MASTER CLASS ON ADVANCED POINTERS IN C

PreCAT @ SunBeam Infotech

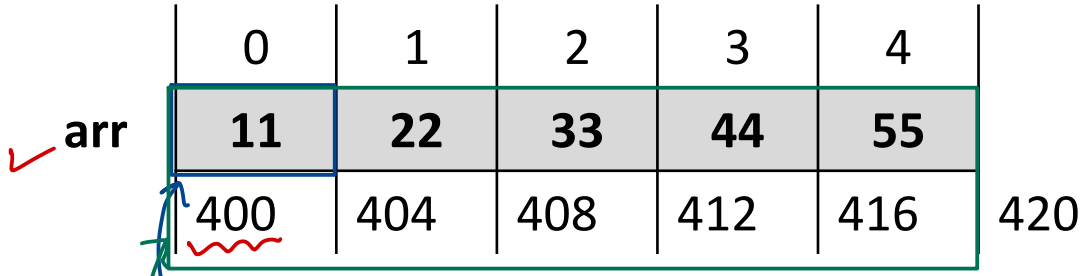
SPEAKER: MR. NILESH GHULE

Contents

1. Pointer to multi-dimensional array
2. Structure dot and arrow operator
3. Function pointers and applications
4. Generic programming and void pointers * e.g. qsort() bsearch()
5. Complex Pointer Declarations

1-D Array and Pointers

"arr" array name is treated as addr of 0th element i.e. &arr[0] in any runtime expr.




```
int main() {
        int arr[5] = { 11, 22, 33, 44, 55 };
        printf("%d, %lu, %lu\n", *arr, arr, &arr);
        printf("%d, %lu, %lu\n", *arr + 1, arr + 1, &arr + 1);
        return 0;
}
```

on stack (local)

11

400 = addr of 0th ele of arr

400 = addr of whole array

*arr + 1 → 12

arr + 1 → 404

&arr + 1 → 420

2-D array

- Logically 2-D array represents m x n matrix i.e. m rows and n columns.
  - int arr[3][4] = { {1, 2, 3, 4}, {10, 20, 30, 40}, {11, 22, 33, 44} };

3 rows 4 cols

if arr elems are initialized partially at its point of declaration, remaining elems are initialized to zero.

	0	1	2	3
r 0	1	2	3	4
r 1	10	20	30	40
r 2	11	22	33	44

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

1	2	0	0
10	0	0	0
11	22	33	0

  - int arr[3][4] = { 1, 2, 10, 11, 22, 33 };

1	2	10	11
22	33	0	0
0	0	0	0

arr[1][2] = 30

arr[1][5] = ?

arr[2][-3] = ?

  - int arr[x][4] = { 1, 2, 10, 11, 22, 33 };

0	1	2	10	11
1	22	33	0	0

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

↳ Num of columns is compulsory.

2-D array

- 2-D array is collection of 1-D arrays in contiguous memory locations. ✓
  - Each element is 1-D array. (all 1D array of same size).
- int arr[3][4] = { {1, 2, 3, 4}, {10, 20, 30, 40}, {11, 22, 33, 44} };

x[y] = *(x + y)

a[i][j] = *(*(a+i)+j)

"arr" is array of 3 elems, each elem is array of 4 ints.

	0				1				2			
	0	1	2	3	0	1	2	3	0	1	2	3
arr	**1**	**2**	**3**	**4**	**10**	**20**	**30**	**40**	**11**	**22**	**33**	**44**
	400	404	408	412	416	420	424	428	432	436	440	444
	400				416				432			

2-D array and Pointer

a + i = addr of ith elem
*(a + i) = ith elem. = a[i]

ptr	arr	0				1				2			
		0	1	2	3	0	1	2	3	0	1	2	3
400		1	2	3	4	10	20	30	40	11	22	33	44
1000		400	404	408	412	416	420	424	428	432	436	440	444
		400				416				432			

arr[1][2]
= *(*(arr + 1) + 2)
arr = 400
arr + 1 = 416
*(arr + 1) = 416
*(arr + 1) + 2 = 424
((arr + 1) + 2) = 30

arr[1][5]
= *(*(arr + 1) + 5)
arr = 400
arr + 1 = 416
*(arr + 1) = 416
*(arr + 1) + 5 = 436
((arr + 1) + 5) = 22

arr[2][-3]
= *(*(arr + 2) - 3)
arr = 400
arr + 2 = 432
*(arr + 2) = 432
*(arr + 2) - 3 = 420
((arr + 2) - 3) = 20

2-D array and Pointer

int arr[3][4];

ptr	arr	0				1				2			
		0	1	2	3	0	1	2	3	0	1	2	3
400		1	2	3	4	10	20	30	40	11	22	33	44
1000		400	404	408	412	416	420	424	428	432	436	440	444
		400				416				432			

pointer to array = pointer to 0th elem of array.

S.F. of pointer = size of 0th elem.
(1D array)

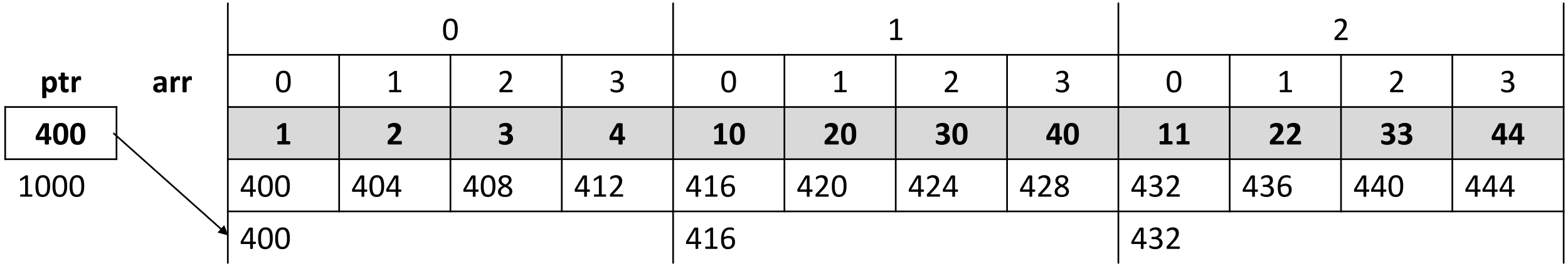
= 16 bytes.

int (*ptr)[4]; // s.f. 16 bytes

ptr = arr; or ptr = &arr[0];

2-D array and Pointer

- Pointer to array is pointer to 0th element of the array.
  - Scale factor of the pointer = number of columns * sizeof(data-type).
- int arr[3][4] = { {1, 2, 3, 4}, {10, 20, 30, 40}, {11, 22, 33, 44} };
- int (*ptr)[4] = arr;



Passing 2-D array to Functions

- 2-D array is passed to function by address.
- It can be collected in formal argument using array notation or pointer notation.
- While using array notation, giving number of rows is optional. Even though mentioned, will be ignored by compiler.

In C, arrays are always pass by address (collected in pointer).

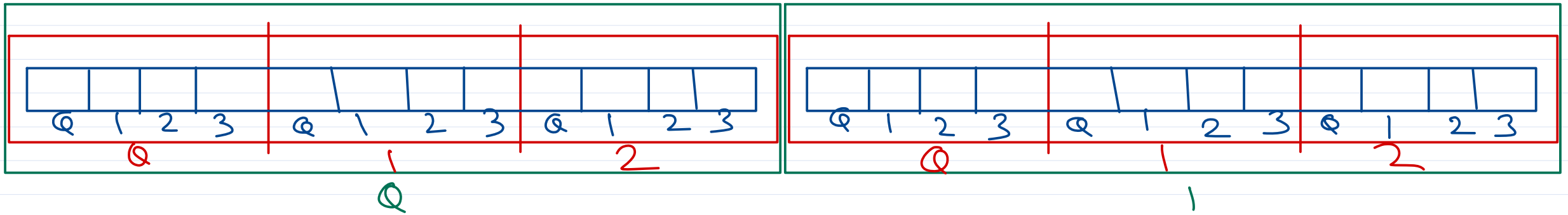
```
int main() {
    int arr[3][4] = { ... };
    print(arr);
    return 0;
}

void print(int (*a)[4])
                ↑ by compiler
//void print(int a[][4])
{
    pf("%d", sizeof(a)); // 4 or 8
    ...
}
```

```
int (*myfun())[4]
{
    static int arr[3][4];
    //input from user - scanf.
    return arr;
}
int main() {
    int (*ptr)[4];
    ptr = myfun();
    ...
    return 0;
}
```

Q. dynamically allocate (contiguous) 2-D array of size m x 3

```
int (*ptr)[3];
ptr = (int(*)[3])malloc(m * 3 * sizeof(int));
memset(ptr, 0, m * 3 * sizeof(int));
for(i=0; i<m; i++) {
    for(j=0; j<3; j++) {
        pf("%d ", ptr[i][j]);
    }
}
free(ptr);
```

```
int arr[2][3][4] = {1, 2, 3, 4, ... , 24};
```



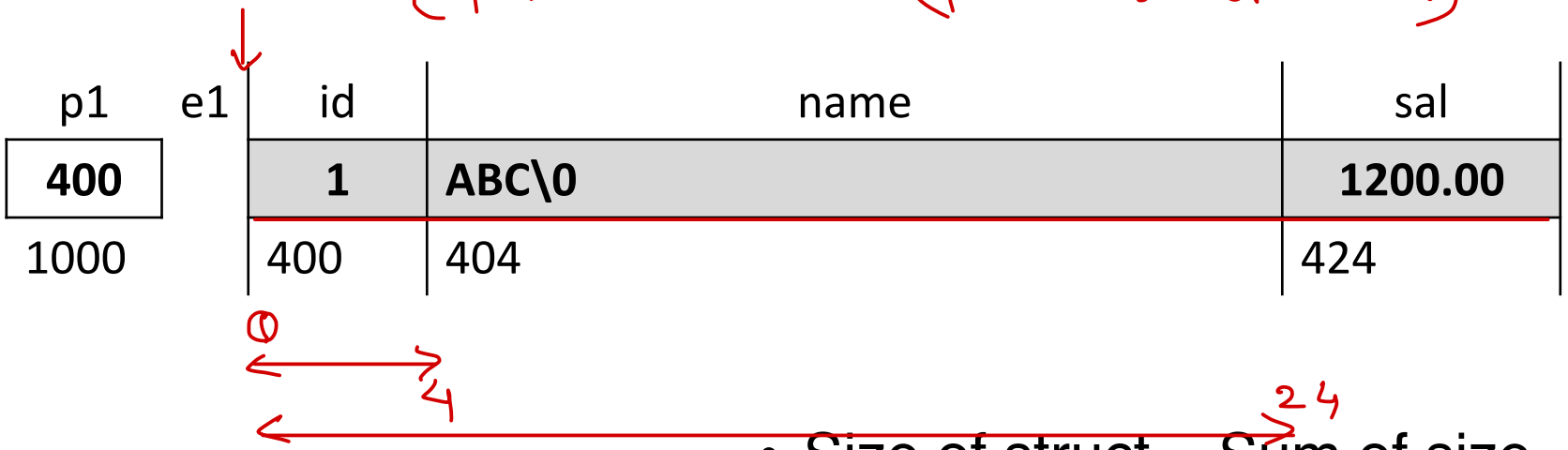
arr[1][2][3] = ?

((*(arr + 1) + 2) + 3)

Structure dot and arrow operator

a[i] = *(a+i)
v.member = *(&v + offset of member)
p→member = *(p + offset of member)

```
#pragma pack(1)
struct emp {
    int id;        // 4
    char name[20]; // 20
    double sal;    // 8
};
int main() {
        struct emp e1 = { 1, "ABC", 1200};
        struct emp *p1 = &e1;
        printf("%d %s %lf\n", e1.id, e1.name, e1.sal);
        printf("%d %s %lf\n", p1→id, p1→name, p1→sal);
        return 0;
}
```



- Size of struct = Sum of size of members
  - sizeof(struct emp) = 32
- Offset of members
  - id = 0
  - name = 4
  - sal = 24

Can be calculated programmatically using offsetof() macro in linux kernel (dev dm).

Structure dot and arrow operator

p1	e1	id	name	sal
400		**1**	**ABC\0**	**1200.00**
1000		400	404	424

```
struct emp e1 = {-, -, -3;
struct emp *p1;
  p1 = &e1;
pf("%lf", *(double*)((char*)&e1 +24));
pf("%lf", *(double*)((char*) p1 +24));
```

&e1 = 400

struct size = 32

~~&e1 + 24 = 400 + 24 * 32~~

(char*)&e1 +24 = 424

~~*((char*)&e1 +24)~~ → will read only 1 byte at addr 424. → S.f. of char*

(double)((char*)&e1 +24) = 1200.00

```
char *p = NULL;
short *q = NULL;
float *r = NULL;
double *s = NULL;
pf("%.1u %.1u %.1u %.1u", p, q, r, s);
pf("%.1u %.1u %.1u %.1u", ++p, ++q, ++r, ++s);
```

NULL == 0 == (void *) 0

0 0 0 0

1 2 4 8

Pointer to Function

fn name = addr of fn (like array)

- Pointer to function stores address of the first instruction of the function.
- It is used to invoke function dynamically.
- Applications
  - Call-back functions
    - ISR
    - Entry point function → main()
    - C functions like qsort() and bsearch()
    - Linux signal handler → signal (SIGINT, my_handler);
    - Linux device driver operations
  - C++ virtual functions (late binding)
- Similar concepts in other programming languages
  - C#.NET – delegate
  - Java – Method object or Method references
  - Java Script / Python / Swift / Kotlin – Functions are first-class objects

Functional Programming
1) immutability
2) fn → first class citizen

Pointer to Function

```
int sum(int p, int q) {
  return p + q;
}
int subtract(int p, int q) {
  return p - q;
}
```

size = 4 or 8 bytes

int (*p)(int, int);

```
int main() {
    int (*p1)(int, int);
    int (*p2)(int, int);
    int res;
    p1 = sum;
    p2 = subtract;
    res = p1(12, 4);
    printf("%d\n", res);
    res = p2(12, 4);
    printf("%d\n", res);
    return 0;
}
```

declaration

initialization

call

Pointer to Function

```
int main() {
    int (*arr[2])(int,int);
    int res, i;
    arr[0] = sum;
    arr[1] = subtract;
    for(i=0; i<2; i++) {
        res = arr[i](12, 4);
        printf("%d\n", res);
    }
    return 0;
}
```

sum()

subtract()

```
int sum(int p, int q) {
  return p + q;
}
int subtract(int p, int q) {
  return p - q;
}
```

Pointer to Function

```
struct op {
  int x, y;
  int (*fn)(int,int);
};
int main() {
  struct op a = { 12, 3, sum };
  struct op b = {12, 3, subtract };
  printf("%d\n", calculate(&a));
  printf("%d\n", calculate(&b));
  return 0;
}
```

fn pointer

a | 12 | 3 | *

b | 12 | 3 | *

```
int sum(int p, int q) {
  return p + q;
}
int subtract(int p, int q) {
  return p - q;
}
int calculate(struct op *ptr) {
  return ptr->fn(ptr->x, ptr->y);
}
```

15

9

clojure = function + external data

```
int x = 3;
stream.map( a -> a+x );
....;
```

closure

void pointer

- Void pointer is generic pointer it can hold address of any data type (without casting).
- Scale factor of void* is not defined, so cannot perform pointer arithmetic.
- To retrieve value of the variable need type-casting.
- void* is used to implement generic algorithms e.g. qsort(), bsearch(), fread(), fwrite(), etc.
- Example:
  - Write a function to swap two variables (of any type).
  - Write a function to sort array of any type.

Complex pointer declarations

- const int *p = &a;
- int const *p = &a;
- int * const p = &a;
- int * p const = &a;
- const int * const p = &a;
- int const * const p = &a;
- int *x[5];
- int* x[5];

int * x[5];

same = x is array of 5 int pointers.

- int (*y)[5];

y is ptr to array of 5 int (s.f. 5*4=20).

- int (*z)[2][3];

z is ptr to 2-D array of 2x3 dim of int type. s.f. = 2*3*4 = 24.

- void (*p)(int);

p is pointer to fn that takes int & return void.

- char* (*q)(char*, const char*);

q = strcpy;

- void* (*q[3])(void*);

q is array of 3 elems each elem is ptr to fn that takes void* & return void*

Complex pointer declarations

- Declarations should be read starting from the name and then following preceding order.
- Precedence Level1: Grouping parenthesis.
- Precedence Level2: Postfix operators i.e. () indicating function, [] indicating array.
- Precedence Level3: Prefix operator i.e. * indicating pointer.
- const/volatile next to type, applies to type. In other cases, const/volatile applies to pointer asterisk before it.

Complex pointer declarations

- char * const * (*next)();
- int (*ptr(float))[4];

ptr is fn that takes float & returns pointer to array of 4 ints.

- int (*(*ptr)(float))[4]; → pointer to above fun.
- char *(*c[10])(int **);

array of 10 fn pointers each fn takes int** & returns char*.

- char *(*c)(int **) [10];
- void (*signal(int, void (*)(int))) (int);

arg1 arg2 = fn*

returns = fn*

typedef and pointers

- typedef is not a macro to replace type.
  - #define char_ptr_t char*
  - char_ptr_t p1, p2;
  - typedef char* char_ptr_t;
  - char_ptr_t p3, p4;
- typedef simplify the declaration.
  - int (*ptr)[5];
  - void (*signal(int, void (*)(int))) (int);

typedef and pointers

- typedef is not a macro to replace type.
  - #define char_ptr_t char*
  - char_ptr_t p1, p2;
  - typedef char* char_ptr_t;
  - char_ptr_t p3, p4;
- typedef simplify the declaration.
  - int (*ptr)[5];
  - void (*signal(int, void (*)(int))) (int);
- Declare a pointer to the function that takes pointer to array of three integers and return array of three integer.

THANK YOU!

NILESH GHULE <nilesh@sunbeaminfo.com>