

BITWISE AND (&)

BITWISE OR (|)

BITWISE XOR (^)

If both are same \rightarrow 0

If both are diff \rightarrow 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 $0\ 101 = 5$

$\boxed{1} \ 101 = -3$
 $\downarrow \quad \downarrow \quad \downarrow$
 $-8 + 4 + 1 = -3$

unsigned \rightarrow ignores the sign connection

BITWISE LEFT SHIFT (<<)

shifts the digits to the left

eg $\text{int } a = 8 << 1; = 16$

$8 = 1000 \Rightarrow 10000 = 16$
 (binary) $=$

eg $5 << 3; = 40$

$5 = 0101 \Rightarrow 0101000 = 40$

BITWISE RIGHT SHIFT (>>)

$5 >> 2; = 1$

$5 = 0101 \Rightarrow \underline{\underline{000101}} = 1$

we remove the last position

Note \rightarrow alternate to check if divisible by 2
 also it is slightly faster
 $\text{if } (a \& 1) = \text{if } (a \% 2 == 1)$

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

$\text{int arr [][3] = \{[1,3,0],[1,5,9]\};}$

STRINGS

```
#include<stdio.h>
eg
int main() {
    char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};
    printf("%s\n", str);
}
  ↗ no &
```

↗ end character

Different string declarations

```
1 #include<stdio.h>
2
3 int main() {
4     char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};
5
6     char str1[] = "Hello again";
7
8     printf("%s, %s\n", str, str1);
9
10    for(int i = 0; i < 11; i++)
11        printf("%c", str1[i]);    i
12    }
13
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
→ class gcc string.c
→ class ./a.out
Hello, Hello again
Hello again
→ class |
```

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

$\text{int } [] a, b, c;$ $\backslash \backslash$

$\backslash \backslash$
 both are arrays

has integer data type

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

<code>int a = 5</code>	<code>&a</code>	Valid
<code>float f = 8.6</code>	<code>&f</code>	Valid
<code>Constant</code>	<code>&26</code>	Invalid
<code>Expression</code>	<code>&(a+f)</code>	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
      ↪ address
```

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 variat
    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 *b = &a; (value of b is address of print ("%ld\n", (long int) b));
 Output → 14076005321006543
 ↪ 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 (*), address (&) and increment/decrement have same precedence and are **RIGHT TO LEFT** associative

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

POINTER TO POINTER & FUNCTIONS

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

datatype *pp-name;**

```

3 int main()
4 {
5     int a = 5;           // Integer variable
6     int *ptr;           // Pointer to int
7     int **ptrt;         // Pointer to pointer to int
8     printf("Address of a : %u\n", &a);
9     printf("Value of ptr : %u\n", ptr);
10    printf("Address of ptr : %u\n", &ptr);
11    printf("Value of ptrt : %u\n", ptrt);
12    printf("Address of ptrt: %u\n", &ptrt);
13
14 }
15

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

$ gcc -w test.c
$ ./a.out
Address of a : 3882647876
Value of ptr : 3882647876
Address of ptr : 3882647880
Value of ptrt : 3882647880
Address of ptrt: 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/Desktop/pointer/pointer_to_pointer
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/Desktop/pointer/pointer_to_pointer

```

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]



We can do p++, p--
but we can't 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("%d %d", x, y);
}

```

2) Call by reference

eg void swap (int *x, int *y){

```

int t;
t = *x;
*x = *y;
*y = t;
printf("%d %d", *x, *y);
}

```

int arr[10]; *(statically allocated array)*
size allocated in compile time

DYNAMICALLY ALLOCATED MEMORY

(size is allocated in run time)

Void Pointer

It is not associated with any datatype
can point to any data type.

```

1 int main(){
2     int a = 7;
3     float b=7.6;
4     void *p;
5     p=&a;
6     printf("Integer variable is= %d",*((int*)p));
7     printf("\nP is= %p",p);
8     p=&b;
9     printf("\nFloat variable is= %f",*((float*)p));
10    printf("\nP is= %p",p);
11 }
12
13 Output
14 Integer variable is= 7
15 P is= 0x7ffdfff9e0f28
16 Float variable is= 7.600000
17 P is= 0x7ffdfff9e0f2c

```

type casting

MALLOC CALLOC

malloc returns a void address

Malloc, Calloc, Realloc →

#include <stdlib.h>

Difference between Malloc & Calloc

We pass byte size in malloc

In calloc we pass elements and how many bytes are present in each element is passed

CALLOC → calloc (no.of-items, size);

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int *ar;
    ar =(int *)calloc(10,sizeof(int));

    for(int i=0;i<12;i++){
        ar[i]=i;
        printf("%d ",ar[i]);
    }
    printf("\n");
}
```

OUTPUT

0 1 2 3 4 5 6 7 8 9 10 11

if we keep on writing extra block just happens to be written on top of old elements.

REALLOC

Reallocates the given area of memory. Must be previously allocated by malloc(), calloc() or realloc()

realloc (void *ptr, size)

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int *ar;
    ar =(int *)calloc(10,sizeof(int));
    printf("p: %p",ar);
    for(int i=0;i<10;i++){
        ar[i]=i;
        printf("%d ",ar[i]);
    }
    printf("\n");
    ar=(int *)realloc(ar,10* sizeof(int));
    printf("p: %p",ar);
    for(int i=0;i<20;i++){

        printf("%d ",ar[i]);
    }
    printf("\n");
}

OUTPUT
p: 0x5620a67ff2a00 1 2 3 4 5 6 7 8 9
p: 0x5620a67ff2a00 1 2 3 4 5 6 7 8 9 1041 0 807418480
842413432 926310704 1630692966 824193072 857748000
891302944 924857888
```

FREE

deallocates the memory allocated to malloc, realloc etc

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int *ar;
    ar =(int *)calloc(10,sizeof(int));
    printf("p: %p",ar);
    for(int i=0;i<10;i++){
        ar[i]=i;
        printf("%d ",ar[i]);
    }
    printf("\n");
    free(ar);
    printf("p: %p\n",ar);
    printf("%d ",ar[1]);
    printf("\n");
}
```

OUTPUT

p: 0x56171a7612a00 1 2 3 4 5 6 7 8 9
p: 0x56171a7612a0

0 → dangling pointer

Memory leak - when programmer fails to free an allotted block of memory when no longer needed

after using free() it is customary to use NULL

eg free(ar);
ar = NULL;

used to protect against dangling pointer bugs.

If we are unable to get address it is a type of memory leak

if we use malloc() after malloc() for the same pointer. We wont have address of 1st malloc, so it is a type of Memory leak

If we use realloc after malloc, realloc will AUTOMATICALLY FREE memory

STRUCTURE

user defined datatype in C that allows to combine data items of diff types

Syntax:

```
struct [structure name]
{
    member definition;
    member definition;
    ...
    member definition;
};
```

```
struct struct_example
{
    int integer;
    float decimal;
    char name[20];
};

Creating an object:
struct struct_example s={10,10.0,"abcdef"};

Access(read/write):
s.integer
s.decimal
s.name
```

```
struct Student{
    int roll;
    int age;
    char name[100];
};

int main(){
    struct Student eleena = {100, 11, "Eleena"};
    printf("name is : %s",eleena.name);
}

OUTPUT
name is : Eleena
```

```
struct Student eleena;
eleena.roll=20184045;
eleena.age=21;
//eleena.name="Eleena";      doesnt work
strcpy(eleena.name, "eleena");
```

```
struct Student array[10];
array[0].roll = 20184006;
array[0].age 22;
strcpy(array[0].name, "Naeraj");
```

variable . name is used
pointer -> name (for pointer ->)

eg p->roll = 2020;

struct in struct

```
struct Name{
    char firstName[50];
    char lastName[50];
};

struct Student{
    int roll;
    int age;
    struct Name n;
};

eleena.n.firstName
```

union

```
struct {
    name;
    roll;
    name;
} name, roll, name
```

UNION

A union is a special data type available in C that allows storing different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time.

Syntax:

```
union [union name]
{
    member definition;
    member definition;
    ...
    member definition;
};

Creating an object:
union union_example u;

Access(read/write):
u.integer
u.decimal
u.name
```

```
union Studentunion a;
a.roll=2020;
a.age=78;
strcpy(a.name,"Eleena");
printf("roll number is %i age is %i name is %s"
,a.roll,a.age,a.name);
}

OUTPUT
roll number is 1701145669 age is 1701145669
name is Eleena
```

examples for union in javatpoint.com/c-union

STORAGE CLASSES

Storage classes are used to describe the features of a variable / function like SCOPE, VISIBILITY & LIFE-TIME

4 uses of storage classes

Storage Specifier	Storage	Initial Value	Scope	Life
auto	Stack	Garbage	within block	end of block
extern	Data Segment	0	Global multiple files	till end of program
Static	Data Segment	0	within block	till end of program
Register	CPU register	Garbage	within block	end of block

#DEFINE

The #define preprocessor directive is used to define constant or macro substitution

A macro is a fragment of code that is given a name. You can define a macro in C using #define preprocessor directive

eg #define C 299792458

here when we use C in the program it is replaced with 299...

TYPE DEF

typedef is a keyword used to assign alternative names to existing datatype. Eg when names of datatype becomes too complicated to use in programs.

SYNTAX

typedef < existing-name > < alias-name >

eg typedef unsigned long ulong;

It can also be used to give a name to a user defined data type as well.

eg typedef struct Elektra

POINTER ARITHMETIC

1. Increment & Decrement

eg ptr1 = &N;
printf("%p", ptr1);
ptr1++;
printf("%p", ptr1);

OUTPUT : 0x950 0x954 (4 is size of int)

eg ptr1 = &N;
printf("%p", ptr1),
ptr1--;
printf("%p", ptr1);

OUTPUT 0x956 0x952
(4 is size of int)

ADDITION

when a pointer is added with a value then

pointer = pointer + (value * size of data)

ptr2 = 0x770
eg ptr2 = ptr2 + 3

print(ptr2)

output = 0x776

)

SUBTRACTION

eg $\text{ptr2} = 0 \times 770$
 $\text{ptr2} = \text{ptr2} - 3$
 $\text{print}(\text{ptr2})$
 output = 0×764

SUBTRACTION OF 2 POINTERS

possible only when pointers are of
same type

eg $\text{ptr1} = \& N$; $\text{ptr2} = \& N$;
 $\text{ptr2} += 3$;
 $\text{int } x = \text{ptr2} - \text{ptr1}$;
 $\text{print}(x)$
 OUTPUT = 3

