Advanced Pointers in C

Master Class at Sunbeam Infotech

Introduction – CPTR-01

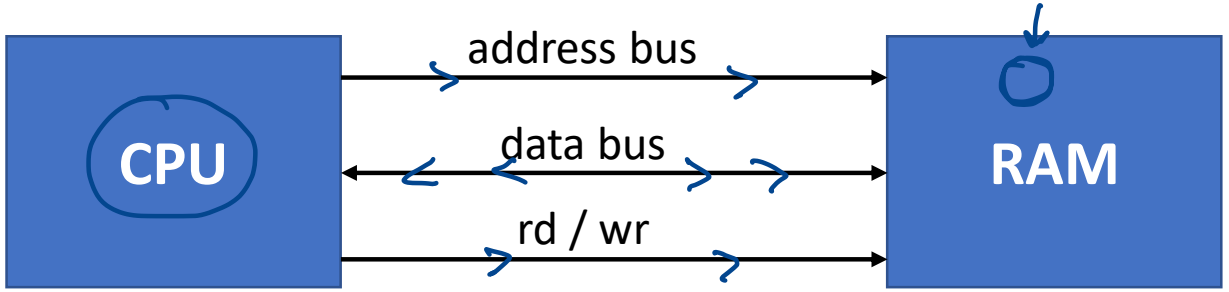
- Trainer: Mr. Nilesh Ghule.
  - 20+ years experience in C
  - Used in C in various domains/platforms: Embedded, IoT, Windows, Linux, Parallel programming, GPU programming.
- Why this webinar?
  - C pointer is essential skill for C programmers, specially in embedded & system domain.
  - Take your pointer skills to the next level.
- What to learn in this webinar?
  - Pointer fundamentals and common misconceptions.
  - Pointers to arrays, multi-dimensional arrays, structures and functions.
  - Complex pointer declarations, typedef and pointer applications.
- What is not covered in this webinar?
  - Not to learn C programming and pointers from scratch.
- Schedule: 9:00 AM to 12:00 PM – Sat (27-Jun) & Sun (28-Jun)

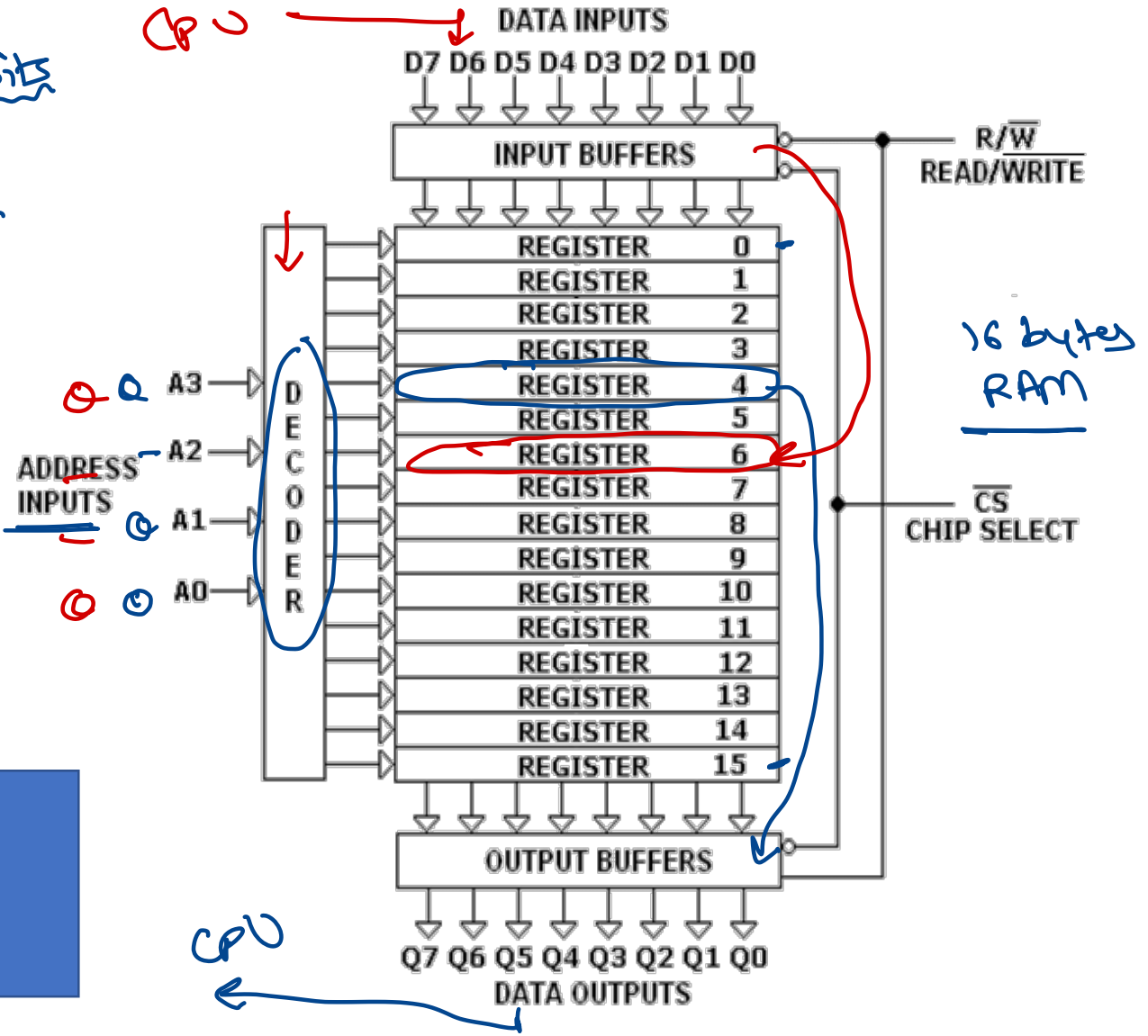
Agenda

- Memory address
- Pointer fundamentals
- Pointer arithmetic
- Call by value vs call by address
- Little endian vs Big endian
- Significance of pointer arithmetic
- Pointer to 1-D array
- Pointer to 2-D array
- Passing array to the function
- Returning array from function
- Void pointer

CPU and RAM interaction

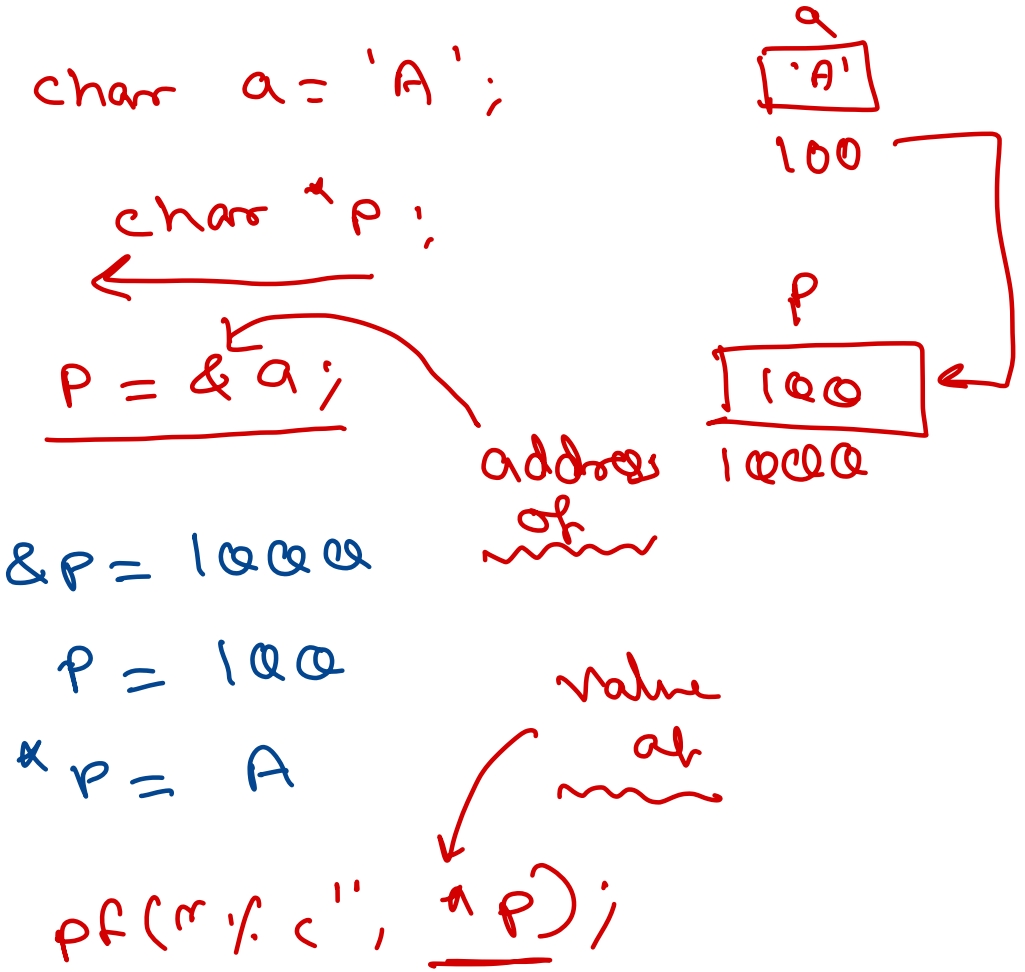
- RAM stores data into locations. Each location is made up of memory cells.
- Each location is identified by a number (0 to n-1), called as address.
- Memory location corresponding to given address is activated by decoder circuit inside the RAM.
- CPU send address to read/write over address bus and data is transferred over data bus.





Pointer fundamentals

- Pointer is a variable that stores address of some memory location.
- Internally it is unsigned integer (as it is RAM address).
- In C, pointer is special data type (and it is not compatible with unsigned int type).
- Also pointer is derived data type (based on primitive data type).
  - To store address of int, we have int pointer.
  - To store address of char, we have char pointer, …
- Size of pointer variable is always same, irrespective of its data type (as it stores only the address).

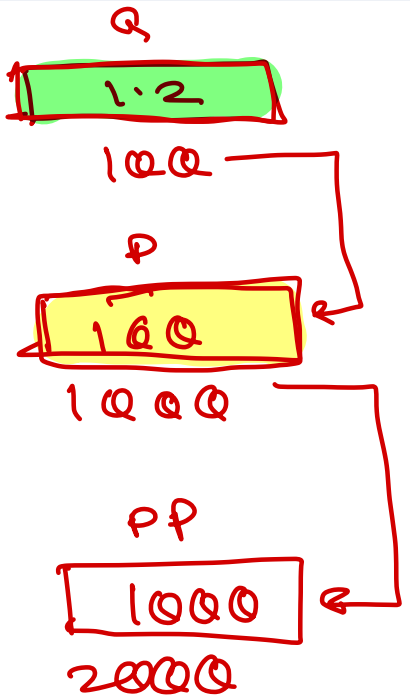


Pointer fundamentals

- Pointer syntax:
  - Declaration:
    - double *p;
  - Initialization:
    - p = &d;
  - Dereferencing:
    - printf("%lf\n", *p);
- Reference operator - &
  - Also called as direction operator.
  - Read as “address of”.
- Dereference operator - *
  - Also called as indirection operator.
  - Read as “value at”.

level of indirection

```
int main() {
  double a = 1.2;
  double *p = &a;
  double **pp = &p;
  printf("%lf\n", a);
  printf("%lf\n", *p);
  printf("%lf\n", **pp);
  return 0;
}
```



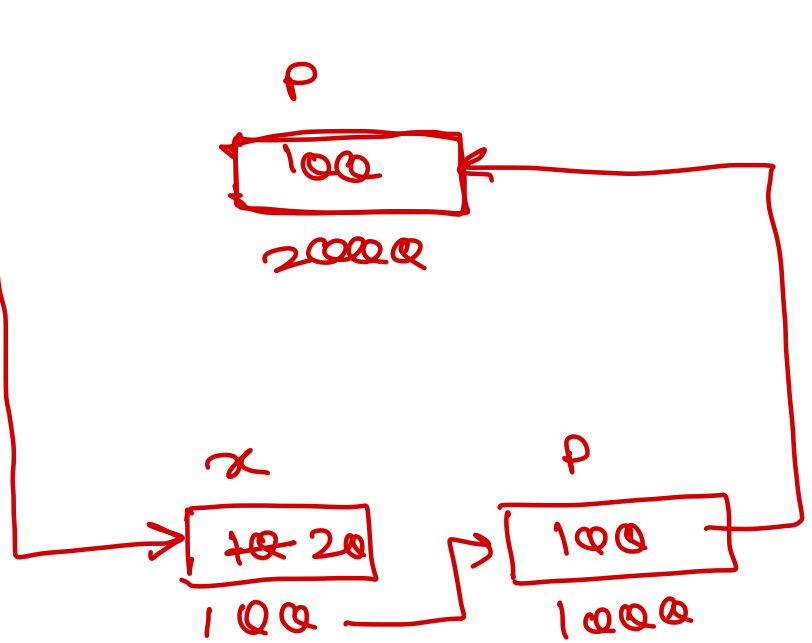
pp = 10000

*pp = 1600

**pp = 1.2

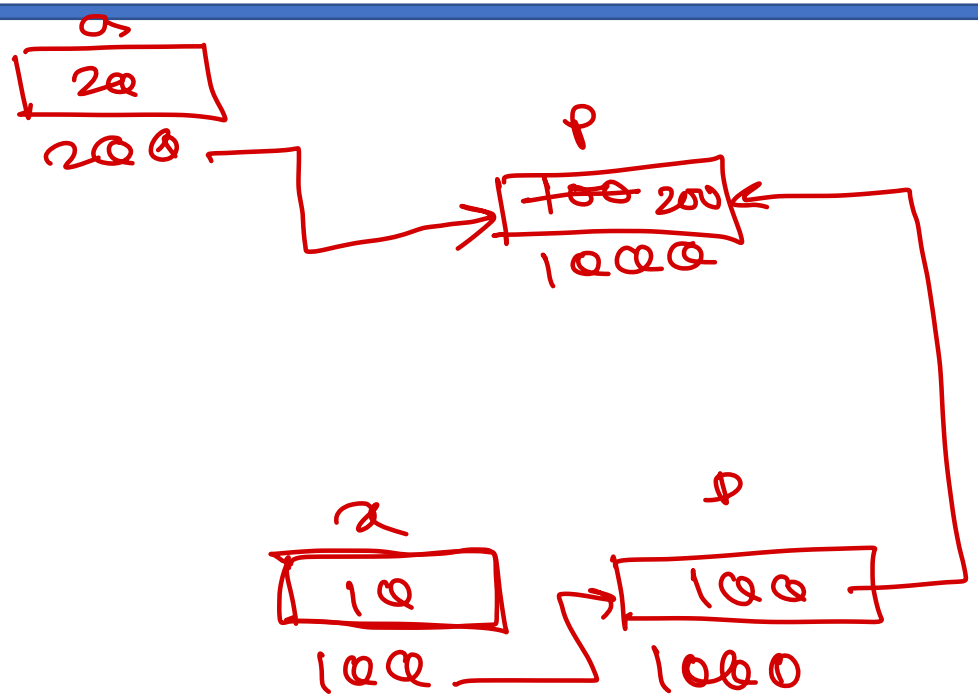
Call by ~~value~~ address for variable x

```
int a = 20;
void myfun(int *p) {
    *p = a;
}
int main() {
    int x = 10;
    int *p = &x;
    myfun(p);
    printf("x = %d\n", x);
    return 0;
}
```



Call by ~~address~~ value for "p"

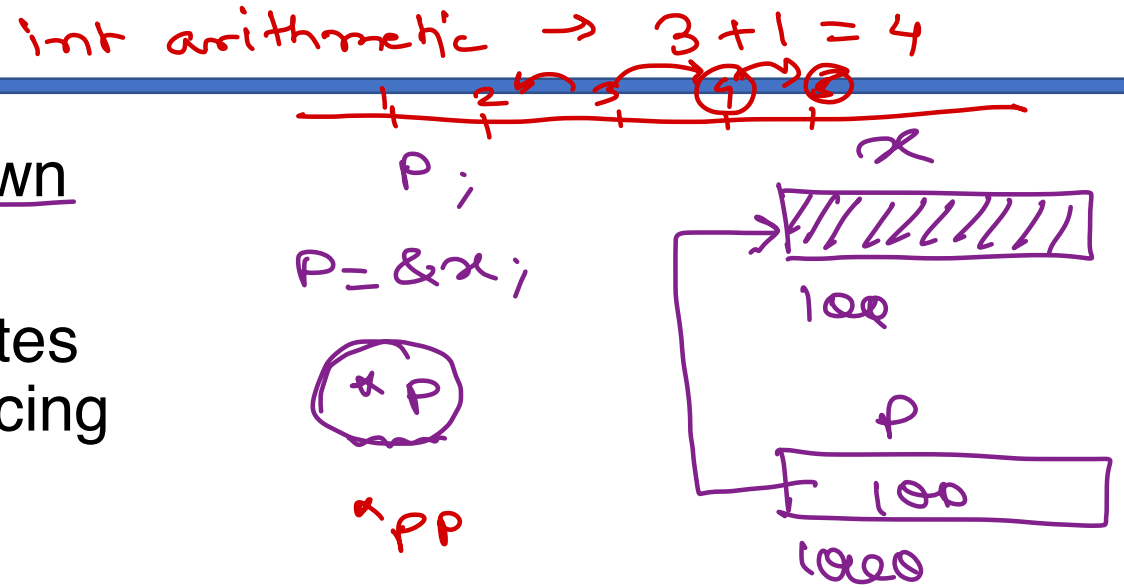
```
int a = 20;
void myfun(int *p) {
    *p = &a;
}
int main() {
    int x = 10;
    int *p = &x;
    myfun(p);
    printf("x = %d\n", x);
    return 0;
}
```



pointers are passed by value. i.e. the copy of address in pointer is created.

only array & function are passed by address.

Pointer arithmetic

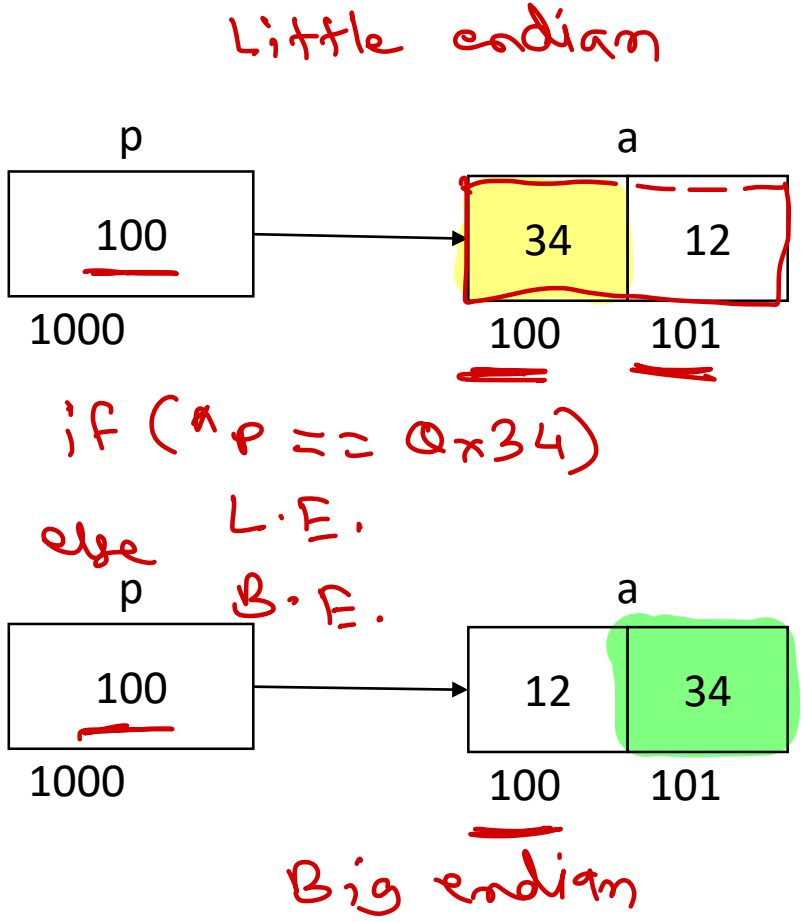


- Size of data type of pointer is known as Scale factor.
- Scale factor defines number of bytes to be read/written while dereferencing the pointer.
- Scale factor of different pointers
  - Pointer to primitive types: char*, short*, int*, long*, float*, double*
  - Pointer to pointer: char**, short**, int**, long**, float**, double**, void**
  - Pointer to struct/union
  - void pointer
  - Function pointer

- Scale factor plays significant role in pointer arithmetic.
- n locations ahead from current location
  - ptr + n = ptr + n * scale factor of ptr
- n locations behind from current location
  - ptr - n = ptr - n * scale factor of ptr
- number of locations in between
  - ptr1 – ptr2 = (ptr1 – ptr2) / scale factor of ptr1

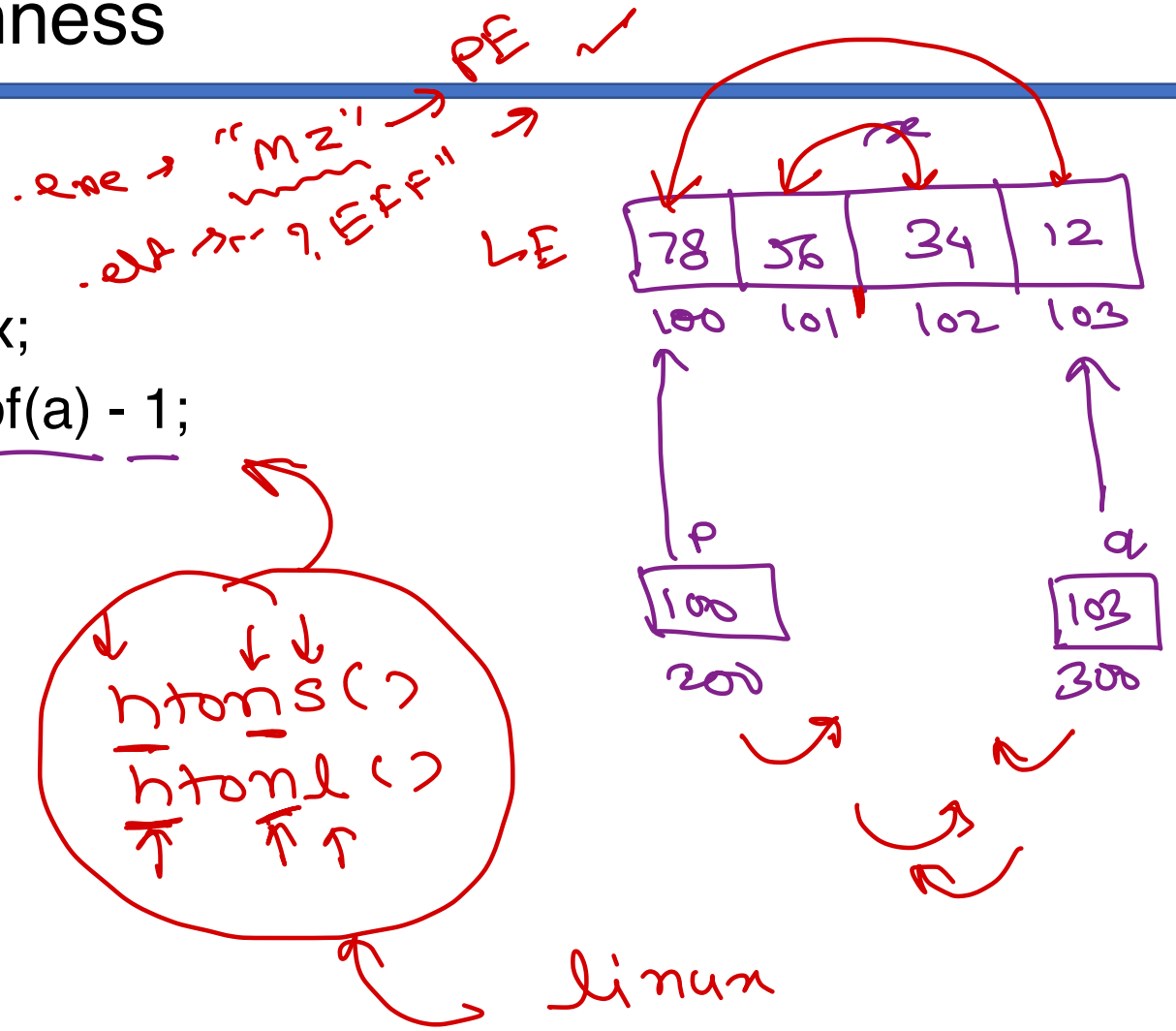
Little endian vs Big endian

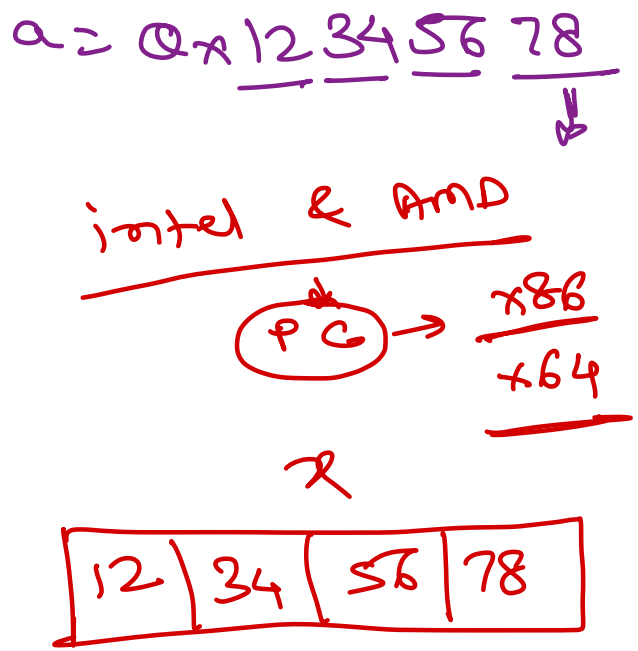
- CPU architecture can be little endian or big endian.
- Little endian: Lower byte is stored on lower address.
- Ex: x86, ARM CM3, …
- Big endian: Lower byte is stored on higher address. → also called as "network order"
- Ex: PowerPC, …
- short int a = 0x1234;
- char *p = (char*)&a;



Reverse endianness

```
int rev(int a) {
    int x = a;
    char *p = (char*)&x;
    char *q = p + sizeof(a) - 1;
    while(p < q) {
        char t = *p;
        *p = *q;
        *q = t;
        p++;
        q--;
    }
    return x;
}
```

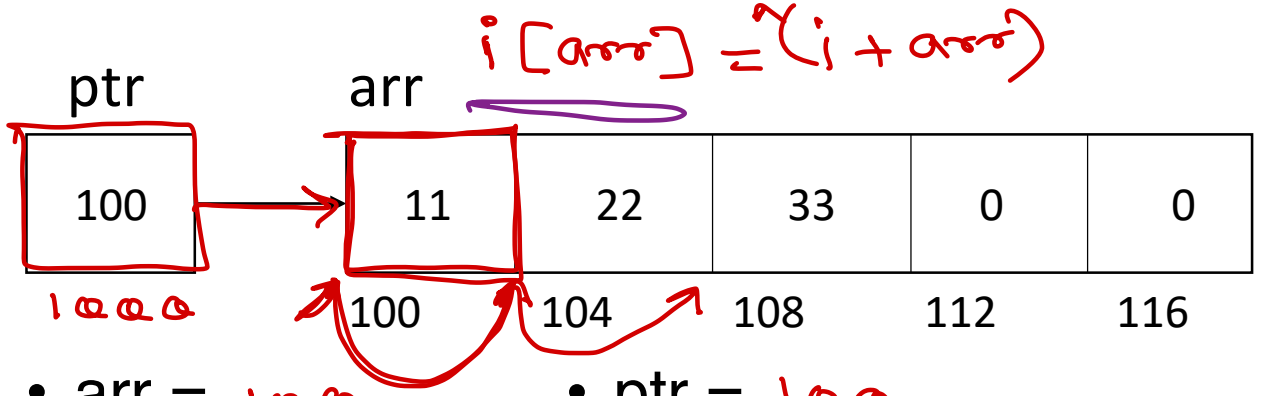




linux socket programming.

Pointer to 1-D array

- 1-D array
  - Collection of similar elements
  - Consecutive memory locations
- int arr[5] = {11, 22, 33};
- Pointer to an array is pointer to 0th element of the array.
- int *ptr = &arr[0];
- In fact, array name itself is treated as pointer to 0th element in any runtime expression.
- In C, subscript notation is converted into pointer notation to access elements.
- x[y] = *(x+y)



- arr = 1000
- arr + 1 = 1004
- arr + 2 = 1008
- *arr = 11
- *(arr + 1) = 22
- *(arr + 2) = 33
- arr[1] = *(arr+1)
- arr[2] = *(arr+2)

- ptr = 1000
- ptr + 1 = 1004
- ptr + 2 = 1008
- *ptr = 11
- *(ptr + 1) = 22
- *(ptr + 2) = 33
- ptr[1] = *(ptr+1)
- ptr[2] = *(ptr+2)

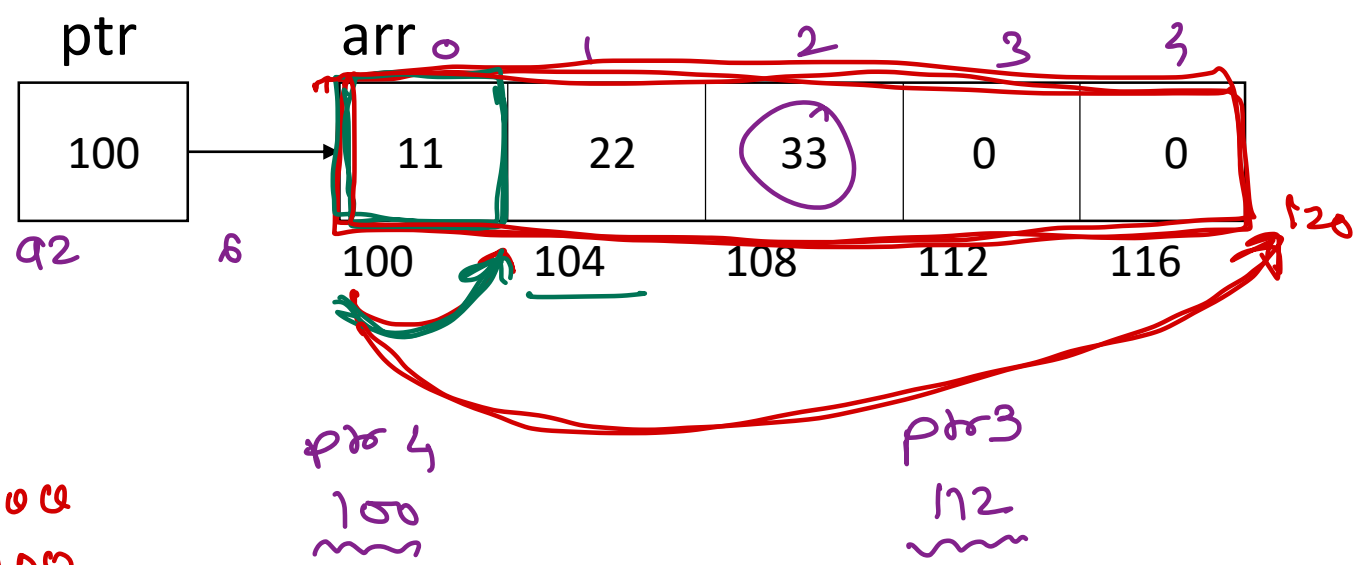
sizeof(ptr) = 4
sizeof(arr) = 20

Pointer to 1-D array

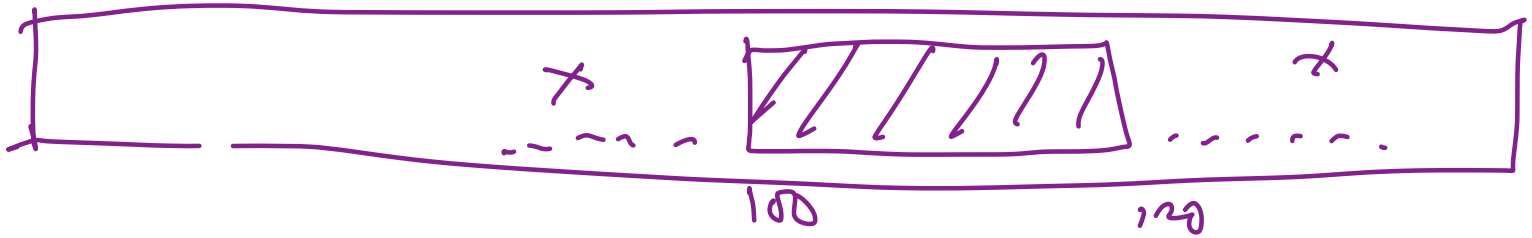
- ptr + 2 = 108 → *(ptr+2) = 33
- ptr – 2 = 92 → *(ptr-2) = ?
- int *ptr1 = &arr[3], *ptr2 = arr;
- ptr1 – ptr2 = 3
- short *ptr3 = &arr[3]; int *ptr4 = arr;
- ptr3 – ptr4 = $\frac{112-100}{\text{s.f.(ptr3)}} = \frac{112-100}{2} = 6$
- arr and &arr[0] are same.
  - arr + 1 = 104
  - &arr[0] + 1 = 104
- arr and &arr are not same.
  - arr + 1 = 104
  - &arr + 1 = 120

```
int arr[5] = {11, 22, 33};
int *ptr = arr;
```

$\frac{112-100}{4} = \frac{12}{4} = 3$



arr = 100
&arr[0] = 100
&arr = 100



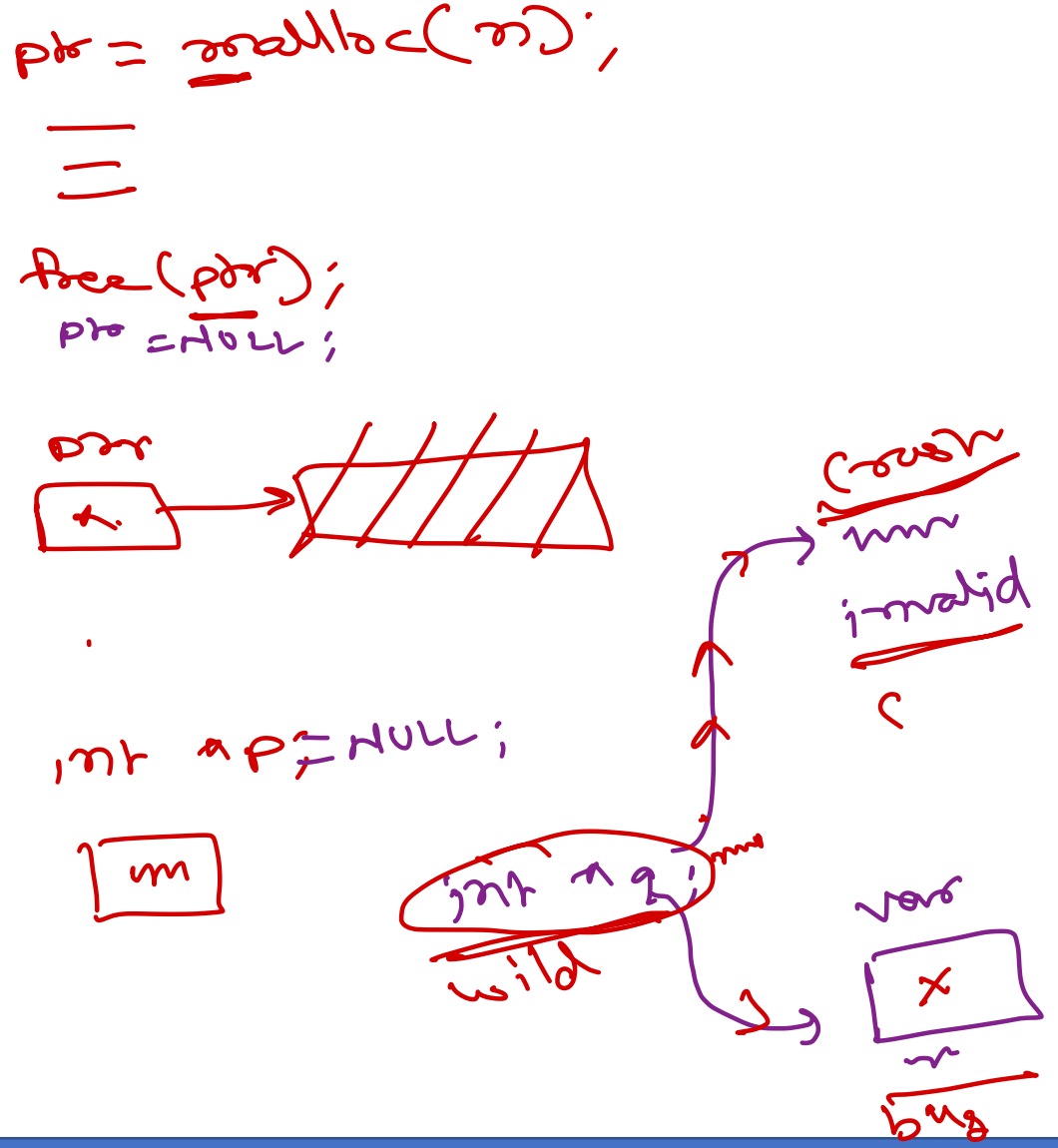
Passing 1-D array to function & returning from function

```
int main() {
    int arr[] = {11, 22};
    display(arr);
    return 0;
}

void display(int a[]) {
    int i;
    for(i=0; i<2; i++)
        printf("%d\n", a[i]);
}
```

```
int main() {
    int i;
    int *ptr;
    ptr = generate();
    for(i=0; i<2; i++)
        printf("%d\n", ptr[i]);
    return 0;
}

int *generate() {
    static int arr[] = {11, 22};
    return arr;
}
```

Dangling pointer

- Pointer keeping address of a memory location that is not valid for the application, is called as “dangling pointer”.
- Accessing value at dangling pointer will abort the program.
- Reasons for dangling pointers:
  - After releasing dynamically allocated memory, then pointer become dangling.
  - Uninitialized pointers contains garbage address, so they may be dangling.
  - The pointer collecting address of local variable from the function may be dangling.



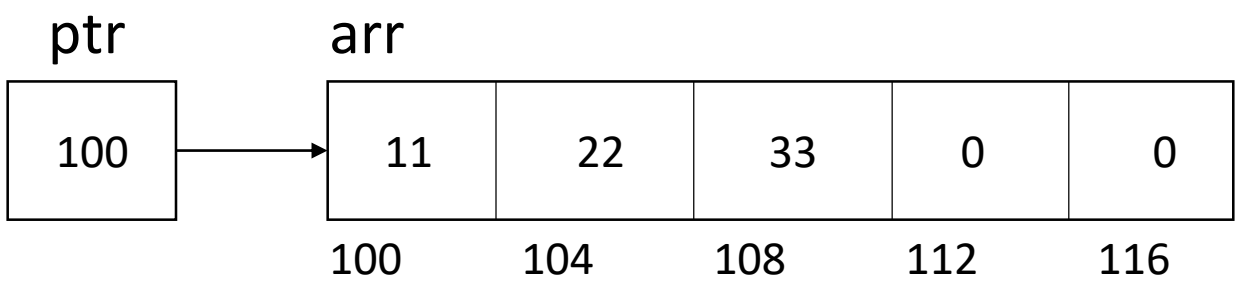
Memory leakage

- If dynamically allocated memory is not released, it is said to be memory leakage.
- This may be possible due to overwriting contents of pointer (returned from malloc function).
- In modern OS, this memory will be automatically released when process is terminated.
- All memory related errors can be tested in Linux using *valgrind* tool.

Operators with pointers

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (*type*) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

```
int main() {
    int arr[5] = {11, 22, 33};
    int *ptr = arr;
    printf("%d, %d, %d, %d, %d, %d\n",
        *ptr, ++*ptr++, (*ptr)++, *ptr++, *++ptr, ++*ptr);
    return 0;
}
```



Pointer to 2-D array

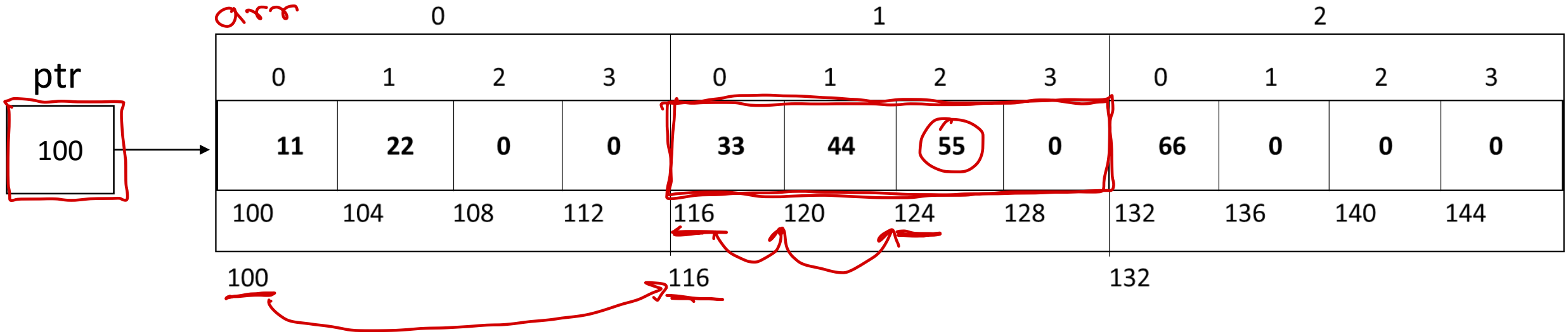
- 2-D array
  - Collection of Rows or Columns?
  - Array of 1-D arrays?
- int arr[3][4] = {{11, 22}, {33, 44, 55}, {66}};
- Elements:
  - arr[1][2] =
  - arr[2][-3] =
- Pointer to 2-D array is pointer to 0th element of the array. The 0th element itself is array of 4 ints.
  -
  -
  - Scale factor of the pointer is 16 bytes.
  - +1 takes to next 1-D array.

	0	1	2	4
0	11	22	0	0
1	33	44	55	0
2	66	0	0	0

- x[y] = *(x + y)
- arr[i][j]
- = (*(arr + i))[j]
- = *(*(arr + i) + j)

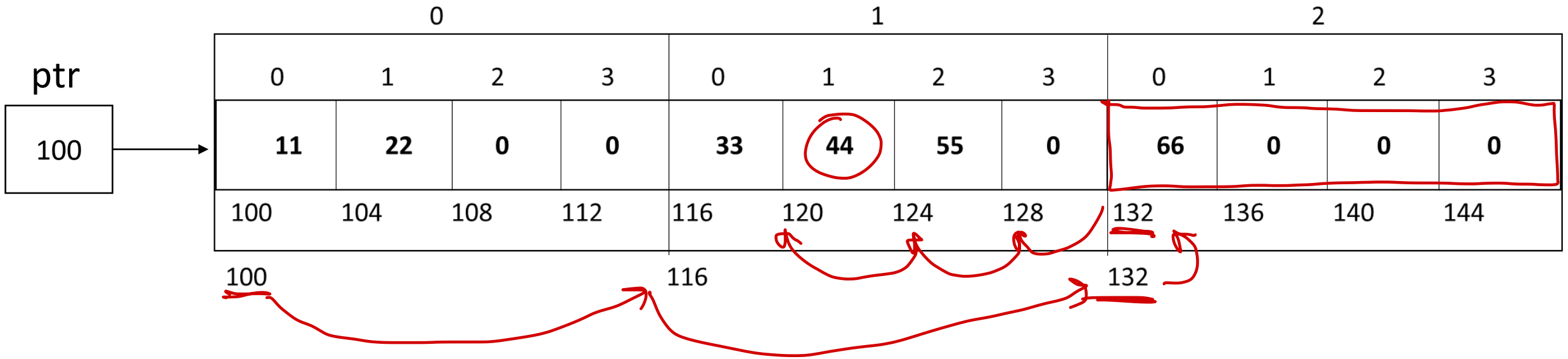
Pointer to 2-D array

arr+n → addr of nth ele.
*(arr+n) → nth ele.



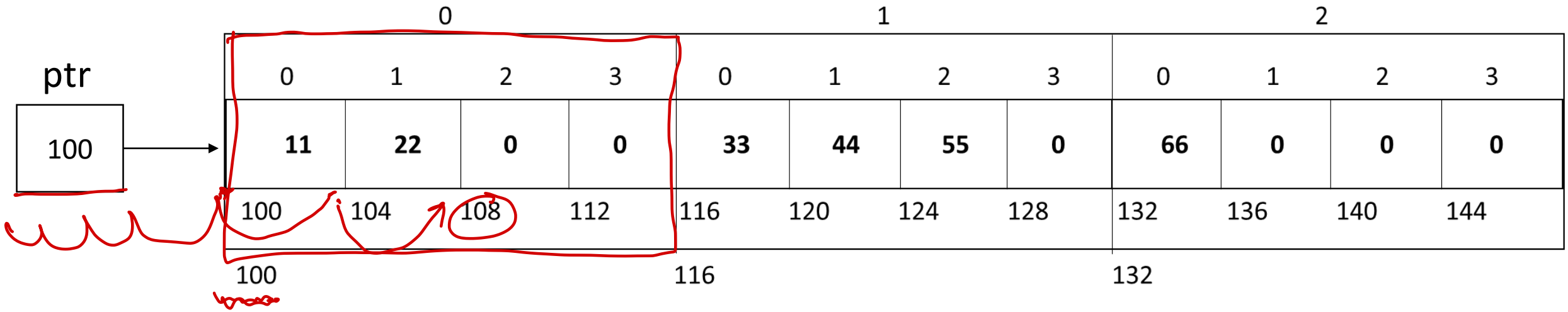
- arr[1][2]
  - arr = 100
  - arr + 1 = 116
  - *(arr + 1) = 116
  - *(arr + 1) + 2 = 124
  - *(*(arr + 1) + 2) = 55
- arr[2][-3]
  - arr =
  - arr + 2 =
  - *(arr + 2) =
  - *(arr + 2) – 3 =
  - *(*(arr + 2) – 3) =
- ptr[-1][6]
  - ptr =
  - ptr - 1 =
  - *(ptr - 1) =
  - *(ptr - 1) + 6 =
  - *(*(arr - 1) + 6) =

Pointer to 2-D array



- arr[1][2]
  - arr =
  - arr + 1 =
  - *(arr + 1) =
  - *(arr + 1) + 2 =
  - *(*(arr + 1) + 2) =
- arr[2][-3]
  - arr = 100
  - arr + 2 = 132
  - *(arr + 2) = 132
  - *(arr + 2) – 3 = 120
  - *(*(arr + 2) – 3) = 44
- ptr[-1][6]
  - ptr =
  - ptr - 1 =
  - *(ptr - 1) =
  - *(ptr - 1) + 6 =
  - *(*(arr - 1) + 6) =

Pointer to 2-D array



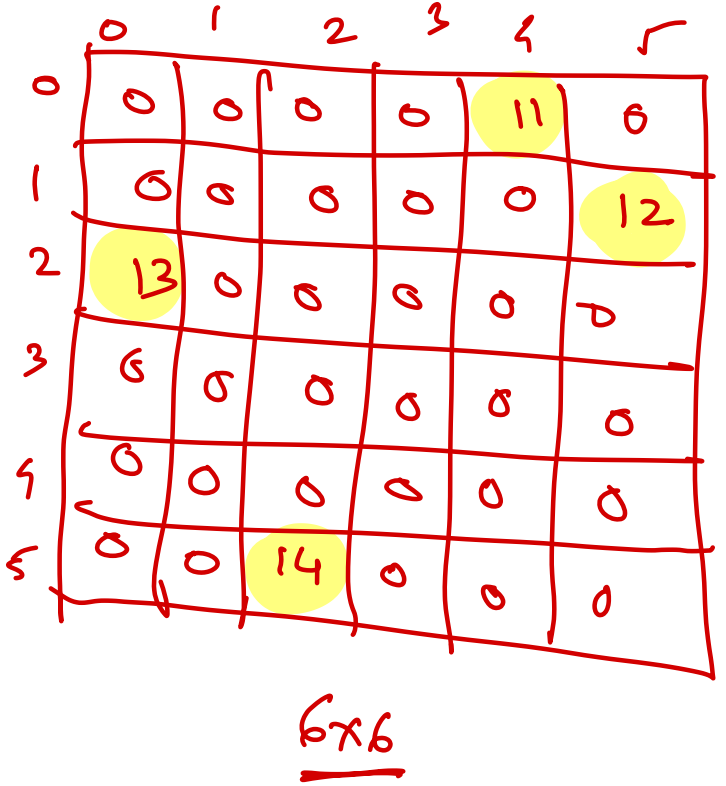
- arr[1][2]
  - arr =
  - arr + 1 =
  - *(arr + 1) =
  - *(arr + 1) + 2 =
  - *(*(arr + 1) + 2) =
- arr[2][-3]
  - arr =
  - arr + 2 =
  - *(arr + 2) =
  - *(arr + 2) – 3 =
  - *(*(arr + 2) – 3) =
- ptr[-1][6]
  - ptr =
  - ptr - 1 =
  - *(ptr - 1) =
  - *(ptr - 1) + 6 =
  - *(*(arr - 1) + 6) =

Passing 2-D array to function & returning from function

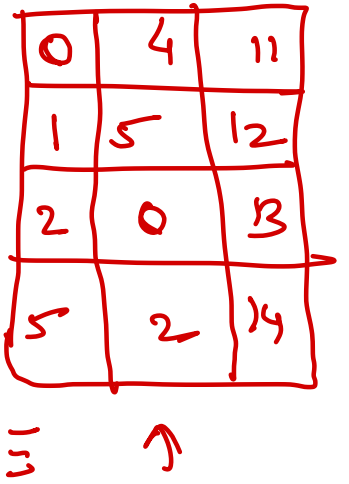
```
int main() {
    int arr[][2] = {11, 22, 33, 44};
    display(arr);
    return 0;
}

void display(int a[][2]) {
    int i, j;
    for(i=0; i<2; i++) {
        for(j=0; j<2; j++)
            printf("%d\n", a[i][j]);
    }
}
```

```
int main() {
    int i, j;
    int (*ptr)[2];
    ptr = generate();
    for(i=0; i<2; i++) {
        for(j=0; j<2; j++)
            printf("%d\n", a[i][j]);
    }
    return 0;
}

int (* generate())[2] {
    static int arr[][2] = {11, 22, 33, 44};
    return arr;
}
```

sparse matrix → data structure → Sahani



6x6



Sparse matrix efficient Storage

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>