CS & IT ENGINEERING

C Programming

Pointers & Arrays

Lecture No.- 02

By- Satya sir

Recap of Previous Lecture

- Pointers
  - A variable that Points to any data
  - Holds address
  - Pointer datatype is Unsigned integer
  - Size of Pointer == Integer size
  - Pointer Arithmetic
  - Types of Pointers (void, Null, Dangling Pointer - - -)

Topics to be Covered

- Null Pointer
- Dangling Pointer
- 1-D array
  - Declaration
  - Initialize
  - Access Elements
  - Address of an Element

Null Pointer : A Pointer that Points to NULL (whose value is NULL)

void *p; // P is of type unsigned integer, that Can hold address of any data type

// it holds address (as value) but No Specific type.

int a = 5;

P = &a;

a — Name — P

[5] — Value — [A1]

A1 — address — A10

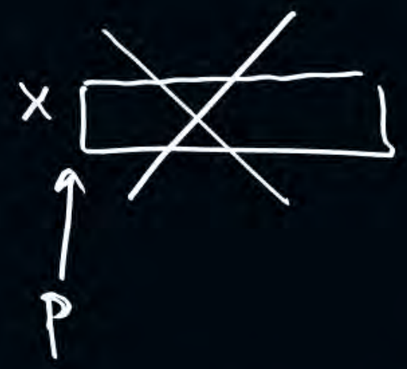
4 Bytes — size — 4 Bytes

char *P = NULL; // P is Pointer, whose Value is NULL

// P has Specific type, but No value

Dangling Pointer : Pointer, Pointing to Invalid location.

```
char x[5], *P;
P = &x[0];
 :
free(P); // deallocating memory, where P Pointing at
```



After 'P' Pointing to some data, if that data is deleted or freed up, Even then, P Points to same address, which is invalid. So, Now P is Dangling Pointer.

Topic : 1-D Arrays

***Arrays** : Collection of Similar (or) homo-geneous data items.

- Why to group Similar data ?

Ex: // 5 Integers as Individual variables

```
int a, b, c, d, e;
```

Let 4 Bytes Per integer

(RAM)

Random Memory Allocation

```
for(i=1; i<=10000000; i++)
{
    a = b+c;
    d = a-b;
    e = d+a;
    b = b-2;
    c++;
}
```

// All these variables accessed repeatedly multiple times.

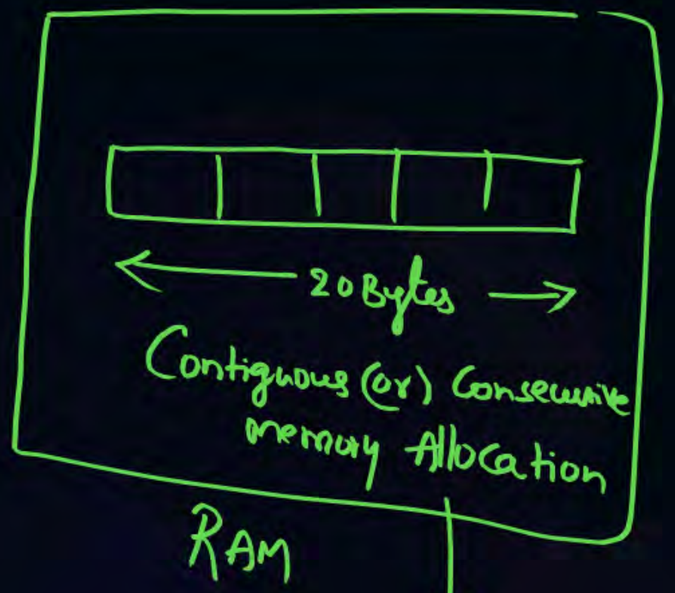
// So, Every access require time for fetching data from memory.

// Access Time will be more.

Topic : 1-D Arrays

5 Integers as Group

```
int X[5];
```



Access Time will be Less.

Topic : 1-D Arrays

Types of arrays

1) One-dimensional (1-D) array

2) Multi-dimensional array

- Two-dimensional (2-D) array
- Three-dimensional (3-D) array

⋮

Topic : 1-D Arrays

1-D array : It is also known as vector.

Array Declaration :

Syntax: datatype Name[size];

Name → Any valid identifier

datatype → Any Primary (or) Secondary

Ex:

```
int x[5];
char s[10];
float i[7];
struct s Var[3];
```

NOTE: 1) Arrays will be allocated memory at the time of Compilation.

2) The Memory allocated for arrays will be Fixed (or) Static.

Topic : 1-D Arrays

Initialization / Assign values

```
datatype  Name [size] = { value1, value2, ....};
```

Ex: int x[5] = {10, 20, 30, 40, 50};

(OR)

```
datatype  Name [size];
        |
        |
        |
Name[Index] = Value;
```

Ex:

```
int x[5];
   |
   |
x[0] = 10;
x[2] = 30;
x[4] = 50;
```

How to access Elements of array?

- Individual access of Elements is Possible, with unique ID called Index.
- By default, Indexing starts from Zero.

Ex: int s[6]; Let 4Bytes Per int

// Arrays support random access.

Topic : 1-D Arrays

Examples :

① int x[5] = {10, 20, 30, 40, 50, 60, 70};

Excess values are ignored.

② int x[5] = {10};

	0	1	2	3	4
X	10	0	0	0	0

Char Y[10] = '@';

	0	1	2	3	4	5	6	7	8	9
Y	@	\0	\0	\0	\0	\0	\0	\0	\0	\0

NULL Character

NOTE:
if atleast one Element is Initialized, Then
Number array ⇒ Zeroes for rest.
Character array ⇒ NULL character (\0) for rest

③ int x[5];

Printf("%d", x[1]);
Printf("%d", x[3]);
} Garbage (or) random values.

NOTE: if None is Initialized, Then random Values are assigned for Elements of array.

Topic : 1-D Arrays

Let 4 Bytes Per int, Starting address = 2000.

'f' == address of

%u = unsigned integer

Ex: int x[5] = {11, 22, 33, 44, 55};

	0	1	2	3	4
x	11	22	33	44	55

Base address → 2000 → Starting addresses

2000	2004	2008	2012	2016
2001,	2005	2009	2013	2017
2002,	2006	2010	2014	2018
2003	2007	2011	2015	2019

Addresses

- Memory is Byte-addressable (Each Byte assigned with Unique address.)
- Starting address == Address of any Content.

$$&x[0] == &x == 2000$$

- Starting address of array is called as <u>Base address</u>.

```
Printf("%u", &x[0]);  // 2000
Printf("%u", &x[2]);  // 2008
Printf("%u", &x[4]);  // 2016
```

Let 4 Bytes Per integer.

// sizeof( ) function returns memory allocated in Bytes.

Ex:

1)
```
int x[5];
Printf(" %d", sizeof(x)); // 20
```

2)
```
int x[5] = {10,20};
Printf(" %d", sizeof(x)); //20
```

3)
```
int x[ ] = {10,20};
Printf(" %d", sizeof(x)); // 8
```

NOTE: when initialized array, size is optional.
Otherwise, size is Mandatory.

Ex:
```
int x[ ]; // Error, Size must be specified.
int x[ ] = {10,20,30}; // Valid
int x[3] = {10,20,30};
```

- Arrays must be assigned values (whole), at the time of declaration only.

Ex:

```
int x[5];
x[5] = {10,20,30,40,50}; // Error.
```

```
int x[5] = {10,20,30,40,50}; // Valid
```

```
int x[5];
x[0] = 10;
x[1] = 20;
x[2] = 30;
x[3] = 40;
x[4] = 50;
```

Individual assignment is valid.

Input / output of 1-D array:

	0	1	2	3	4
x	10	20	30	40	50

```
int x[5] = {10,20,30,40,50}; // Static input
```

```
int x[5], i;
Printf(" Enter array values");
for( i=0; i<5; i++)
    Scanf(" %d", &x[i]); // Accepting Dynamic input

// Printing array Elements
for(i=0; i<5; i++)
    Printf(" x[%d] value is %d", i, x[i]);
```

O/P:

```
Enter array values 10 ↵
20 ↵
30 ↵
40 ↵
50 ↵
x[0] value is 10
x[1] value is 20
x[2] value is 30
x[3] value is 40
x[4] value is 50
```

2 mins Summary

- Null Pointer
- Dangling Pointer
- Array
  - Definition
  - Types of arrays
    - 1-D array

THANK - YOU