

# Arrays and pointers

**R. Inkulu**

**<http://www.iitg.ac.in/rinkulu/>**

# Motivation with Example usage

```
int func(void) {  
    int states[5], i; //defining  
    for (i=0; i<5; i++)  
        states[i]=i; //initializing  
    for (i=0; i<5; i++)  
        printf("%d, ", states[i]); //accessing the value  
    return 0;  
}
```

- array is used in storing multiple objects of same type
- objects in the array are indexed starting from 0
- array scope is limited to the function in which it is defined

# sizeof operator

```
int func(void) {  
    int a[5];  
    printf("%d, %d, %d \n",  
           sizeof(int), sizeof(a[3]), sizeof(a));  
           //outputs 4, 4, 20  
}
```

sizeof is a compile-time unary operator to find the size (in bytes) of a type or the size of the type returned by an expression:

- `size_t sizeof(type)`
- `size_t sizeof expression`<sup>1</sup>

---

<sup>1</sup>object is an expression  
(Arrays and pointers)

# Two-dimensional arrays

```
int func(void) {  
    int a[2][3], i, j;    //defining  
  
    for (i=0; i<2; i++)  
        for (j=0; j<3; j++)  
            a[i][j] = i+j;    //initializing  
  
    for (i=0; i<2; i++)  
        for (j=0; j<3; j++)  
            printf("%d, ", a[i][j]);    //accessing the value  
  
    printf("\n %d, %d, %d", sizeof(a),  
           sizeof(a[1]), sizeof(a[1][2]));  
    //prints 24, 12, 4  
  
    return 0;  
}
```

# Three-dimensional arrays

```
int func(void) {
    int a[2][3][4], i, j, k;    //defining

    for (i=0; i<2; i++)
        for (j=0; j<3; j++)
            for (k=0; k<4; k++)
                a[i][j][k] = i+j;    //initializing
    for (i=0; i<2; i++)
        for (j=0; j<3; j++)
            for (k=0; k<4; k++)
                printf("%d, ", a[i][j][k]);    //accessing

    printf("\n %d, %d, %d, %d \n", sizeof(a), sizeof(a[1]),
        sizeof(a[0][2]), sizeof(a[1][0][2]));
    //prints ?
    return 0;
}
```

# Address vs Value

```
int a, b[2];

a = 10;
printf("%d, %p \n", a, &a);
    //prints 10, 0xbfeac818

b[0] = 22; b[1] = 24;
printf("%d, %p, %d, %p \n", b[0], &b[0], b[1], &b[1]);
    //prints 22, 0xbfeac820, 24, 0xbfeac824
```

- memory for arrays is allocated contiguously

# Arrays are stored in row-major order

```
int c[2][3], i, j;  
for (i=0; i<2; i++)  
    for (j=0; j<3; j++) {  
        printf("%p, ", &c[i][j]);  
    }  
    //prints 0xbf d8f650, 0xbf d8f654, 0xbf d8f658,  
    //0xbf d8f65c, 0xbf d8f660, 0xbf d8f664,
```

- contiguous memory is allocated for the zeroth row, immediately thereafter for the first row, etc., — known as *row-major order*
- further, two-dimensional array is treated as *an array of arrays*: the elements of  $c$  are  $c[0], c[1]$ , each of which is an array i.e., the  $i^{th}$  row of  $c$  is named  $c[i]$

## Arrays are stored in row-major order (cont)

```
int a[2][5][3], i, j, k;
for (i=0; i<2; i++)
    for (j=0; j<5; j++)
        for (k=0; k<3; k++)
            printf("%p, ", &a[i][j][k]);
//observed: difference between any two successive
//addresses printed is four
```

- three-dimensional array is treated as an *array of arrays of arrays*: elements of  $a$  are  $a[0], a[1]$ , each of which is an array i.e., the  $i^{th}$  row of  $a$  is denoted with  $a[i]$  each element of which is an array i.e.,  $j^{th}$  element of  $a[i]$  is denoted with  $a[i][j]$ , which is again an array in depth
- further, contiguous memory is allocated for arrays in the following order:  $c[0][0], \dots, c[0][4], c[1][0], \dots, c[1][4]$  — known as *row-major order*



# Initializing arrays while defining

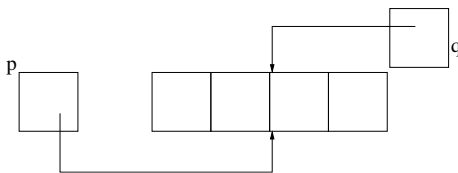
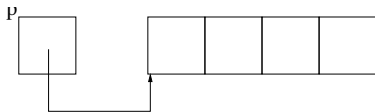
```
int i, j;  
int a[2][5] = { //also allowed a[][5]  
    {0, 11, 22, 33, 44},  
    {1, 12, 23, 34, 45}  
    //with no flower brackets is fine too  
};  
  
for (i=0; i<2; i++)  
    for (j=0; j<5; j++)  
        printf("%d, ", a[i][j]);  
//prints 0, 11, 22, 33, 44, 1, 12, 23, 34, 45,
```

# Introducing pointers

```
int x = 1, y = 2, z[2];  
int *p = NULL;  
    //p is a pointer to int, initialized with NULL  
  
p = &x;        //p points to x  
y = *p;        //y is 1  
*p = 0;        //x is now 0  
p = &z[1];     //p points to z[1]  
*p = 5;        //z[1] is 5  
*p = *p + 5;   //z[1] is 10
```

- *pointer* is just a variable that saves the address of a memory location that contains an object (or value) of specific type
- when  $p$  points to a *typeA*,  $p$  is termed as a *pointer to typeA*
- *dereferencing pointer p* is denoted with  $*p$

# Incrementing a pointer



```
int z[4];  
int *p = &z[0];  
*(p+1) = 7;           /*(p+(1*sizeof(int))) = 7  
                       //i.e., z[1] has 7  
  
p += 2;               //p = p + (2*sizeof(int))  
*p = 5;               //z[2] has 5  
*p = *p + 3;          //z[2] has 8  
*p += 2;              //z[2] has 10  
++(*p);               //z[2] has 11  
int *q = p;           //q also points to z[2]  
*q = 89;              //z[2] has 89
```

- when  $p$  is a pointer to typeA, expression  $p + i$  increments  $p$  by  $i * \text{sizeof}(\text{typeA})$

# Valid pointer operations

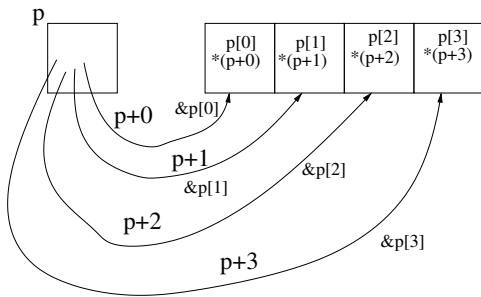
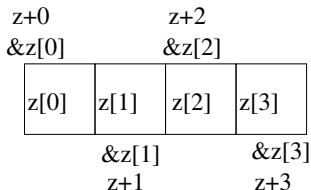
- assignment of pointers of the same type
- adding (resp. subtracting) an integer to (resp. from) a pointer  $p$  (new address is meaningful whenever  $p$  points to an appropriate entry in an array)
- subtracting<sup>2</sup> or comparing two pointers to members of the same array
- assigning or comparing to zero

All other pointer arithmetic is illegal: not legal to add two pointers, or to multiply or divide or shift or mask them, or to add float or double to them, or even, except for void \*, to assign a pointer of one type to a pointer of another type without a type cast.

---

<sup>2</sup>for an array that holds objects of type  $T$ , such a subtraction yields the number of objects of type  $T$  that block of array memory can hold (consistent with the way the compiler expands  $p + i$ )

# Ways to access an array



```
double z[4], *p = z;  
...  
z[3] = 10;  
...
```

- ex. compiler replaces `z[i]` with `*(z + (i * sizeof(double)))`  
same is true in case of multi-dimensional arrays

# Name of an array

```
float z[4], *p=z;
printf("%d, %d, %d \n", sizeof(p),
      sizeof(float), sizeof(*p));
//prints 4, 4, 4

printf("%p, %p, %p \n", z, &z[0], &z);
//identical values are printed: 0xbfd30990

printf("%p, %p, \n", z+1, &z[0]+1);
//identical values (&z[1]) are printed: 0xbfd30994

printf("%p \n", &z+1);
//prints (&z[0] + (1*sizeof(z))): 0xbfd309a0
```

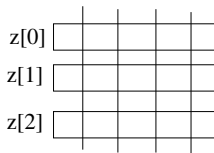
- name of an array  $z$  is an alias for the address of zeroth element of  $z$ ; further,  $\&z[i] + j$  denotes the address of  $z[i + j]$
- statements like  $z = p$  and  $z++$  are illegal as  $z$  is not a variable

# Array name and pointer variable

```
float z[4];  
float *p = &z[0], *q;  
  
q = z;  
printf("%p, %p, %p \n", z, p, q);  
    //identical values are printed  
printf("%p, %p \n", p, &p);  
    //addresse stored in p vs address of p  
  
z[2] = 30;        //z[2] has 30  
*(z+2) = 20;     //z[2] has 20  
q[3] = 52;        //z[3] has 52  
*(q+3) = 58;     //z[3] has 58
```

- $z[i]$  gets expanded as  $*(z + i)$

# Row-major order of arrays

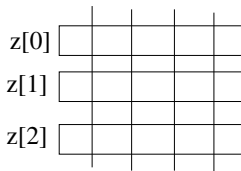


```
int z[3][4];
printf("%p, %p, %p, \n", z[0], z[1], z[2]);
    //prints addresses of three rows
printf("%p, %p, %p \n", &z[0], &z[1], &z[2]);
    //prints addresses of three rows
printf("%p, %p \n", &z[1]+1, &z[0]+2);
    //prints identical values; z is interpreted as &z[0]
printf("%p, %p \n", z[2]+1, &z[2][1]);
    //prints identical values
```

- two-dimensional array is an array of rows:  
 $z[i]$  is the name of the array that comprises the  $i^{th}$  row;  
and,  $\&z[i]$  is the address of the  $i^{th}$  row



# Row-major order of arrays (cont)



```
int z[3][4];  
printf("%p, %p \n" z[1], &z[1]);  
//prints identical addresses  
printf("%p, %p, %p \n", z, z+1, z+2);  
//prints z[0], z[1], z[2]
```

- $z[i]$  is the name of the array containing  $i^{th}$  of row of  $z$
- for any  $i$ ,  $z[i]$  does not occupy additional space; exists only for notational convenience
- $z$  is the name of the array comprising  $z[0], \dots, z[n]$ ; further,  $z$  is the symbol for the address of  $z[0]$

# Row-major order of arrays (cont)

```
int b[3][4];  
b[2][1] = 23;
```

`b[2][1]` gets expanded to `*( *(b+2) + 1 )`:

- `*(b+2)` is `*(&b[0]+2)` i.e., `*(&b[0] + (2*sizeof(b[0])))`, meaning `b[2]`
- further, `*(b[2]+1)` is `*(&b[2][0]+1)` i.e., `*(&b[2][0] + (1*sizeof(b[2][0])))`, meaning `b[2][1]`

# Row-major order of arrays (review)

- consider 'double z[3]':

$z[0]$ ,  $z[1]$ ,  $z[2]$

$\&z[0]$ ,  $\&z[1]$ ,  $\&z[2]$

$z[0]+1$ ,  $\&z[0]+1$ ,  $\&z[0]+2$

$z$ ,  $z+1$ ,  $z+2$

$\&z$ ,  $\&z+1$

- consider 'double z[3][4]':

$z[0]$ ,  $z[1]$ ,  $z[2]$

$\&z[0]$ ,  $\&z[1]$ ,  $\&z[2]$

$z[0]+1$ ,  $\&z[0]+1$

$z[0][0]+1$ ,  $\&z[0][0]+1$

$z$ ,  $z+1$ ,  $z+2$

$\&z$ ,  $\&z+1$

$\&z[i] + j$  is equivalent to the address of  $z[i]$  plus  $(j * \text{sizeof}(z[i]))$

$z + j$  gets converted to  $\&z[0] + j$ , whereas  $\&z + j$  is the address of  $z$  plus  $j * \text{sizeof}(z)$

# Operator precedence and associativity

Precedence (highest to lowest)	Associativity
() [] -> .	left to right
! ++ -- & *(type) sizeof	right to left
* / %	left to right
+ -	left to right
= += -= *= /=	right to left

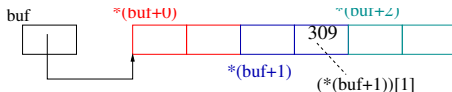
```
a+b*c-d/e;    //(a+(b*c))-(d/e)
*p[];         //*(p[])
val = *--p;    //val = *(--p)
*p++ = val;    //*(p++) = val i.e., *p = val; p = p+1
val = *p++;    //val = *(p++) i.e., val = *p; p = p+1
```

3

<sup>3</sup> result of the postfix increment (resp. decrement) operation is the value of the operand; after the value is obtained, the value of the operand is incremented (resp. decremented) as a side effect

(Arrays and pointers)

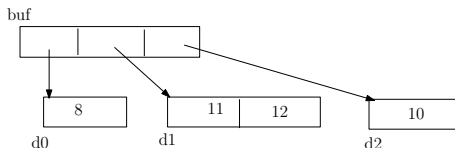
# Pointer to an array



buf points to double[2] i.e., two doubles that are contiguously allocated

```
double (*buf) [2];  
...  
printf("%d, %d, %d \n", sizeof(double), sizeof(*buf),  
sizeof>(* (buf+2)));  
//prints 8, 16, 16  
printf("%p, %p \n", buf, buf+1);  
//prints 0xbffd7e30, 0xbffd7e40  
buf[1][1] = 309;  
printf("%p, %lf \n", &buf[1][1], buf[1][1]);  
//prints 0xbffd7e48, 309.000000
```

# Array of pointers



buf is an array of pointers to doubles

```
double *buf[3];  
double d0 = 8, d1[2] = {11, 12}, d2 = 10;  
buf[0] = &d0; buf[1] = &d1[0]; buf[2] = &d2;  
  
printf("%d, %d \n", sizeof(void*), sizeof(buf));  
//prints 4, 12  
printf("%p, %p, %p, %lf \n",  
       buf, &buf[0], buf[1], *buf[1]);  
//prints 0xbfed51d4, 0xbfed51d4, 0xbfed51e0, 11.000000
```