POINTERS

DOUBLE POINTERS

- As pointers are also variables, they occupy space in the memory.
- Pointers hold addresses i.e integer values so they occupy 2 bytes in the memory.
- Hence, there is an address for pointers also.
- We can have another pointer variable pointing to that address.
- Such pointer variables are called as double pointers.

DOUBLE POINTERS

- Syntax:

datatype **pointer_name;

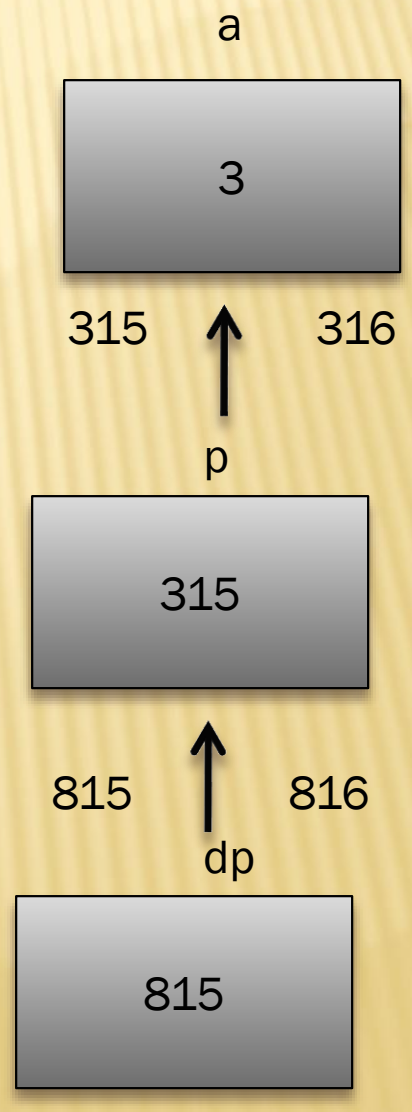
- Example:

int a, *p, **dp;

a=3;

p=&a;

dp=&p;



QUICK EXERCISE

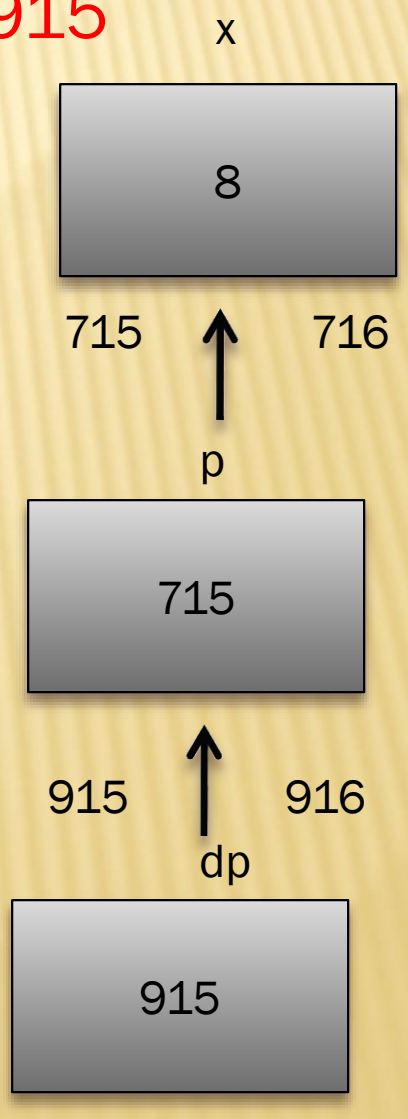
- Guess the appropriate format specifiers:

```
int x, *p, **dp;
x=8;
p=&x;
dp=&p;
printf("%_", p);
printf("%_", dp);
printf("%_", *p);
printf("%_", *dp);
printf("%_", **dp);
```

QUICK EXERCISE

- Assume the address of x is 715 and p is 915

```
int x, *p, **dp;
x=8;
p=&x;
dp=&p;
printf("%p", p);  → 715
printf("%p", dp); → 915
printf("%d", *p); → 8
printf("%p", *dp); → 715
printf("%d", **dp); → 8
```



POINTER ARITHMETIC

- Following arithmetic operations are possible on the pointer in C language:
  - Increment
  - Decrement
  - Addition
  - Subtraction
  - Comparison
- There are various operations which can not be performed on pointers.
  - Address + Address
  - Address * Address
  - Address % Address
  - Address / Address
  - Address & Address
  - Address ^ Address
  - Address | Address
  - ~Address

INCREMENTING/DECREMENTING POINTERS

- Incrementing Pointer in C
  - If we increment a pointer by 1, the pointer will start pointing to the next ***base address***.
  - The value of the pointer will get increased by the size of the data type to which the pointer is pointing.
- The Rule to increment the pointer is given below:

  new_address= current_address + sizeof(data type)
- Example:
  - For 16-bit variable, it will be incremented by 2 bytes (2 + current_address).
  - For 32-bit variable, it will be incremented by 4 bytes (4 + current_address).

EXAMPLE(ASSUMING ADDRESS = 715)

```
int number=50;
int *p;
p=&number;
printf("Address of p variable is %p\n",p);
p=p+1;   // Or p++; Or ++p;
printf("After increment: Address of p variable is %p \n",p);
```

Note: The same rule applies for decrementing pointers also

Output:

Address of p variable is 715

After increment: Address of p variable is 717

ADDITION OPERATION WITH POINTERS

- We can only add an int value to the pointer variable.
- The formula of adding integer value to pointer is given below:

new_address= current_address + (number * size_of(data type))

- Example:
- For 16-bit variable, it will add 2 * number.
- For 32-bit variable, it will add 4 * number.

EXAMPLE : ASSUMING ADDRESS = 715

```
int number=50;
int *p;
p=&number;
printf("Address of p variable is %p \n",p);
p=p+3;
printf("After adding 3: Address of p variable is %p \n",p);
```

Output:

Address of p variable is 715

After increment: Address of p variable is 721

SUBTRACTION WITH POINTERS

- Like pointer addition, we can subtract an integer value from the pointer variable.
- Subtracting any number from a pointer will give an address
- However, instead of subtracting a number, we can also subtract an address from another address (pointer).
- This will result in an integer number.
- It will not be a simple arithmetic operation, but it will follow the following rule.
- If two pointers are of the same type,

  Address2 - Address1 = (Subtraction of two addresses)/size of data type

EXAMPLE

```
int i = 100;
int *p = &i;
int *temp;
temp = p;
p = p + 3;
printf("Pointer Subtraction: %u - %u = %u",p, temp, p-temp);
```

Note: %u can also be used instead of %p for printing addresses. %u is for unsigned decimal numbers.

Output:

Pointer Subtraction: 1872817300 - 1872817288 = 3

QUICK EXERCISE
ASSUME INITIAL ADDRESS = 3024147676

```
float *ptr, f=3.25, d;
ptr = &f;
*ptr = 10.9;
printf("%u\n",ptr);
ptr++;
d=*ptr = 5.67;
printf("%f %f\n %u", f,d, ptr );
```

Output:

```
3024147676
10.900000  5.670000
3024147680
```

QUICK EXERCISE

```
int *p, a, *q, b, **dp;
a=14; b= 23;
p=&a;  q=&b;
dp=&p;
**dp = a +18;
*dp = q;
**dp = b + 26;
printf("%d  %d", a,b);
```

Output:
32 49

ARRAYS

- An array is a collection or a series of *similar* values stored in *consecutive* memory locations.
- Arrays are helpful when we want to store multiple items of the same type.
- Example application of an array:
  - Marks of all students in one class
  - Prices of all books in a bookstore
  - Names of all cities in a country.

ARRAYS

- Declaration of an array:

  datatype array_name [MAX_SIZE];

- datatype indicates that all elements of the array can only be of that type.
- array_name will be used further in the program to access the values of the array
- MAX_SIZE indicates the total number of elements that can be present in the array.
- Declaration allocates memory for the entire array.
- Can you guess the size of the space allocated for an array?

  sizeof(datatype) * MAX_SIZE

ARRAYS

- Initialization of an array in the line of declaration(Method 1):

  datatype array_name[MAX_SIZE]= {val1, val2, …valn};

- This can only be done in the line of declaration of the array.
- We can initialize any number of elements up to MAX_SIZE only.
- Accessing elements of an array:

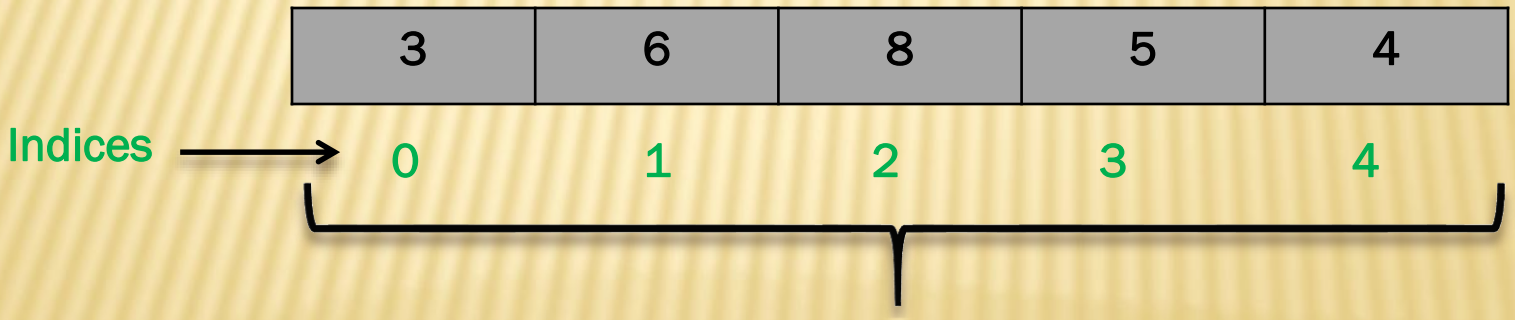
  array_name[position/index]

EXAMPLE

int a[5]= {3,6,8,5,4}; ← Initialization of the array

Declaration of the array

printf("%d", a[2]); ⟶ Accessing index = 2 i.e 3rd element = 8



Total size of array = size of each element*Max_size of array = 2*5 = 10 bytes

QUICK EXERCISE

- Find valid and invalid statements and show output wherever applicable:

int a[10]={ 2,6,7,1,5}; ✓

int b[10]; ✓

b={3,5,6,7}; X (Can only be done in the line of declaration)

int c[10]; ✓

printf("%d", a[3]); ⟶ 1

printf("%d", (a[0]*4)); ⟶ 8

printf("%d", (a[1*4]); ⟶ 5

printf("%d", a[6]); ⟶ 0. Uninitialized elements of "a" will automatically be set to 0

printf("%d", c[2]); ⟶ Garbage. Array "c" not initialized

printf("%d", a[10]); ⟶ X (Error. Index 10 does not exist for array "a")

printf("%d" a[2*0.5]); ⟶ X (Error. Array index can only be integers)