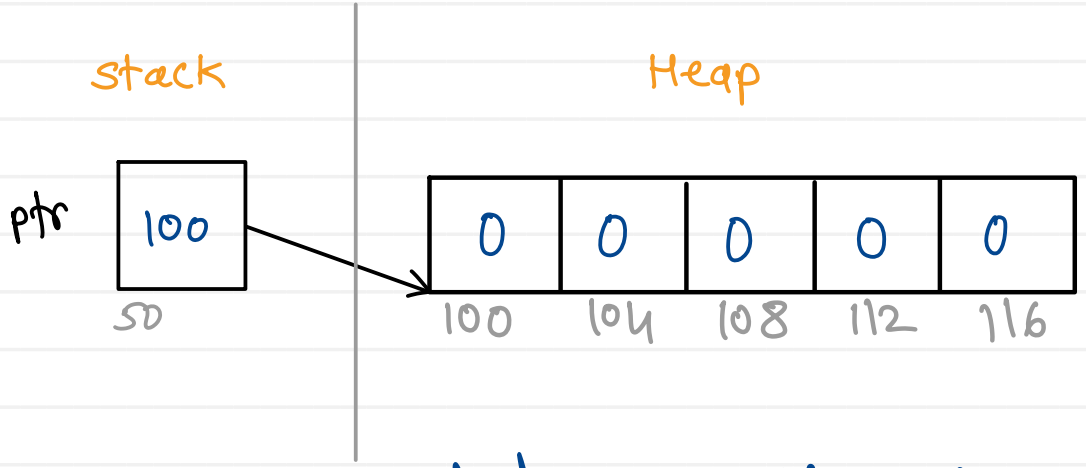
malloc Vs calloc

```
int *ptr = (int *)malloc(length * sizeof(int));
```



- space allocated by malloc contains garbage

```
int *ptr = (int *)calloc(length, sizeof(int));
```



- space allocated by calloc is initialized to 0.
- integer overflow of multiplication of nmemb & size is checked & error is returned

realloc()

newsize < oldsize (shrink)
- data from starting address to newsize will be unchanged.

newsize > oldsize (grow)
- data from starting address to oldsize will be unchanged.
- added memory will not be initialized

1. if immediate free space is available
   - only space is extended to newsize
   - same starting address will be returned
2. if immediate free space is not available
   - new space of newsize will be allocated
   - old content will be copied into new space
   - old memory space will be released.
   - address of new space will be returned

realloc (ptr , size)

realloc (NULL, size);
↳ will work like malloc

realloc (ptr , 0);
↳ will work like free

Memory leakage and Dangling pointer

```
int main( void )
{
    int num = 10;
    ptr = malloc(20);
    =
    =
    ptr = &num;
    =
    =
    return 0;
}
```

Memory leakage

- When we loose address of allocated space, that space we can not access/ use in our program.
- also we can not free/release that memory space.

```
int main(void)
{
    ptr = malloc(20);
    =
    =
    free(ptr);
    =
    =
    return 0;
}
```

ptr will become dangling

Dangling pointer :

- pointer which stores invalid address (garbage value / address of deallocated space/ address of local variables returned from function)

Memory leakage:

```
int main(void){
    num = 10;
    ptr = malloc(20);
    =
    =
    ptr = &num;
    =
→   free(ptr);
    return 0;
}
```

ptr is holding address of allocate space by malloc

ptr is holding address of num

Dangling pointer:

- pointer which stores invalid address

```
int main(){
    int *ptr;
    =
    =
    return 0;
}
```

ptr is dangling

```
int main(){
    int *ptr = malloc(20);
    =
    free(ptr);
    =
    =
    return 0;
}
```

ptr is dangling

```
int *fun(void){
    int num = 10;
    return &num;
}
int main(){
    int *ptr = fun();
    =
    =
    return 0;
}
```

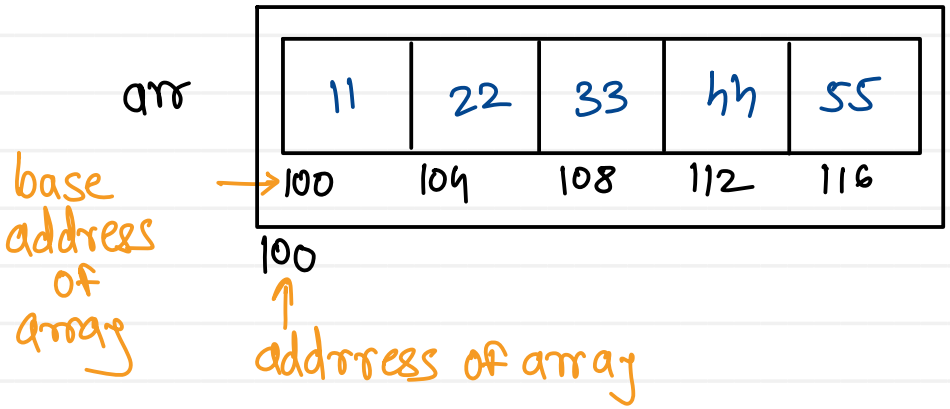
ptr is dangling

```
int main(){
    int *ptr = 100;
    =
    =
    return 0;
}
```

ptr is dangling

Array pointer

```
int arr[5] = {11, 22, 33, 44, 55};
```



arr = 100 - base address

```
int *ptr1 = arr;
```

scale factor of ptr1 = 4 bytes

```
ptr1 = 100
*ptr1 = 11
```

&arr = 100 - address of array

```
int (*ptr2)[5] = &arr;
```

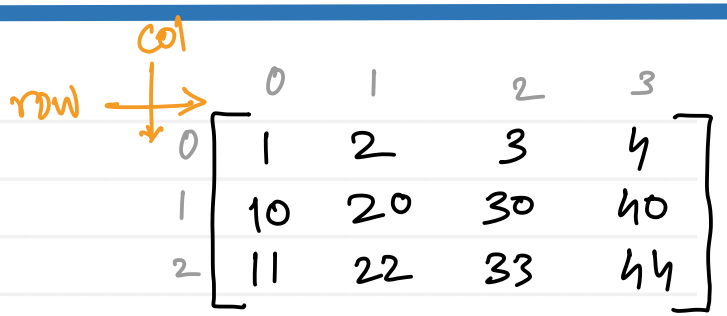
Scale factor of ptr2 = 20 bytes (size of array)

```
ptr2 = 100
*ptr2 = 100   <- base address of array
```

2D array

Declaration :

<data type> <name>[rows][cols];



e.g. int arr[3][4];

int arr[3][4] = {1, 2, 3, 4, 10, 20, 30, 40, 11, 22 33, 44};

int arr[3][4] = {1, 2, 3, 4, 10, 20, 30, 40, 11, 22};

int arr[ ][4] = {1, 2, 3, 4, 10, 20, 30, 40, 11, 22 33, 44};

arr[row][col]

arr[1][1] = 20

arr[2][3] = 44

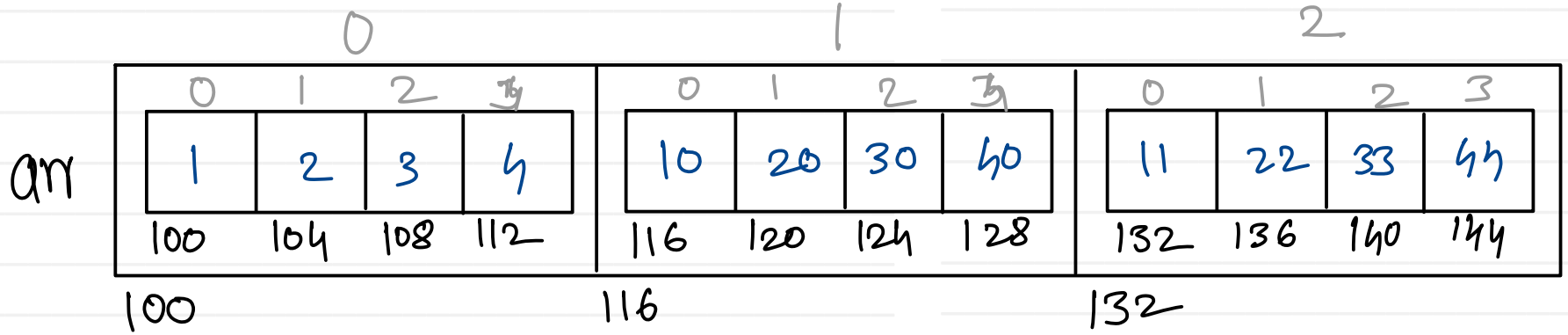
```
int arr[ ][4] = {{1, 2, 3, 4},
                 {10, 20, 30, 40},
                 {11, 22 33, 44}};
```

```
int arr[ ][4] = {{1, 2, 3, 4},
                 {10, 20, 30},
                 {11, 22}};
```

1	2	3	4
10	20	30	0
11	22	0	0

Pointer to 2D array

- 2D array is collection of 1D arrays

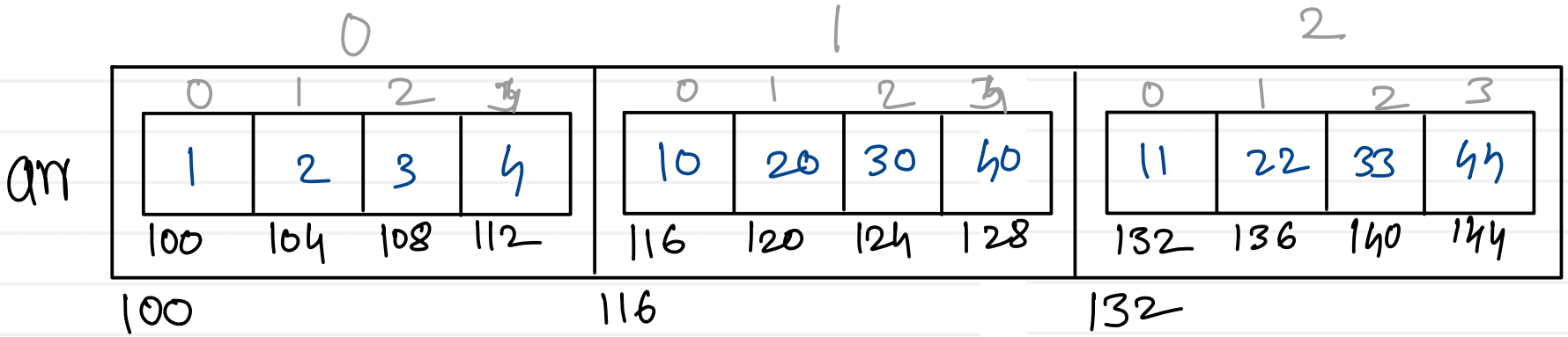


arr = 100 – base address of 2D array
(address of first 1D array)

int (*ptr)[4] = arr;

ptr = 100 ptr+1 = 116 ptr+2 = 132
*ptr = 100 *(ptr+1) = 116 *(ptr+2) = 132

2D array pointer notation




```
int (*ptr)[4] = arr;
```

int ptr[][4]

ptr = 100 ptr+1 = 116 ptr+2 = 132

*(ptr+0) = 100 *(ptr+1) = 116 *(ptr+2) = 132

**(ptr+0) = 1

((ptr+0)+1) = 2

((ptr+0)+2) = 3

((ptr+0)+3) = 4

((ptr+2)+0) = 11

((ptr+2)+1) = 22

((ptr+2)+2) = 33

((ptr+2)+3) = 44

```
arr[i][j] = *(*(arr+i)+j)
```

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com