GLOBAL EDGE
Intelligence Of Things™

Pointers

Agenda :

- ✔ What is pointer ?
- ✔ Usage of pointers
- ✔ Pointer Arithmetic
- ✔ Pointer and One Dimensional Arrays
- ✔ Pointer to an Array
- ✔ Array of pointers
- ✔ const qualifier
- ✔ Void pointer
- ✔ Character array

What is Pointer ?

- A pointer is a address.
- A pointer is a variable which holds the address of other variable & we can apply the dereference operator on that variable.

Declaration of pointer :

```
data-type * variable_name ;
```

Initialization (or definition) of pointer :

```
int a;
int *p = &a;
```

Assigning the address to a pointer variable:

```
int a;
int *p;
p = &a;
```

Example:

```
int main (void)
{
        int x = 10;
        int *p = NULL;
        p = &x;

        printf (“*p = %d, x = %d \n”, *p, x);

        *p = 15;

        printf (“*p = %d , x = %d \n”, *p, x);

        return 0;
}
```

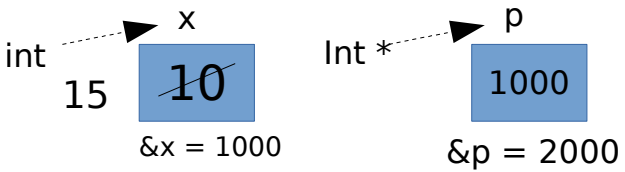


Fig 1 : Integer pointer

Usage of pointers :

- Accessing array elements.
- Returning more than one value from a function.
- Accessing dynamically allocated memory.
- Implementing data structures like linked lists, trees.

Pointer Arithmetic :

- Fetching (or accessing) the data through pointer, inserting the data using pointer & increment/decrement operation on pointer depends on the size of pointer data type.

- Size of pointer for any data type is 4 bytes for 32bit compiler & 8 bytes for 64 bit compiler .

```
Ex:   int a = 5, *pi = &a;        //&a = 1000
      char c = 'x', *pc = &c;     //&c = 5000

      pi++;                       //pi = 1000 + 1 * 4 = 1004    (since int is 4 bytes)
      pi = pi – 3;                //pi = 1004 – 3 * 4 = 992
      pi = pi + 5;                //pi = 992 + 5 * 4 = 1012
      pi--; (or)      --pi;       //pi = 1012 – 1 * 4 = 1008
      pc++; (or) ++pc;            //pc = 5000 + 1 * 1 = 5001    (since char is 1 byte)
      pc = pc – 3;                //pc = 5001 – 3 * 1 = 4998
      pc = pc + 5;                //pc = 4998 + 5 * 1 = 5003
      pc--;                       //pc = 5003 – 1 * 1 = 5002
```

Ex 1:

```
int main (void)
{
    char buf[64];
    int *n = NULL:
    buf[40] = '2';
    n = (int *)buf;
    printf ("%c \n", n[10]);
    return 0;
}
```

Ex 2:

```
int main(void)
{
        int x = 10, y = 20;
        int *p = &x;
        int *q = &y;
        int *ptr = p + q;

        printf (“%p \n”, ptr);
        return 0;

}
```

Ex 3:

```
int main (void)
{
        int a = 320;
        char *ptr = (char *)&a;
        printf ("%d \n", *ptr);

        return 0;
}
```

(320)d = 00000000 00000000 00000001 01000000

Pointer comparisons:

- Relational operators (==, !=, <, <=, >, >=) can be used with pointers

1) ==, !=

- If both pointers are NULL (or) both contain address of same variable.
- Between void pointer & other pointer.

2) >, < , <= , >=

- Valid between pointers of same type.
- Both pointers point to elements of arrays of same data type.

```
int main(void)
{
    int *i = NULL;
    float *f = NULL;
    char *c = NULL;
    void *v;

    int a = 10;

    v = &a;
    c = &a;
    f = &a;

    if (f == c)  //true - but with warnings
        printf ("true \n");
    else
        printf ("false \n");
    return 0;
 }
```

```
    if (v  !=  i)    //true - but with warnings

        printf ("false \n");

    else
        printf ("true \n");


if (v >= c){
        printf ("%d, %d", *(int*)v, *c);

        printf ("true \n");
    }else
        printf ("false \n");
```

Pointer to pointer:

```
int a = 5;
int *pa = &a;
int **ppa = &pa;
```

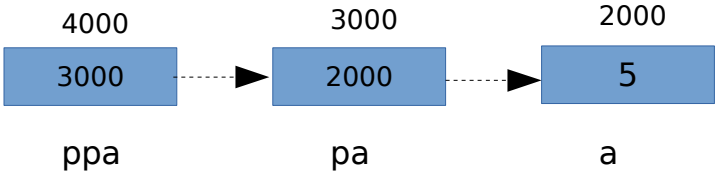


Fig 2 : Visualization of pointer to pointer

Table 1 : Illustration of pointer to pointer

Value of a	a	*pa	**ppa	5
Addresss of a	&a	pa	*ppa	2000
Value of pa	&a	pa	*ppa	2000
Address of pa		&pa	ppa	3000
Value of ppa		&pa	ppa	3000
Address of ppa			&ppa	400

Pointer & One dimensional Arrays:

- Three main points.

```
int arr[5] = {1, 2, 3, 4, 5};
int *p;
p = arr;
```

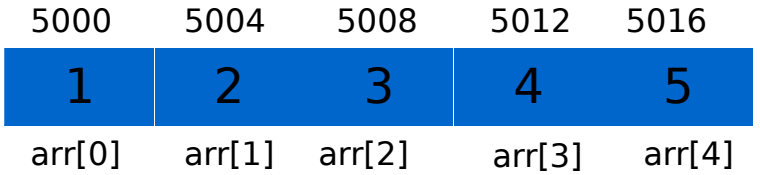


Fig 3 : Array of 5 elements

p[i] ----> *(p + i) ----> value at i^{th} position

&p[i] ----> (p + i) ----> address at i^{th} position

- a[i] = *(a + i) = *(i + a) = i[a]
- &a[i] = &*(a + i) = (i + a) = &i[a]

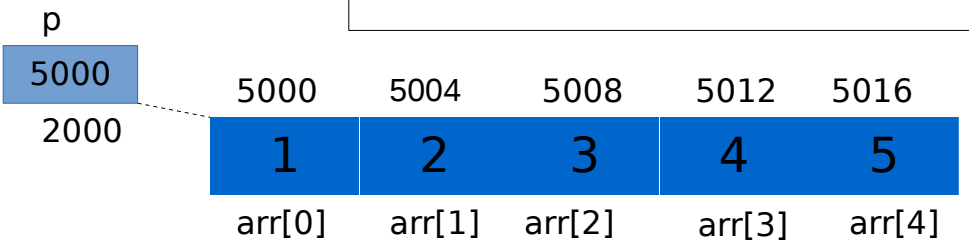


Fig 4 : Assigning the address of an array to pointer

Pointer to an Array :

int (*ptr)[5];

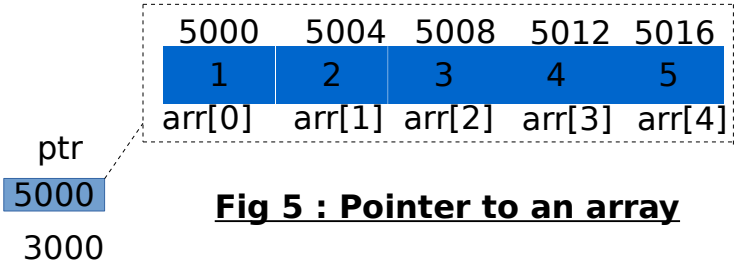


Fig 5 : Pointer to an array

Array of Pointers :

int *a[ ] = {arr, arr + 1, arr + 2, arr + 3, arr + 4};

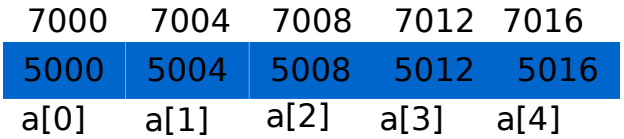


Fig 6: Array of Pointers

Example :

```
int main(void)
{
   int arr[ ] = {1, 2, 3, 4, 5};    //initialization of array

   int *p = arr;                    //assigning address of array to pointer

   int (*ptr)[5] = &arr;                    //pointer to an array

   int *a[ ] = {arr, arr + 1, arr + 2, arr + 3, arr + 4};//array of pointers

   printf ("arr = %p, p = %p, ptr = %p,a[0] = %p \n", arr, p, ptr, a[0]);
   printf ("*a[0] = %d \n", *a[0]);

   p++;
   ptr++;

   printf ("arr + 1 = %p, p = %p, ptr = %p,a[1] = %p \n", arr + 1, p, ptr, a[1]);

   return 0;
}
```

Const Qualifier:

There are three types of declarations of pointers using the qualifier const :

1. Pointer to constant data.
2. Constant pointer.
3. Constant pointer to constant data.

Ex 1 :
```
const int a = 2, b = 6;
const int *p1 = &a;

*p1 = 9;     //invalid
p1 = &b;     //valid
```

Ex 2 :
```
int a = 2, b = 6;
int *const p2 = &a;

*p2 = 9;     //valid
p = &b;      //invalid
```

Ex 3 :
```
const int a = 2, b = 6;
const int * const p3 = &a;

*p3 = 9;     //invalid
p3 = &b;     //invalid
```

Void Pointer :

- Generic pointer.
- Before de-referencing, it should be type cast to proper data type.

Ex:

```
int main(void)
{
   int a = 3;
   float b =3.4;
   void *vp;
   vp = &a;
   printf ("value of a = %d \n", *(int *)vp);
   *(int *)vp = 12;
   printf ("value of a = %d \n", *(int *)vp);
   vp = &b;
   printf ("value of b = %f \n", *(float *)vp);
   return 0;
}
```

Character Array:

```
char buf [8] = {'a', 'b', 'c', 'd', 'e'};
char arr[ ] = “array”;
char *str = “string”;
```

Ex :
```
int main (void)
{
        char buf [8] = {'a', 'b', 'c', 'd', 'e'};
        char arr[ ] = “array”;
        char *str = “string”;

        buf[3] = ‘z’;
        arr[3] = ‘y’;
        str[3] = ‘x’;          //invalid

        str = “hello”;                 //valid
        arr[ ] = “world”;   //invalid

        return 0;
}
```

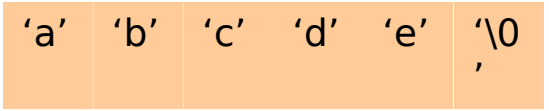


Fig 7 : character Array

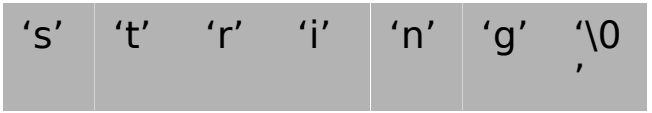


Fig 8 : String

Table 2: Difference between Character constant & String constant

Character constant	String constant
A character constant is enclosed within single inverted commas.	A sequence of characters enclosed in double quotes.
The maximum length of a character constant can be one character.	A string constant can be any length.
A single character constant has an equivalent integer value.	A single string constant does not have an equivalent integer value.
The character constant ‘A’ consists of only character A.	The string constant "A" consists of character A and \0.
A single character constant occupies one byte.	A single string constant occupies two bytes.
Every character constant does not end up with a NULL character.	Every string constant ends up with a NULL character which is automatically assigned (before the closing double quotation mark) by the compiler.

Any queries.....?

Output.. ?

1) printf (“%d %d \n”, sizeof(‘A’), sizeof(“A”));

2)int main (void)
```
{
    char str[50] = “global edge”;
    printf (“str = %s \n”, str);
    printf (“str = %s \n” + 1, str);
    printf (“str = %s \n”, str + 1);

    printf (“%d \n” + 1, 123);
    return 0;
}
```

3) int main (void)
```
{
    int arr[ 5 ];

    printf (“%d \n”, ((arr + 2) – (arr + 1));
    return 0;
}
```

4) int main (void)
```
{
    Int *p = 10;
    printf (“%u \n”, (unsigned)p);

    return 0;
}
```

5) int main (void)
```
{
  int arr[ ] = {95, 12.....,20,30};

  /* your code to print array elements
    without using sizeof operator*/

  return 0;
}
```

Large enough to Deliver, Small enough to Care

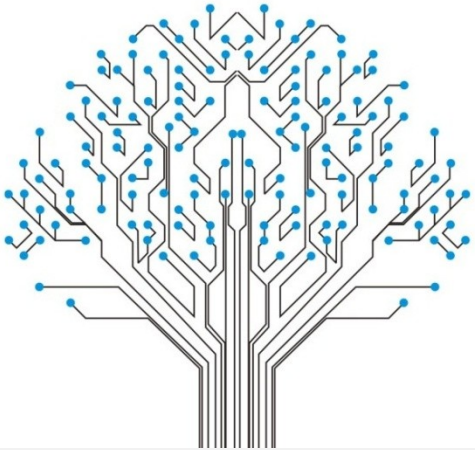
Global Village
IT SEZ
Bangalore

South Main Street
Milpitas
California

Raheja Mindspace
IT Park
Hyderabad

www.globaledgesoft.com

Thank you



Fairness
Learning
Responsibility
Innovation
Respect