BITWISE AND (&)
BITWISE OR (|)
BITWISE XOR (^)

If both are same → 0
If both are diff → 1

		Result
0	0	0
0	1	1
1	0	1
1	1	0

ONE'S COMPLEMENT

Invert the number

operand 1	Result
0	1
1	0

Note: First bit represents the sign

eg 0 1 0 1 = 5

1 1 0 1 = −3

−8 + 4 + 1 = −3

unsigned → ignores the sign convection

BITWISE LEFT SHIFT (<<)

Shifts the digits to the left

eg int a = 8 << 1 ; = 16

8 = 1000 (binary) => 10000 = 16

eg 5 << 3 ; = 40

5 = 0101 => 0101000 = 40

BITWISE RIGHT SHIFT (>>)

5 >> 2 ; = 1

5 = 0101 => 0001 = 1

we remove the last position

Note → alternate to check if divisible by 2 also it is slightly faster

if (a & 1) = if (a % 2 == 1)

2D ARRAY

We know that when we initialise 1D array, we need not specify size. But for 2D array we must always specify second dimension even if you are specifying elements during declaration

int arr [][3] = {{1,3,0},{-1,5,9}};

STRINGS

eg

```c
#include<stdio.h>
int main() {
    char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};

    printf("%s\n", str);
}
```

end character

no &

Different string declarations

```c
#include<stdio.h>

int main() {
    char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};

    char str1[] = "Hello again";

    printf("%s, %s\n", str, str1);

    for(int i = 0; i < 11; i++)
        printf("%c", str1[i]);
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
➜ class gcc string.c
➜ class ./a.out
Hello, Hello again
Hello again%
➜ class
```

Note int a[] = {1,2,3} , b, c;

int[] a, b, c;

both are arrays

has integer data type

Sr.No.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

POINTERS

'&' operator is the address of operator

It cannot be used with a constant or expression

Eg

int a = 5	&a	Valid
float f = 8.6	&f	Valid
Constant	&26	Invalid
Expression	&(a+f)	Invalid

```
int a = 5;
printf("%ld\n", &a);
```

Output → format ld but expects arg type 'long int' but argument has type 'int *'

```
int a = 5;
printf("%ld\n", (long int) &a);
```

output → 147865231515 (adress)

Pointer Variable → syntax for declaration of pointer variable

datatype *p_name

* is used to dereference to pointer

- In the program attached,
  - ptr can be used to access the address of variable a and *ptr can be used to access its value
  - Writing *(&a) and a is same

```
int main()
{
    int a = 5;
    int *ptr = &a; // Address of a assigned to a pointer variab
    printf("Value of ptr : %u\n", ptr);
    /*
     Pointer variable can be copied and then
     both the pointers point to same address
    */
    int *ptr2 = ptr;
    printf("Value of ptr2 : %u\n", ptr2);
    //Dereferencing operator
    printf("Value of data at address ptr is: %d\n", *ptr);
    return 0;
```

```
int a = 5;
int *b = &a;   (value of b is address of a)
printf("%ld\n", (long int) b);
```

Output → 140760053210065 43

CONT

```
printf("%d\n", *b);
```

Output → 5

Writing *(&a) is same as a

- Subtraction of two pointer variables of same base type returns the number of values present between them

Eg - int *ptr1 = 2000, *ptr2 = 2020;

printf("%u\n", ptr2-ptr1); Output : 5

COMBINATION OF DEFERENCE & INCREMENT/DECREMENT

deference (*), adress (&) and increment / decrement have same precedence and are RIGHT to LEFT associative

Expression		Evaluation	
x = *ptr++	→ ++ is after so post increment	x = *ptr	ptr = ptr + 1
x = *++ptr		ptr = ptr + 1	x = *ptr
x = (*ptr)++		x = *ptr	*ptr = *ptr + 1
x = ++*ptr		*ptr = *ptr + 1	x = *ptr

POINTER TO POINTER

pointer to pointer variable is used to store address of pointer variable

Syntax

```
datatype **pp_name;
```

```c
int main()
{
    int a = 5;          // Integer variable
    int *ptr = &a;      // Pointer to int
    int **pptr = &ptr;  // Pointer to pointer to int
    printf("Address of a   : %u\n", &a);
    printf("Value of ptr   : %u\n", ptr);
    printf("Address of ptr : %u\n", &ptr);
    printf("Value of pptr  : %u\n", pptr);
    printf("Address of pptr: %u\n", &pptr);
    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

$ gcc -w test.c
$ ./a.out
Address of a   : 3882647076
Value of ptr   : 3882647076
Address of ptr : 3882647080
Value of pptr  : 3882647080
Address of pptr: 3882647088
$
```

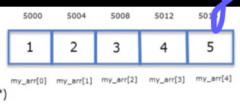
eg
```
int b = 10;
int *p = &b;
int **q = &p;
printf("Address of b = %u\n", &b)
printf("Value of pointer p = %u", p)
printf("Address of pointer p = %u", &p)
printf("Value of pointer p = %u", q)
printf("Address of pointer p = %u", &q)
```

```
ankit@LAPTOP-69EE82MG:/mnt/c/Users/sonal/De
Address of b = 1243951652
Value of pointer p = 1243951652
Address of pointer p = 1243951656
Value of pointer q = 1243951656
Address of pointer q = 1243951664
ankit@LAPTOP-69EE82MG:/mnt/c/Users/sonal/De
```

POINTER w/t 1D Array

Consider an array

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



Here arr is a pointer to the first element aka arr is a pointer to int or (int*)

Remember
arr = &arr[0]
arr + 1 = &arr[1]
arr + 2 = &arr[2]
arr + 3 = &arr[3]

Thus
*(arr) = arr[0]
*(arr + 1) = arr[1]
*(arr + 2) = arr[2]
*(arr + 3) = arr[3]

```
int *p;
int arr[] = {11, 22, 33, 44, 55};
p = arr;
```

We **can do** p++, p--
but we **can not do** arr++, arr--

POINTER & FUNCTIONS

1) Call by Value

```
eg void swap(int x, int y){
    int t;
    t = x;
    x = y;
    y = t;
    printf("x=%d y=%d", x, y);
}
```

2) Call by reference

```
eg void swap(int* x, int *y){
    int t;
    t = *x;
    *x = *y
    *y = t;
    printf("x=%d y=%d", *x, *y);
}
```