

## C programming and Data Structures

Kernighan and Ritchie 2<sup>nd</sup> edition, Neso academy, Code Vaults and  
MIT Advanced Data structure and algorithms.

```
/**  
List of topics:  
1. Variables  
2. Type casting a variable  
3. Scoping:  
3.1 Auto Modifier  
3.2 Extern Modifier  
3.3 Register Modifier  
3.4 Static modifier  
3.5 Volatile Modifier  
3.6 Header File  
4. Const and #define  
5. Random Number Generation  
6. Scanf and Printf using Strings  
7. Operators:  
7.1 Mathematical operators  
7.2 Logical Operators  
7.3 Bitwise operators  
8. Conditional Statements  
9. Break and Continue  
10. Static scoping vs Dynamic scoping  
12 Parallelism:  
12.1 Processes  
12.2 Threading  
13 Race Condition:  
13.1 Mutex  
13.2 Semaphores  
14. Recursion  
15. Arrays 1-D  
16. Arrays 2-D and 3-D  
16. Pointers  
17. Dynamic memory allocation  
18. Data Structure:  
18.1 Stack  
18.2 Circular Queue  
18.3 Link List  
18.4 Doubly Link list  
**/
```

**Author:** Harsh Dubey

## Chapter 1: Variables

### **Variable names:**

**Variable:** Variable in C is something whose value can change during runtime/execution.

**Declaration:** Rules for declaring the variables.

1. **First character must be a letter.**
2. **Underscore** counts as a letter.
3. **Upper and lower** are different names. X and x are different names.

At least the first 31 characters of an internal name are significant. For function names and external variables, the number may be less than 31, because external names may be used by assemblers and loaders over which the language has no control. For external names, the standard guarantees uniqueness only for 6 characters and a single case. Keywords like `if`, `else`, `int`, `float`, etc., are reserved: you can't use them as variable names. They must be in lower case.

"Internal names" are names of identifiers within a function (effectively local variable names).

"External names" would be the names of the other identifiers, including the names of functions and any identifiers declared at global scope or declared with storage class `extern`.

Basically, anything that needs to be "externally visible" is only guaranteed to have 6 (non case sensitive) unique characters, which is extremely limiting.

In practice, this is no longer an issue. C99 increased these limits, and most modern compilers do away or significantly increase these limits. For example, [Visual C++ allows 247 characters for uniqueness](#) in all identifiers (internal or external) when compiling C.

### **Data type, size qualifier and sign qualifier:**

1. **Four kinds of data types: Char, int, float, double**
  - **Char:** character whose size is **1 byte**.
  - **Integer:** + or - whole numbers. Size is **4 bytes**.
  - **Float:** Single precision floating point numbers or fractions (**7-digit precision**). Size is **4 bytes**.
  - **Double:** Double precision floating point numbers (**16-digit precision**). Size is **8 bytes**.
2. **Qualifiers based on size:** Change size of data types.
  - **Short and long** apply to **integers**. Typically, `int` is 4 bytes, `short int` is 2 bytes and `long int` is 8 bytes. You can **omit int** keyword, and the default is `int`.
  - **Long** can also be used for **double**.
3. **Qualifiers based on sign:**
  - **Signed and unsigned** can be applied to any characters or integers. Will change the range of the data type.
4. `<limits.h>` and `<float.h>` contain symbolic constant.

CHAR_BIT	8	Defines the number of bits in a byte.
SCHAR_MIN	-128	Defines the minimum value for a signed char.
SCHAR_MAX	+127	Defines the maximum value for a signed char.
UCHAR_MAX	255	Defines the maximum value for an unsigned char.
CHAR_MIN	-128	Defines the minimum value for type char and its value will be equal to SCHAR_MIN if char represents negative values, otherwise zero.
CHAR_MAX	+127	Defines the value for type char and its value will be equal to SCHAR_MAX if char represents negative values, otherwise UCHAR_MAX.
MB_LEN_MAX	16	Defines the maximum number of bytes in a multi-byte character.
SHRT_MIN	-32768	Defines the minimum value for a short int.
SHRT_MAX	+32767	Defines the maximum value for a short int.
USHRT_MAX	65535	Defines the maximum value for an unsigned short int.
INT_MIN	-2147483648	Defines the minimum value for an int.
INT_MAX	+2147483647	Defines the maximum value for an int.
UINT_MAX	4294967295	Defines the maximum value for an unsigned int.
LONG_MIN	-9223372036854775808	Defines the minimum value for a long int.
LONG_MAX	+9223372036854775807	Defines the maximum value for a long int.
ULLONG_MAX	18446744073709551615	Defines the maximum value for an unsigned long int.

## 5. Question: What kind of printing qualifier you should use for all these ranges?

### Question: 2.1:

**Exercise 2-1.** Write a program to determine the ranges of `char`, `short`, `int`, and `long` variables, both `signed` and `unsigned`, by printing appropriate values from standard headers and by direct computation. Harder if you compute them: determine the ranges of the various floating-point types.

```
#include <stdio.h>
#include <limits.h>
#include <float.h>

// To execute C, please define "int main()"

int main() {

    printf("Char: %d\n", sizeof(char));
    printf("Int : %d\n", sizeof(int));
    printf("float : %d\n", sizeof(float));
    printf("Double : %d\n", sizeof(double));

    printf("Size of Int : %d \n", INT_MIN);
    printf("Size of Int : %d \n", INT_MAX);

    printf("Size of char : %d \n", CHAR_MIN);
    printf("Size of char : %d \n", CHAR_MAX);

    return 0;
}
```

## Integer:

1. Store's integers (+ and -)
2. Default is signed – “int x”
3. Find size using “`sizeof`” operator.

Use “`sizeof`” operator

```
#include <stdio.h>
int main()
{
    printf("%d", sizeof(int));
    return 0;
}
```

Note: `sizeof` is a unary operator and not a function.

Output:

4

Sizeof integer is 4 bytes in my machine. May be it is 2 bytes in your machine.

Note- `sizeof` is a unary operator

4. Find range using `<limits.h>`.

- Unsigned range is 0 to  $(2^n - 1)$ .

### RANGE OF INTEGER



2 bytes [16 bits]

Unsigned range: 0 to 65535 (by applying:  $2^n - 1$ )

Signed range: -32768 to +32767

- Signed range is  $(-2^{n-1})$  to  $(2^{n-1}-1)$ .

2's complement range:  $-(2^{n-1})$  to  $+(2^{n-1} - 1)$

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

int main()
{
    int var1 = INT_MIN;
    int var2 = INT_MAX;

    printf("range of signed integer is from: %d to %d", var1, var2);
    return 0;
}
```

Output:

range of signed integer is from: -2147483648 to 2147483647

I have used symbolic constant

- Print using %d for **signed** and for **unsigned** print using %u.
- If printed using %d then only signed representation can be printed doesn't matter if you've assigned it as unsigned or not.

```
unsigned int c = 2147483649;

printf(" Integer : %d \n", c);
```

Integer : -2147483647

- Overflow:** Circle of integer repeats when values go beyond the range.

## SUMMARY

- sizeof (short) <= sizeof (int) <= sizeof (long).
- Writing **signed int some\_variable\_name;** is equivalent to writing **int some\_variable\_name;**
- %d is used to print "signed integer"
- %u is used to print "unsigned integer"
- %ld is used to print "long integer" equivalent to "signed long integer"
- %lu is used to print "unsigned long integer"
- %lld is used to print "long long integer"
- %llu is used to print "unsigned long long integer"

## Character:

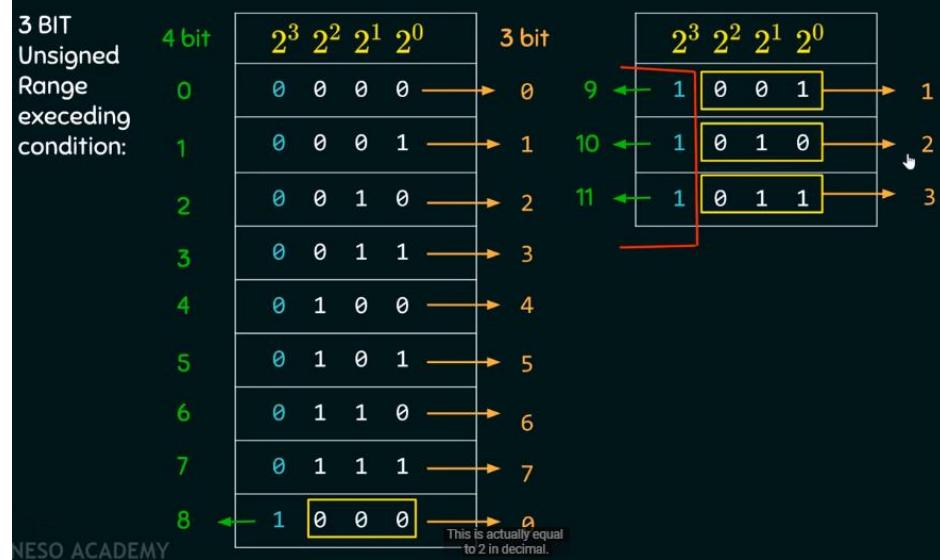
**Character:** Every character has a binary representation in machine.



The diagram shows the word "HELLO!" above a row of binary digits: 01001000 01100101 01101100 01101100 01101111 00100001. Red arrows point from each letter to its corresponding binary value below.

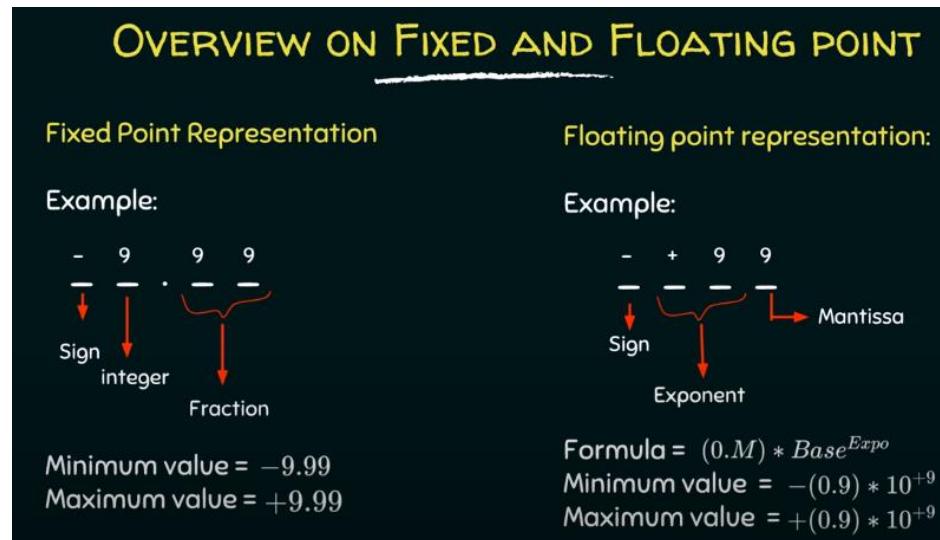
4	2	(PART OF TEXT)	34	22	66	42	96	62	6
3	3	[END OF TEXT]	35	23	#	67	43	99	63
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	100	64
5	5	[ENQUIRY]	37	25	%	69	45	101	65
6	6	[ACKNOWLEDGE]	38	26	&	70	46	102	66
7	7	[BELL]	39	27	'	71	47	103	67
8	8	[BACKSPACE]	40	28	(	72	48	104	68
9	9	[HORIZONTAL TAB]	41	29	)	73	49	105	69
10	A	[LINE FEED]	42	2A	*	74	4A	106	6A
11	B	[VERTICAL TAB]	43	2B	+	75	4B	107	6B
12	C	[FORM FEED]	44	2C	.	76	4C	108	6C
13	D	[CARRIAGE RETURN]	45	2D	,	77	4D	109	6D
14	E	[SHIFT OUT]	46	2E	.	78	4E	110	6E
15	F	[SHIFT IN]	47	2F	/	79	4F	111	6F
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	112	70
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	113	71
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	114	72
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	115	73
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	116	74
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	117	75
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	118	76
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	119	77
24	18	[CANCEL]	56	38	8	88	58	120	78
25	19	[END OF MEDIUM]	57	39	9	89	59	121	79
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	122	7A

- Character are ASCII encoded; everything must have a number. Hence, even **nonprintable character has to be encoded**, remember printing space on screen is what you used to clear the LCD in PIC 18. ASCII uses 7 bits.
- Difference between **unsigned char** and **signed char**



### Float, Double and long double:

- Float: 4 bytes;** use `%f` to print it; by default `%f` and `lf` will print 7 digits. `.16f` will print it upto 16 decimal places; however, float can only print upto 7 digits starting from before the decimal point.



- Double: 8 bytes;** use `%lf` to print; double can print upto **16 so higher precision**
- Long double 12 bytes;** use `%Lf` to print; capital L; long double can print upto **19**.
- Why use Double instead of float:** Size and precision
- To round up or round down use. f, nothing printed after the decimal.**

```

#include <stdio.h>
int main()
{
    int var = 4/9;
    printf("%d\n", var);

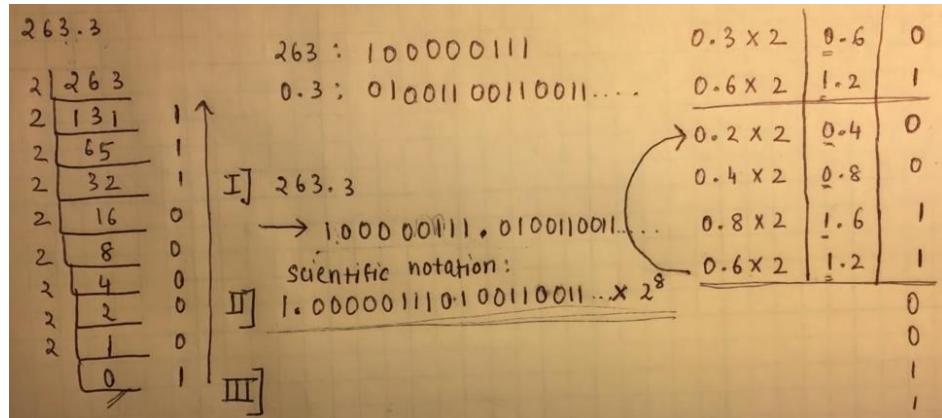
    float var1 = 4/9;
    printf("%.2f\n", var1); // Click here

    float var2 = 4.0/9.0;
    printf("%.2f\n", var2);
}

```

0  
0.00  
0.44  
Process returned 0 (0x0) execution time : 0.282 s  
Press any key to continue.

6. For int, 0 is printed there, since int will **truncate whatever is after the decimal point**.
7. For float is print 0.00 since **4 and 9 are an integer value**
8. Therefore, just put **.0 at the end**, by default it is double. To make it float just put float at the end.
9. IEEE format:



## 10.Data types:

### 11.Exercise: Reverse integer Apple interview

12. **Question:** Given a 32-bit signed integer, reverse digits of an integer. And if it's more than the signed range then return -1.

```

//Range of signed integer -2^31 to 2^31 - 1
int num;
printf("Enter a signed number: \n");
scanf("%d", &num);

//count the number of digits
int temp = num;
int count = 0;
while (temp !=0)
{
    temp = temp/10;
    count++;
}

int arr[count];
for(int i =0; i<count; i++)
{
    arr[i] = num%10;
    num = num/10;
}

//reconstructed the number
int finalnum = 0;
int c = 0;
int c_temp = count-1;

while(c <count)
{
    finalnum = finalnum+(arr[c]*pow(10,c_temp));
    c_temp--;
    c++;
}
printf("Reversed number : %d", finalnum);
}

13. }

```

```

//prototype
long int reversenum(long int );

int main()
{
    long int num, rnum;
    printf("Enter num: ");
    scanf("%d", &num);
    //reverse a number
    rnum = reversenum(num);
    printf("Reverse num : %ld", rnum);
}

long int reversenum(long int num)
{
    printf("\nGiven signed number: %ld\n", num);

    int count = 0;
    long int temp = num;

    while(temp!=0)
    {
        temp = temp/10;
        count++;
    }

    int arr[count];
    for(int i = 0; i<count; i++)
    {
        arr[i] = num%10;
        num=num/10;
    }

    long int rnum = 0;
    int count2 = count;

    for(int i = 0; i<count; i++)
    {
        rnum = rnum+(arr[i]*pow(10,count2-1));
        count2--;
    }

    if(rnum < -2147483648)
    {return 0;}
    else if(rnum > 2147483647)
    {return 0;}
    else
    {return rnum;}
}

```

14. }

15. Question: Print size of all data types and print all different data types.

## SUMMARY

1. `sizeof(short) <= sizeof(int) <= sizeof(long)`.
2. Writing `signed int some_variable_name;` is equivalent to writing `int some_variable_name;` ↴
3. `%d` is used to print “signed integer”
4. `%u` is used to print “unsigned integer”
5. `%ld` is used to print “long integer” equivalent to “signed long integer”
6. `%lu` is used to print “unsigned long integer”
7. `%lld` is used to print “long long integer”
8. `%llu` is used to print “unsigned long long integer”

16.

17.

18.

19.

20.

## 21. Questions on datatype: Neso Academy

Q1: what is the output of the following C program fragment:

```
#include <stdio.h>

int main() {
    printf("%d", printf("%s", "Hello World!"));
    return 0; ↴
}
```

22.

23. Answer: First It will print “Hello World”. Then it will print number of characters on the screen.

Q2: what is the output of the following C program fragment:

```
int main() {
    printf("%10s", "Hello");
    return 0;
}
```

24.

25. Answer: `%10s` means, print up to 10 characters wide. Therefore, 5 spaces+ Hello = 5

**Q3:** what is the output of the following C program fragment:

```
int main() {
    char c = 255;
    c = c + 10;
    printf("%d", c);
    return 0;
}
```

- a) 265
- b) Some character according to ASCII table
- c) 7
- d) 9

26.

27. **Answer:** 9

**Q4:** Which of the following statement/statements is/are correct corresponding to the definition of integer :

- |                  |                               |
|------------------|-------------------------------|
| I. signed int i; | a) Only I and V are correct   |
| II. signed i;    | b) Only I is correct          |
| III. unsigned i; | c) All are correct            |
| IV. long i;      | d) Only IV, V, VI are correct |
| V. long int i;   |                               |
| VI. long long i; |                               |

28.

29. **Answer:** All are correct

**Q5:** What does the following program fragment prints?

```
int main() {
    unsigned i = 1;
    int j = -4;
    printf("%u", i+j);
    return 0;
}
```

- a) garbage
- b) -3
- c) Integer value depends from machine to machine
- d) None of the above

30.

31. **Answer:** it will print unsigned of 2's compliment of 3 i.e. 1111.....01 = 4294967293

32. Other interview questions on Datatypes:

**What is the output of the following program?**

```
#include<stdio.h>
#include<limits.h>
void main()
{
    printf("%d", USHRT_MAX);
}
```

33.

34. **Answer:** -1

## **MCQ'S ON DATA TYPES:-**

What will be output of the following C snippet:-

1) void main()

```
{  
    char ch='E';  
    printf("%c",ch);  
}
```

- a) E    b) e    c)69    d)none

2) void main()

```
{  
    char ch='E';  
    printf("%d",ch);  
}
```

- a) E    b) e    c)69    d)none

35.

36. **Answer:** 1: E 2: 69

37.

3) void main()

```
{  
    char ch='E';  
    printf("%f",ch);  
}
```

- a) E    b) e    c)69.000000    d)none

4) void main()

```
{  
    int a=10;  
    printf("%d",a);  
}
```

- a) 10    b)97    c)a    d)none

38.

39. **Answer:** 3: none 2: 10

5) void main()  
{  
    char ch='a';  
    ch= ' ';  
    printf("%c",ch);  
}  
a) a    b) 65    c) A    d)none

6)int main()  
{  
    float f=2.900345;  
    printf("%.2f",f);  
    return 0;  
}

40. a) 2.900345    b)2.90    c)compile error    d)none

41. Answer: IMPORTANT. 5: A 6: 2.90

7.void main()  
{  
int x = 10;  
float x = 10.98;  
printf("%d", x);  
}  
a) Compilations Error    b) 10    c)10.0000    d)10.10

8. void main()  
{  
int i=70;  
char c='B';  
float f=37.243;  
i=i+c+f;  
printf("%d",i); }  
a) 173    b)173.243    c)compile error    d)none

42.  
43. Answer: 7: Compilation Error 8: 173

9. What is size of int in C ?  
a) 2 bytes  
b) 4 bytes  
c) 8 bytes  
d) Depends on the system/compiler

10. Array is \_\_\_\_\_ datatype in C Programming language.  
a)Derived Data type  
b) Primitive Data type  
c) Custom Data type  
d) None of these

44.  
45. Answer: 9: D 10: A

11. What is the value of  $1E^{+2}$  in float  
a) 100 b) 100.000000 c) 1000000 d) 10

12. What is the value of  $13123E^{-2}$  :-  
a) 131.23 b) 13123 c) 1312300 d) 131

13. State whether following declaration is valid or not  
char ch=97;  
a) Valid b) Invalid

14. void main()

```
{  
    char ch=259;  
    printf("%d",ch);  
}
```

46. a) 0 b) 3 c) 258 d) compile error

47. Answer: 13: A 14: B

48.

## Type casting a variable:

1. **What is type casting:** Type casting is better explained with an example.
  - i. **Let's analyze the behavior of data types in C:**
  - ii. Case 1: Printing a double/float using a %d operator: %d will truncate anything beyond the decimal.
  - iii. Case 2: printing a integer using %lf: 0.0000.

```
//truly random numbers  
int main()  
{  
    int a = 3, b = 2;  
  
    printf("%d", a/b);  
}
```

Harsh Dubey  
1

  - iv. **Type casting:** C thinks if int/int or int\*int will be a integer. That's why type casting.
  - v. **Type casting operator:** Type casts single operand to the right.

```
int main()  
{  
    int a = 3, b = 2;  
  
    printf("%lf", (double)a/b);  
}
```

Harsh Dubey  
1.500000

  - vi. **Rules for type casting:** Smaller into bigger or same size into same size. Can type cast bigger in smaller. But then loss of data occurs.
2. **Use of typecasting and an example:**

```
for (i = 0; i < sizeof(int); i++) {
    char byte = ((char*)&a)[i];
    for (j = 8; j >= 0; j--) {
        char bit = (byte >> j) & 1;
        printf("%hd", bit);
    }
    printf(" ");
}
printf("\n");
```

## Scope of Variable:

**Strict Definition:** a block or a region where a variable is declared, defined and used and when a block or a region ends, variable is automatically destroyed.

```
#include <stdio.h>

int main() {
    int var = 34;
    printf("%d", var);
    return 0;
}

int fun()
{
    printf("%d", var);
}
```

Scope of this variable is within main() function only. Therefore called **LOCAL** to main() function.

#include <stdio.h>	#include <stdio.h>
int main() { int var = 3; int var = 4; printf("%d\n", var); printf("%d", var); return 0; }	int main() { int var = 3; { int var = 4; printf("%d\n", var); } printf("%d", var); return 0; }
error: redefinition of 'var'	"C:\Users\jaspr\Down 4 3

1. Block of code is not just separate functions but also can be created by using **curly brackets**.
2. **Global Variable:** Is also called external variable.

```

#include <stdio.h>
int fun();

int var = 10;

int main() {
    int var = 3;
    printf("%d\n", var);
    fun();
    return 0;
}

int fun()
{
    printf("%d", var);
}

```

This variable is outside of all functions.  
Therefore called a **GLOBAL** variable

Output: 3

It will access the GLOBAL variable.

## Modifier:

**Auto Modifier:** Automatically destroyed.

1. Every variable is a auto variable by default.

### WHAT IS AUTO MODIFIER?

Auto means Automatic

Variables declared inside a scope by default are automatic variables.

Syntax: auto int some\_variable\_name;

```

#include <stdio.h>
int main() {
    int var;
    return 0;
}

```

≡

```

#include <stdio.h>
int main() {
    auto int var;
    return 0;
}

```

2. Declare an auto variable or a **normal variable** and print it, It will print a **garbage value**
3. Declare a **global variable** and try to print it. It will print **0**.

TAKE AWAYS

1. If you won't initialize auto variable, by default it will be initialized with some garbage (random) value

```

#include <stdio.h>
int main() {
    auto int var;
    printf("%d", var);
    return 0;
}

```

TAKE AWAYS

2. On the other hand, global variable by default initialized to 0.

```

#include <stdio.h>
int var;
int main() {
    printf("%d", var);
    return 0;
}

```

## Extern Modifier:

Scoping rules:

1. **Why need scoping rules?** Bc the whole program of C **need not to be compiled at the same time**.

We can have **different files** that are compiled at different times. And previously compiled routines may be loaded from other **libraries**.

2. Scope of **automatic variable**:

The **scope** of a name is the part of the program within which the name can be used. For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared. Local variables of the same name in different functions are unrelated. The same is true of the parameters of the function, which are in effect local variables.

3. Scope of **extern variable and functions**:

The scope of an **external variable** or a function lasts from the point at which it is declared to the end of the file being compiled. For example, if `main`, `sp`, `val`, `push`, and `pop` are defined in one file, in the order shown above, that is,

```
main() { ... }

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }
```

then the variables `sp` and `val` may be used in `push` and `pop` simply by naming them; no further declarations are needed. But these names are not visible in `main`, nor are `push` and `pop` themselves.

4. **Extern variable** in same file and **out of scope** declaration. Variable referred to before it is defined.

On the other hand, if an external variable is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used, then an `extern` declaration is mandatory.

It is important to distinguish between the *declaration* of an external variable and its *definition*. A declaration announces the properties of a variable (primarily its type); a definition also causes storage to be set aside. If the lines

```
int sp;
double val[MAXVAL];
```

appear outside of any function, they *define* the external variables `sp` and `val`, cause storage to be set aside, and also serve as the declarations for the rest of that source file. On the other hand, the lines

```
extern int sp;
extern double val[];
```

*declare* for the rest of the source file that `sp` is an `int` and that `val` is a `double` array (whose size is determined elsewhere), but they do not create the variables or reserve storage for them.

5. In **extern keyword** below, the `extern` declares the variable but doesn't define it. Hence no memory space is reserved, hence no value is stored and hence no value is printed.

```

extern int a;

int main()
{
    printf("%d", a);
}

```

/tmp/ccVH9Qyv.o: In function `main':  
/home/coderpad/solution.c:14: undefined reference  
/home/coderpad/solution.c:14: undefined reference  
collect2: error: ld returned 1 exit status

6. Let us define it out of scope now.

<pre> extern int a;  int main() {     printf("%d", a); }  a = 10; </pre>	<pre> int a;  int main() {     printf("%d", a); }  a = 10; </pre>
--	---

Output is 10 in both cases.

7. Let us define a global variable with extern.

```

int a;

int main()
{
    printf("%d", a);
}

```

Output is 0 in this case. Since, it is a global variable.

8. Only one definition is allowed of extern variable.

extern declarations in the file containing the definition.) Array sizes must be specified with the definition, but are optional with an extern declaration.

*in file1:*

```

extern int sp;
extern double val[];

void push(double f) { ... }

double pop(void) { ... }

```

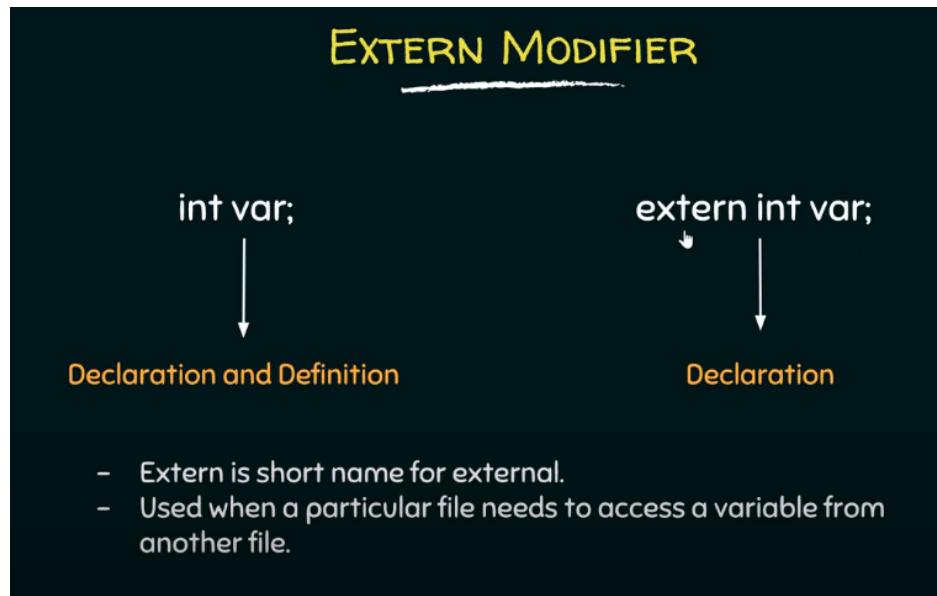
*in file2:*

```

int sp = 0;
double val[MAXVAL];

```

9. Declaration means variable exists, and definition means book a memory space for it.  
10. Extern modifier: The variable is externally declared (out of scope) and defined in some other file.  
Therefore, don't define it.  
11. Who can modify the variable? - Linker links all the files together. Compiler compiles the file.



12. Extern just says to the block that the variable is defined out of scope. **Used when a particular file needs to access a variable from another file.**

main.c × other.c ×

```

1 #include <stdio.h>
2
3 int a = 9;
4 int main()
5 {
6     extern int a;
7     printf("%d\n", a);
8     return 0;
9 }
10
  
```

C:\Users\jaspr\D  
9  
Process returned  
Press any key to

## TAKE AWAYS

1. When we write `extern some_data_type some_variable_name;` no memory is allocated. Only property of variable is announced.
2. Multiple declarations of extern variable is allowed within the file. This is not the case with automatic variables.
3. Extern variable says to compiler "go outside from my scope and you will find the definition of the variable that I declared".
4. Compiler believes that whatever the extern variable said is true and produce no error. Linker throws an error when he finds no such variable exist.
5. When an extern variable is **initialized**, then memory for this variable is allocated and it will be considered **defined**.

The screenshot shows a C IDE interface with two code files. The left file contains:

```
#include <stdio.h>

extern int a;
extern int a;
extern int a;
int main()
{
    printf("%d", a);
    return 0;
}
```

The right file contains:

```
#include <stdio.h>

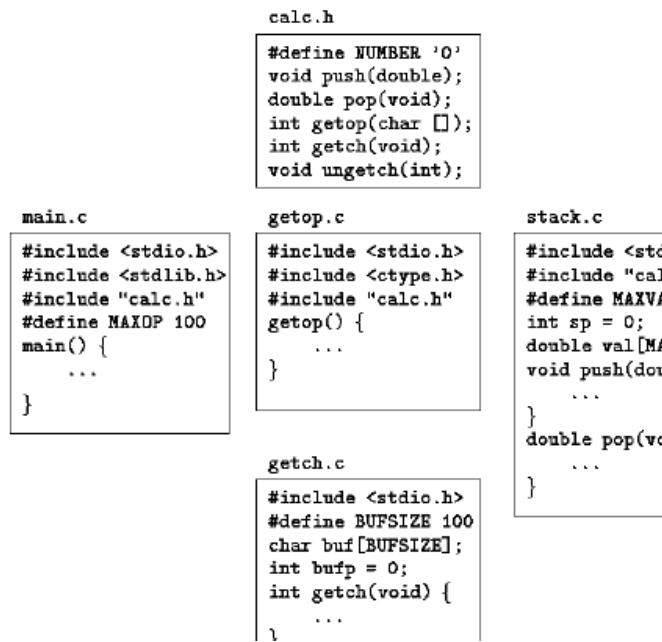
extern int a;
extern int a;
extern int a;
int main()
{
    printf("%d", a);
    return 0;
}
```

A cursor is positioned over the third declaration of 'a' in the right file. A tooltip window on the right side of the interface displays the text "C" and "5 Press".

13. Multiple **declarations** are allowed but **multiple definitions are not allowed**.
14. Once the **extern variable is initialized** then memory for it is allocated and will be **considered defined**.

## Header file:

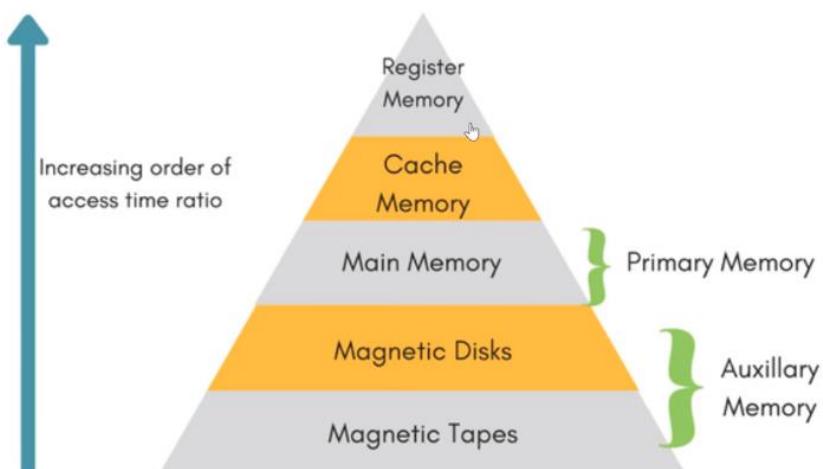
Keep things centralized. Below picture explains it very nicely.



## Register Modifier:

- A `register` declaration advises the compiler that the variable in question will be heavily used. The idea is that `register` variables are to be placed in machine registers, which may and so on. The `register` declaration can only be applied to automatic variables and to the formal parameters of a function. In this later case, it looks like

```
f(register unsigned m, register long n)  
{  
    register int i;  
    ...  
}
```



## WHAT IS REGISTER MODIFIER?

Syntax: register some\_data\_type some\_variable\_name

```
#include <stdio.h>
int main() {
    register int var;
    return 0;
}
```

## WHAT IS REGISTER MODIFIER?

Register keyword hints the compiler to store a variable in register memory.

This is done because access time reduces greatly for most frequently referred variables

This is the choice of compiler whether it puts the given variable in register or not.

Usually compiler themselves do the necessary optimizations

1. Whichever variable needs to be accessed multiple times, put that in register file. Since access time is fastest in it.

## Static modifier:

The external `static` declaration is most often used for variables, but it can be applied to functions as well. Normally, function names are global, visible to any part of the entire program. If a function is declared `static`, however, its name is invisible outside of the file in which it is declared.

The `static` declaration can also be applied to internal variables. Internal `static` variables are local to a particular function just as automatic variables are, but unlike automatics, they remain in existence rather than coming and going each time the function is activated. This means that internal `static` variables provide private, permanent storage within a single function.

1. How to protect a global variable from other files? – Declare it static

```
main.c x add.c x
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 extern int count;
5 int main()
6 {
7     int value;
8     value = increment();
9     value = increment();
10    value = increment();
11    count = count + 3;
12    value = count;
13    printf("%d", value);
14    return 0;
15 }
16

*add.c x
static int count;

int increment()
{
    count = count + 1;
    return count;
}
```

2. How to protect It from other functions within the same file? Declare it within the same function as static.
3. Static also makes the variable “STATIC” which means once it’s initialized with certain value within a function, then it won’t be reinitialized again if the function is called again.
4. By default, it’s initialized to 0, Just like global variable even though it’s declared locally.
5. And can’t be initialized using a variable.

```
main.c x add.c x
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 //extern int count;
5 int main()
6 {
7     int value;
8     value = increment();
9     value = increment();
10    value = increment();
11    //    count = count + 3;
12    //    value = count;
13    printf("%d", value);
14    return 0;
15 }
16

C:\Users\3
Process ID: 3
Press Any Key
```

## TAKEAWAYS

1. Static variable remains in memory even if it is declared within a block on the other hand automatic variable is destroyed after the completion of function in which it was declared.
2. Static variable if declared outside the scope of any function will act like global variable but only within the file in which it is declared.
3. You can only assign a constant literal (or value) to a static variable.

## Volatile modifier:

1. Why volatile: Everything is fine until the bloody compiler optimizes the variable.

```
int main(void)
{
    uint32_t    value = 0;
    uint32_t    *p = (uint32_t*) SRAM_ADDRESS1;
    while(1)
    {
        value = *p;
        if(value ) break;
    }

    while(1);

    return 0;
}
```

```
int main(void)
{
    uint32_t    value = 0;
    uint32_t    volatile *p = (uint32_t*) SRAM_ADDRESS1;

    while(1)
    {
        value = *p;
        if(value ) break;
    }

    while(1);

    return 0;
}
```

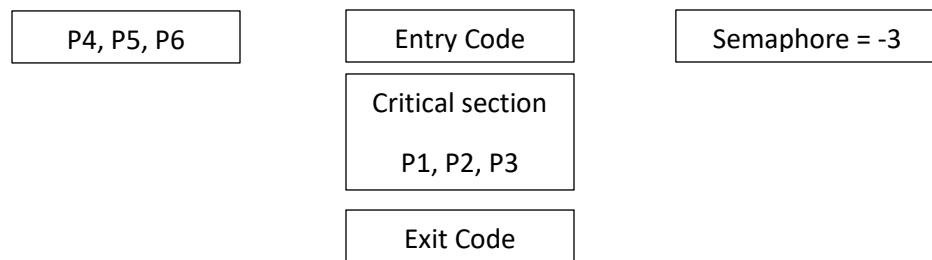
1. **Drawback of volatile keyword:** It can affect the variable if it is in the loop.

## Semaphore:

**Why semaphore:** To protect from **volatile** keyword.

**What is semaphore:** Integer value/ flag/semaphore variable used to protect the running code.

1. **Counting semaphore:** P wait and V signal.



- I. **Entry code:** P – wait – down.

```
int down(semaphore_value)
{
    semaphore_value = semaphore_value-1;

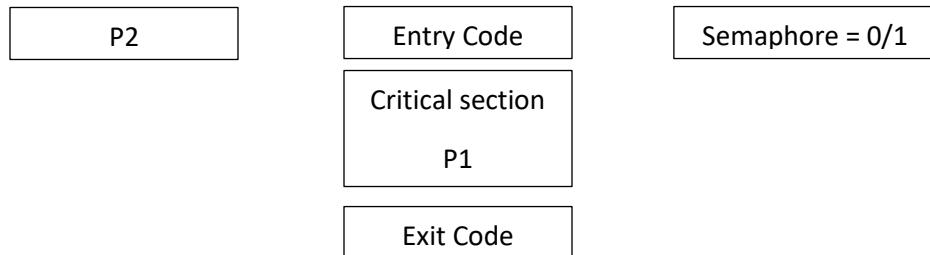
    if(semaphore_value<0)
    {
        put the process in sleep/block list;
    }
    else
    {
        return; //Enter critical section
    }
}
```

- II. **Exit code:** V – signal – up.

```
int up(semaphore_value)
{
    semaphore_value = semaphore_value+1;

    if(semaphore_value<=0)
    {
        Select a process in suspended list and wake up;
    }
}
```

2. **Binary semaphore:** P wait and V signal.



I. **Entry code:** P – wait – down.

```
//Initially semaphore is 1
int down_P(semaphore_value)
{
    while(semaphore_value == 0);

    semaphore_value = semaphore_value-1;

}
```

II. **Exit code:** V – signal – up.

```
int up_V(semaphore_value)
{
    semaphore_value = semaphore_value+1;
}
```

### Processes and Multi-processes programming in C:

**What is Process:** Main process is a process. Can create child process using **fork ()** function.

1. **Header file:** #include <unistd.h>
2. **Fork:** Create a new timeline from the line of fork instantiation. Two fork () will create 2 to power 2 processes timeline.
3. **Fork** returns a integer ID. Baby timeline has a id of 0. Father timeline has a non-zero ID.

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

int main()
{
    fork();
    fork();

    printf("Hello World!\n");
}
```

Harsh Dubey ran 26 lines  
Hello World!  
Hello World!  
Hello World!  
Hello World!

```
int main()
{
    int i = fork();

    if(i == 0)
    {
        printf("Hello baby\n");
    }else
        printf("Hello Father\n");
```

Harsh Dubey ran 26 lines  
Hello Father  
Hello baby

#### 4. How to create 3 processes?

```
int main()
{
    int id = fork();
    if(id)
    { fork();}

    if(id)
    {
        printf("Hello from main process\n");
    }else
    {
        printf("Hello from baby process \n");
    }
}
```

Hello from main process  
Hello from baby process  
Hello from main process  
Hello from main process  
Hello from main process

#### 5. Create a **fork process** and print first 5 number of 1-10 using **baby process** and then next five using **main process**: **fflush(stdout);** and **wait();**

- i. Main process is executed first here, but the idea is that **both main process and child process are happening at the same time.** **fflush(stdout)** will print the **characters as they are written in the LCD buffer.** Otherwise it will wait for it to fill the buffer and then print every thing.

```
int main()
{
    int id = fork();
    int n;

    if(id)
    {n = 6;}
    else
    {n=1;}
```

```
for(int i = n; i<n+5; i++)
{
    printf("%d ",i);
    fflush(stdout);
}
```

Harsh Dubey ran 29 lines  
6 7 8 9 10 1 2 3 4 5

- ii. To make the main process wait. Use **wait()** function.

```
int main()
{
    int id = fork();
    int n;

    if(id)
        {n = 6;}
    else
        {n=1;}

    if(id)
        {wait();}

    for(int i = n; i<n+5; i++)
    {
        printf("%d ",i);
        fflush(stdout);
    }
}
```

```
0 mean main ? [-wimp]
{wait();}
^~~~~~
main
1 2 3 4 5 6 7 8 9 10
```

6. How are process id's assign? 2^fork(): 00,01,10,11

- iii. Child of child = 0,0
- iv. Main: 1,1
- v. Whatever Id the child is born of will be Zero.
- vi. Rest whatever is left the child will inherit.

7. Pipe the process: Transfer data between the processes.

```

int a =0, b =0, rw[2];

//piped
if(pipe(rw) == -1)
{
    printf("Pipe not opened");
}
//forked
int id = fork();

if(id == -1)
{
    printf("Error forking");
}

//parallel
if(id == 0)
{
close(rw[0]);
for(int i = 0; i<5; i++)
{
    a++;
    sleep(1);

} if((write(rw[1], &a, sizeof(int))) == -1)
{
    printf("Error writing");
}
close(rw[1]);
}
else
{
close(rw[1]);
for(int i = 0; i<6; i++)
{
    b++;
    sleep(1);
}
if((read(rw[0], &a, sizeof(int))) == -1){
    printf("Error reading");
}
close(rw[0]);
printf("Sum %d", a+b);
}

```

8. Ex: Find sum of array by exploiting multiple processors on your system
- In the below example: Not **everything is faster** with parallel processing.

```

#define limit 10
int main()
{
    int fh = 0, sh = 0, arr[limit] =
{1,2,3,4,5,6,7,8,9,10}, rw[2];
    if(pipe(rw) == -1)
    {
        printf("Error in pipe");
    }
    int id = fork();
    if(id == -1)
    {
        printf("Error forking");
    }
    //divide and conquer
    if(id == 0)
    {
        close(rw[0]);
        for (int i = 0; i<limit/2; i++)
        {
            fh = fh + arr[i];
        }
        write(rw[1], &fh, sizeof(int));
        close(rw[1]);
    }
    else
    {
        close(rw[1]);
        for (int i = limit/2; i<limit; i++)
        {
            sh = sh + arr[i];
        }
        read(rw[0], &fh, sizeof(int));
        close(rw[0]);
        printf("Sum : %d ", sh+fh);
    }
}

```

---

```

#define limit 10
int main()
{
    int sum = 0, arr[limit] = {1,2,3,4,5,6,7,8,9,10};

    for(int i = 0; i<limit; i++)
    {
        sum = sum +arr[i];
    }
    printf("Sum : %d " ,sum);
}

```

## 9. Piping with more than two processes:

```

//processes: fork()
int main() {
    int a = 0, b = 0, c = 0, d=0, e=0, rw[2], id2;
    if(pipe(rw) == -1)
    {
        printf("Error opening the pipe");
    }
    int id = fork();
    //create a third child
    if(id)
    {id2 = fork();}

    if(!id)
    {
        //close read
        close(rw[0]);
        //computation
        for(int i = 0; i<5 ; i++)
        {
            a++;
            d++;
            sleep(1);
        }

        //write
        write(rw[1], &a, sizeof(int));
        write(rw[1], &d, sizeof(int));
        //close write
        close(rw[1]);
    }
    else if(!id2&&id)
    {
        //close read
        close(rw[0]);
        for(int i = 0; i<5 ; i++)
        {
            c++;
            e++;
            sleep(1);
        }
        write(rw[1], &c, sizeof(int));
        write(rw[1], &e, sizeof(int));
        close(rw[1]);
    }

    else if(id2&&id)
    {
        //close write
        close(rw[1]);
        for(int i = 0; i<5 ; i++)
        {
            b++;
            sleep(1);
        }
        read(rw[0], &a, sizeof(int));
        read(rw[0], &c, sizeof(int));
        read(rw[0], &d, sizeof(int));
        read(rw[0], &e, sizeof(int));
        close(rw[0]);
        printf("a: %d ",a);
        printf("C: %d ",c);
        printf("Sum: %d ",a+b+c+d+e);
    }
    return 0;
}

```

## Threading in C:

What is threading:

1. **Pthread\_t** : API will store information about the thread.
2. **Pthread\_create**: Create Thread: look at the image for more understanding.
3. **Pthread\_join**: Wait (); from process. Will wait for the function to execute and send the **return value**.

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

// To execute C, please define "int main()"

int a = 3;
void* routine()

{
    printf("Test from thread : %d secs\n",a);
    sleep(3);
    a++;
    printf("Ending thread %d\n",a);
}

int main() {

    pthread_t t1, t2;
    //Pointer to thread 1, attributes, function pointer,
    //function argument
    pthread_create(&t1, NULL, &routine, NULL);
    pthread_create(&t2, NULL, &routine, NULL);

    //What does pthread_join does: wait fro the thread to
    //finish it's execution
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

    return 0;
}
```

4. **Difference between Threads and Processes:**
  - i. Processes: Different ids, Threads: Same Id.
  - ii. **Single process can have multiple threads**, but **single thread can't have multiple processes**.
  - iii. Processes: **Copied resources**. Threads: **Shared Resources**.
5. **Race conditions:** Below is called the race condition.
  - i. Figure 2
  - ii. Creation of threads. If less conditions are there, then by the time the 2<sup>nd</sup> thread is created the first thread is already done with its work.

```

int mails = 0;
void *seemail()
{
    for(int i = 0; i<100000; i++)
    {mails++;}
    return 0;
}

int main()
{
    pthread_t t1,t2;
    if(pthread_create(&t1, NULL,&seemail, NULL )!=0)
    {
        return 1;
    }
    if(pthread_create(&t2, NULL,&seemail, NULL )!=0)
    {
        return 2;
    }
    if(pthread_join(t1, NULL) != 0)
    {
        return 3;
    }
    if(pthread_join(t2, NULL) != 0)
    {
        return 4;
    }

    printf("Mails: %d" , mails);
}

```

	t1	t2
// read mails	29	23
// increment	29	30
// write mails	30	24

6. **Mutex:** Way to solve a race condition:

- i. Declare: `pthread_mutex_t`
- ii. Define: `pthread_mutex_init()`;
- iii. Destroy: `pthread_mutex_destroy()`;

```

void *routine()
{
    for(int i = 0; i<100000; i++)
    {
        pthread_mutex_lock(&mutex);
        pthread_mutex_lock(&mutex2);
        mail++;
        pthread_mutex_unlock(&mutex);
        pthread_mutex_unlock(&mutex2);
    }
    return 0;
}

int main() {
    //thread: declare, define, wait

    pthread_t t1,t2;
    pthread_create(&t1, NULL, &routine, NULL);
    pthread_create(&t2, NULL, &routine_deliver, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    //mutex: declare, init, destroy
    pthread_mutex_init(&mutex, NULL );
    pthread_mutex_init(&mutex2, NULL );
    pthread_mutex_destroy(&mutex);
    pthread_mutex_destroy(&mutex);

    printf("Mails: %d", mail);

    return 0;
}

```

**7. Creating multiple threads using for loop:**

```

pthread_t arr[2];
for(int i = 0;i <2; i++)
{
    pthread_create(&arr[i], NULL, &routine, NULL);
    printf("Thread %d has started\n", i);
}
for(int i = 0;i <2; i++)
{
    pthread_join(arr[i], NULL);
    printf("Thread %d has finished execution\n",
i);
}

```

**8. Get a return value from thread: pthread\_join (Study pointer and memory allocation for that)**

**9. Pass arguments to thread: to 18**

**19. Deadlock: Nothing will execute.**

- i. **Single deadlock:** Lock the same mutex twice.
- ii. **Swap deadlock:** `rand()%2` could lead to dead lock.
- iii. **Two thread deadlock:** Two threads executing different functions and locking two different deadlocks in inappropriate order.

```

void* routine(void* args) {
    if (rand() % 2 == 0) {
        pthread_mutex_lock(&mutexFuel);
        pthread_mutex_lock(&mutexWater);
    } else {
        pthread_mutex_lock(&mutexWater);
        pthread_mutex_lock(&mutexFuel);
    }

    fuel += 50;
    water = fuel;
    printf("Incremented fuel to: %d set water to %d\n", fuel, water);
    pthread_mutex_unlock(&mutexFuel);
    pthread_mutex_unlock(&mutexWater);
}

```

**20. Recursive Mutex:** A mutex that can be **locked multiple times** is called a **recursive mutex**.

- i. **Why recursive mutex?** Recursive functions need recursive mutex, since the function calls itself.

**21. Semaphore: Binary and counting semaphore:** Semaphores are basically used to create queues. Binary semaphore acts like mutex but doesn't give ownership to one thread.

- ii. **How to create a semaphore:** Declare, init, destroy

```

sem_t semaphore;

//semaphore: declar, init, destroy
sem_init(&semaphore, 0, 1);

//semaphore destroy
sem_destroy(&semaphore);

```

- iii. **How to use:** `sem_wait(&semaphore)` : `sem_post(&semaphore)`

```

//pthread_mutex_lock(&mutex);
sem_wait(&semaphore);
mail++;
sem_post(&semaphore);

//pthread_mutex_unlock(&mutex);

```

- iv. **How to change it to counting or binary?** In init use to 3<sup>rd</sup> argument as n value for semaphore.

```

//semaphore: declar, init, destroy
sem_init(&semaphore, 0, 1);

```

- v. **How to add multiple process?** In init use to 2<sup>nd</sup> argument as number of processes.

**22. Semaphore vs Mutex:** Semaphores doesn't give ownership to any one thread, mutex gives complete ownership. Basically, another thread in the program could unlock the mutex locked by a different thread.

**23. Practical example of semaphore:** Game server example

### Random number generation: <stdlib>

1. **To create random numbers in C:**
  - i. We can use in built functions.
  - ii. Or we can create our own random number generator. However, before we do that, let's look at the in-built functions.
2. **Anytime you talk about any function:** The first thing you need is the library. And in the case of rand() and srand() we need <stdlib.h>.
3. **Random number generation:** using rand () and srand().
  - i. **Rand ():** int rand(void): basically rand() function just returns an integer.
    - **Return type:** Integer.
    - **What's the maximum and minimum value can rand () generate:**

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

int main()
{
    printf("%d", RAND_MAX);
}
```

Harsh Dubey ran 11  
2147483647

This makes sense because the **return type of rand ()** function is **integer**. There is no RAND\_MIN but the minimum value we can generate is int\_min.

- **One caveat:** it always returns the same number.

```
int main()
{
    printf("%d", rand());
}
```

1804289383  
Harsh Dubey  
1804289383  
Harsh Dubey  
1804289383

- We'll see why that happens when we write our own rand ().

- ii. **Srand():** void srand(unsigned int seed): There is no return type and it accepts unsigned int.
- iii. **Srand(x):** We can seed rand () with a **unsigned integer** and get a new pattern. But again, the new pattern will repeat if program executes multiple times. Rand () by default runs with **seed of 1**.
- iv. **To create real random numbers:** We have to **seed srand()** with a **value**, that's **constantly changing**. Hence, we'll used time.h library and time (); function.
- v. **time ():** time\_t time(time\_t\*second), if second is not a NULL ptr then the value returned is stored in the object pointed by second. Otherwise, **time\_t returns a time\_t value**. And can be stored in a time\_t type variable.

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

int main()
{
    time_t x;
    //get the time of the system
    time(&x);
    // x = time(NULL)

    printf("Time: %lu\n",x);
    //create random numbers
    srand(x);
    for(int i = 0; i<5; i++)
    {
        printf("%d\n",rand());
    }
}
```

Harsh Dubey ran 26 lines  
Time: 1619144863  
2048893596  
386105984  
1878181519  
1412656404  
1626994194

Harsh Dubey ran 26 lines  
Time: 1619144864  
680374736  
212233640  
399437820  
1038013222  
426159663  
□

- vi. **Creating real random number upto to a certain value:** Use modulus operator.

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

int main()
{
    time_t x;
    //get the time of the system
    time(&x);
    // x = time(NULL)

    printf("Time: %lu\n",x);
    //create random numbers
    srand(x);
    for(int i = 0; i<5; i++)
    {
        printf("%d\n",rand()%10);
    }
}
```

Harsh Dubey ran 26 lines  
Time: 1619144904  
9  
8  
7  
5  
9

Harsh Dubey ran 26 lines  
Time: 1619144908  
1  
7  
0  
0  
5  
□

- vii. **Print random number in positive range:**

`(rand() % ( upper-lower+1 )) + lower ;`

---

```
int main()
{
    srand(time(NULL)); //don't use __TIME__
    for(int i = 0; i<10; i++){
        printf("%d ",(rand()%61)+20); //from 20 to 80
    } // from 20 to 80 - 20 +1
}
```

- viii. **Print random numbers in negative to positive:**

```
printf("%d\n", (rand()%upper+lower+1)-lower);
```

- ix. **Floating point random number:** To generate a random floating point we need to do math. **Rand()** will return an **integer** hence type cast it and divide it by another integer to get random floating point value.

```

double random_float();
//floatign point random number and return the value
int main()
{
    double random_num;
    random_num = random_float();
    printf("%lf",random_num);

}

double random_float()
{
    srand(time(NULL));
    //if we want a floatign random number then we have to
    do some math.

    return (double)rand()/15;
}

```

Harsh Dubey ran 24  
63064657.466667  
Harsh Dubey ran 24  
92162804.600000  
Harsh Dubey ran 24  
10678707.800000

**x. Floating point numbers from 0 to x:**

```

double random_float(int );
//floatign point random number and return the value
int main()
{
    int num;
    printf("Enter a number: ");
    scanf("%d",&num);

    double random_num;
    random_num = random_float(num);
    printf("\nRandom number: %lf",random_num);

}

double random_float(int num)
{
    srand(time(NULL));
    //if we want a floatign random number then we have to
    do some math.

    return ((double)rand() / RAND_MAX) * num;
}

```

Random number: 0.573666  
Harsh Dubey ran 28 lines of C  
Enter a number: 10  
Random number: 2.551377  
Harsh Dubey ran 28 lines of C  
Enter a number: 20  
Random number: 0.930941  
Harsh Dubey ran 28 lines of C  
Enter a number: 25  
Random number: 17.573496

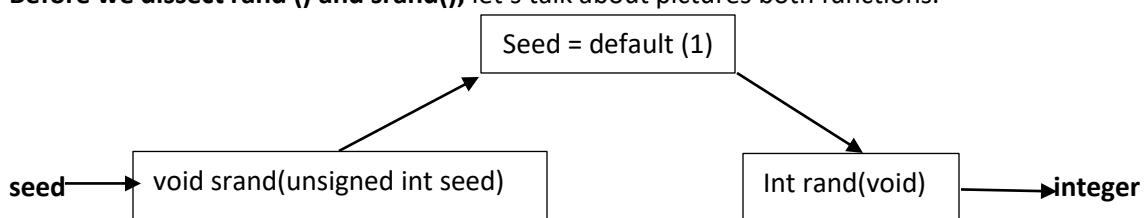
- xi. Floating point random number in a range:** Just add the upper value to the return. If num is 5 then the range will be from 5.5 to 5.5+num.

```
return (((double)rand() / RAND_MAX) * num) + 5.5;
```

**4. How C implements random number generator:** Linear Congruential Generator.

- i. **Formula:** integer = [ a x Seed + c ] % m; m > a&c&Seed

**5. Before we dissect rand () and srand(), let's talk about pictures both functions.**



**6. Let's implement or copy of rand() and srand() function:**

```

unsigned int seed = 1;
int random(void)
{
    int rand;
    static unsigned int a = 8191, c = 131071;
    rand = (a * seed + c)%2147483648;
    seed = rand;
    a = a * 524287;
    c = c * 6700417;
    return rand;
}
void srand(unsigned int u_seed)
{
    seed = u_seed;
}
int main()
{
    time_t x;
    //get the time of the system
    time(&x);
    // x = time(NULL)
    printf("Time: %lu\n",x);
    //create random numbers

    srand(x);
    for(int i =0; i< 5; i++)
    {
        printf("%d\n", random());
    }
}

```

1727423562  
1924720329  
2065661494  
1399145909  
3466314  
  
Harsh Dubey ran 48  
Time: 1619147707  
1727472708  
1525311171  
1116552764  
1648510395  
303604804  
  
Harsh Dubey ran 48  
Time: 1619147713  
1727521854  
1125902013  
167444034  
1897874881  
603743294  
[]

7. Let's write a simple function to generate random numbers without using rand():

8. Let's change the upper limit of rand(): Very important.

```

unsigned int seed = 1;
long int random(void)
{
    long int rand;
    static long int a = 8191, c = 131071;

    rand =(a * seed + c)%((unsigned long)pow(2,63));

    seed = rand;
    a = a * 524287;
    c = c * 6700417;

    return rand;
}
void srand(unsigned int u_seed)
{
    seed = u_seed;
}
int main()
{
    time_t x;
    //get the time of the system
    time(&x);
    // x = time(NULL)
    printf("Time: %lu\n",x);
    //create random numbers

    srand(x);
    for(int i =0; i< 5; i++)
    {
        printf("%ld\n", random());
    }
}

```

2103257819266719230  
7801538317933176189  
4375123187282855042  
[]

## Constant: #define and const:

**Macro:** Defining constant using #define.

### USING #DEFINE

Syntax: `#define NAME value`

Job of preprocessor (not compiler) to replace NAME with value.

```
#include <stdio.h>
#define PI 3.14159
int main() {
    printf("%.5f", PI);
    return 0;
}
```

Also called Macro

```
C:\Users\jaspr\Documents\consta
3.14159
Process returned 0 (0x0)   e
Press any key to continue.
```

We can use macros like functions.

```
#include <stdio.h>
#define add(x, y) x+y
int main() {
    printf("addition of two numbers: %d", add(4, 3));
    return 0;
}
```

⑤ We can write multiple lines using \

```
#include <stdio.h>
#define greater(x, y) if(x > y) \
                    printf("%d is greater than %d", x, y); \
                    else \
                    printf("%d is lesser than %d", x, y);
int main() {
    greater(5, 6);
    return 0;
}
```

1. You can use macros to print and compute. But code blocks don't let ya just compute.
2. Below is not allowed.

```

#define MulDiv(x,y) if(x>y) \
                    ( x*y); \
                else \
                    (x/y); \
int main()
{
    printf("%f",MulDiv(2.9,2.0));
    return 0;
}

```

3. Below is allowed.

```

#define MulDiv(x,y) if(x>y) \
                    printf("%f", x*y); \
                else \
                    printf("%.f", x/y); \
int main()
{
    //printf("%f",MulDiv(2.9,2.0));
    MulDiv(3.0,3.0);
    return 0;
}

```



## First expansion then evaluation.

```

#include <stdio.h>
#define add(x, y) x+y

int main() {
    printf("result of expression a * b + c is: %d", 5 * add(4, 3));
    return 0;
}

```

6. It will print  $5*4+3 = 23$ , not 35

7

Some predefined macros like `__DATE__`, `__TIME__` can print current date and time.

```
#include <stdio.h>
int main() {
    printf("Date: %s\n", __DATE__);
    printf("Time: %s\n", __TIME__);
    return 0;
}
```

```
C:\Users\jaspr\Documents\
Date: Mar 11 2018
Time: 08:33:21
```

7. Macros to print date and time.

Defining constant using Const:

```
#include <stdio.h>
int main()
{
    const int var = 57;
    var = 57;
    printf("%d", var);
}
```

8. Can't change the value later in the code.

```
9     const int baby = 3;
0
1     int main()
2     {
3
4     |     baby = 1;
5     |     printf("%d\n\n", baby);
6 }
```

9. But can be reinitialized.

```

const int baby = 3;

int main()
{
    const int baby = 100;
    printf("%d\n\n", baby);
}

```

10. But can't be reinitialized in the same scope.

```

0
9     const int baby = 3;
10
11     int main()
12     {
13
14         const int baby = 100;
15         printf("%d\n\n", baby);
16
17     const int baby = 177;
18     printf("%d\n\n", baby);
19

```

Questions:

**Q1:** What is the output of the following C program?

```

int main() {
    int var = 052;
    printf("%d", var);
    return 0;
}

```

- a) 52
- b) 56
- c) 42
- d) Compiler error

**Answer:** When we put zero in front of any value then we are making it a octal value.

$$\begin{matrix} 8^1 & 8^0 \\ 5 & 2 \end{matrix}$$

$$5 * 8 + 2 * 1 = 40 + 2 = 42$$

**Q2:** What is the output of the following C program?

```
#include <stdio.h>

#define STRING "%s\n"
#define NESO "Welcome to Neso Academy!"

int main() {
    printf(STRING, NESO);
    return 0;
}
```

- a) Compiler error
- b) "Welcome to Neso Academy!"
- c) Garbage value
- d) Welcome to Neso Academy!

Answer: D, no double quotes

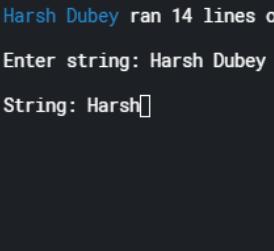
## Scanf and Printf and File terminal I/O operations:

### 1. Scanf and how to read a line of text in C?

- i. Using scanf with "%s" : Will terminate after a "Space". Will not take what's written after the space.

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

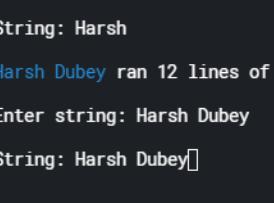
int main()
{
    char c[100];
    printf("Enter string: ");
    scanf("%s",c);
    printf("\nString: %s",c);
}
```



Harsh Dubey ran 14 lines of  
Enter string: Harsh Dubey  
String: Harsh

- ii. Use scanf with [^\n] operator to accept the space: This will tell the scanf that take everything before the \n.

```
int main()
{
    char c[100];
    printf("Enter string: ");
    scanf("%[^\\n]s",c);
    printf("\nString: %s",c);
}
```



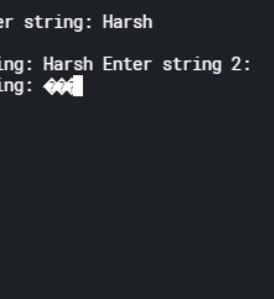
String: Harsh  
Harsh Dubey ran 12 lines of  
Enter string: Harsh Dubey  
String: Harsh Dubey

- iii. But problem comes when two string are supposed to be taken from the user: Since the old \n from the display buffer is not taken out. The [^\n] reads the \n from last string and the terminates.

```
#include<string.h>

int main()
{
    char c[100],c2[100];
    printf("Enter string: ");
    scanf("%[^\\n]s",c);
    printf("\nString: %s",c);

    printf("Enter string 2: ");
    scanf("%[^\\n]s",c2);
    printf("\nString: %s",c2);
}
```



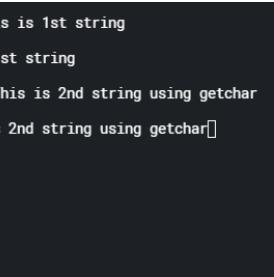
Enter string: Harsh  
String: Harsh Enter string 2:  
String: ◆◆◆

- iv. How to fix this: remove the \n from the buffer by using getchar() function to remove \n. However, bounds of the array are still insecure.

```
#include<string.h>

int main()
{
    char c[100],c2[100];
    printf("Enter string: ");
    scanf("%[^\\n]s",c);
    printf("\nString: %s",c);
    getchar();

    printf("\n\nEnter string 2: ");
    scanf("%[^\\n]s",c2);
    printf("\nString 2: %s",c2);
}
```



Enter string: This is 1st string  
String: This is 1st string  
Enter string 2: This is 2nd string using getchar()  
String 2: This is 2nd string using getchar()

- v. We can secure the size of array by using scanf: If we type extra characters then it won't store in the string, but it will store that in the display buffer.

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

int main()
{
    int size = 10;
    char c[size], c2[10];
    printf("Enter string: \n");
    scanf("%9[^\\n]s",c);
    getchar();
    printf("\nString: %s\n", c);

    printf("Enter string 2: \n");
    scanf("%9[^\\n]s",c2);
    printf("\nString 2: %s\n", c2);

}
```

Harsh Dubey ran 23 lines of C

Enter string:  
aaaaaaaaaaaaaaaaaaaaaa

String: aaaaaaaaa  
Enter string 2:  
String 2: aaaaaaaaa  
█

2. However we can use something better to secure the bounds: fgets(array, number of characters, stdin);

i. **Caveat:** It takes enter as an extra character. Notice the string length.

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

int main()
{
    char c[10];
    printf("Enter string: \n");
    fgets(c,10,stdin);
    printf("\nEnterd string : %s\n",c);
    printf("String length : %lu\n",strlen(c));

    for(unsigned int i = 0; i<=strlen(c);i++)
    {
        printf("%d ",c[i]);
    }

}
```

Harsh Dubey ran 20 lines

Enter string:  
aaa

Entered string : aaa

String length : 4  
97 97 97 10 0 █

ii. Fix this by putting 0 at strlen – 1: But now the string will only accept 8 characters.

```
int main()
{
    char c[10];
    printf("Enter string: \n");
    fgets(c,10,stdin);
    c[strlen(c)-1] = 0;
    printf("\nEnterd string : %s\n",c);
    printf("String length : %lu\n",strlen(c));

    for(unsigned int i = 0; i<=strlen(c);i++)
    {
        printf("%d ",c[i]);
    }

}
```

aaa

Entered string : aaa

String length : 4  
97 97 97 10 0

Harsh Dubey ran 21 lines of C (finished in 6.30s):

Enter string:  
aaa

Entered string : aaa

String length : 3  
97 97 97 0 █

3. Working with scanf and fgets(): scanf() leaves \n and “enter” that needs to be cleared if scanf and fgets have to be used together. Use fgetc() to remove the enter.

```
int main()
{
    int id;
    char message[256];
    while(1)
    {
        printf("Enter ID: \n");
        scanf("%d", &id);
        fgetc(stdin);
        printf("Enter Message: \n");
        fgets(message,256, stdin);
        printf("Id[%d] Message is : %s \n", id, message);
    }
}
```

Enter ID:  
2

Enter Message:  
Hello world

Id[2] Message is : Hello world

Enter ID:  
█

#### 4. Printf:

- i. Align the printed numbers to the right? By default, everything is aligned to left. **Pad it by some constant number.**

```
//floatign point random number and return the value
int main()
{
    int a= 255, b=300000;

    printf( "%10d\n%10d", a,b);

}
```

255  
300000

- ii. **Print Hex:** Hex : %x: small hex and %X: capital hex and %#x: Oxhex

```
int main()
{
    int a= 255;

    printf( "%x\n",a);
    printf( "%X\n",a);
    printf( "%#x\n",a);
```

ff  
FF  
0xff  
[]

- iii. **Print address of the variable:**

```
int main()
{
    int a= 255;

    printf( "%p\n",&a);
```

0x7fff5fce3030  
Harsh Dubey ran  
0x7ffe1e007380

- iv. **Print oct:** use %o.

#### 5. Reverse a string using fgets and scanf:

- i. **Fgets:**

```
#include<string.h>

int main()
{
    char c[100];

    printf("Enter the string : ");
    fgets(c,100,stdin);

    printf("\nString length: %lu\n", strlen(c));
    char reverse[strlen(c)];

    for(int i = 0; i<strlen(c)-1; i++)
    {
        reverse[i] = c[strlen(c)-2-i];
    }
    reverse[strlen(c)-1] = 0;

    printf("\nString : %s\n", c);
    printf("Reversed String : %s\n", reverse);
```

Enter the string : ABCD  
String length: 5  
String : ABCD  
Reversed String : DCBA  
[]

- ii. **Scanf:**

```

int main()
{
    char c[100];

    printf("Enter the string : ");
    scanf("%[^\\n]s",c);

    printf("\nString length: %lu\n", strlen(c));
    char reverse[strlen(c)+1];

    for(int i = 0; i<strlen(c); i++)
    {
        reverse[i] = c[strlen(c)-1-i];
    }
    reverse[strlen(c)] = 0;

    printf("\nString : %s\n", c);
    printf("Reversed String : %s\n", reverse);

    return 0;
}

```

String length: 4  
String : abcd  
Reversed String : dcba  
█

## 6. **Getc, getch, getche, getchar :**

- i. **Getc and getche:** are basically obsolete.
- ii. **Getc(stdin)/fgetc(stdin):** basically, just gets a character from the screen and returns it.
- iii. **Getchar():** is basically getc(stdin)

## 7. **Putc, putch, putchar():**

- i. **Putchar():** Takes an integer and then converts that to a character and prints it. Also, takes in character and prints it.

```

putchar('b');
putchar(10);
putchar(255+66);

```

b  
A█

- ii. **Putc(): putc('e',stdout);** takes two parameters, one is the character and other is the stream to write to. Stream, i.e., input /output.
- iii. **Return value of putchar and putc:** is the character they print on the screen.
- iv. **Putc('e',stdin):** Will return an EOF.

## 8. **Ungetc(c,stdin):** To read what's in then output buffer and then put it back in input buffer.

```

int main()
{
    ungetc('e',stdin);
    int a = getchar();
    printf("%c",a);

    return 0;
}

```

e█

## 9. **Gets() vs fgets():**

- i. **Vulnerability of gets() and a good example to understand memory layout and type casting:**

```

char str[10];
int var = 0;
fgets(str, 100, stdin);

printf("%s", str);
printf("%x", (char*)str - (char*)&var);
return 0;
}

```

```

prin
hh4
hh4
40

```

## 10. Printf: 32 different functions

- i. **S – buffer:** Example: sprint: write formatted string to a buffer.

```

int main()
{
    char buffer[100];
    sprintf(buffer, "Hello %d", 1);
    printf("%s", buffer);
}

```

Harsh Dubey  
Hello 1

- ii. **N- buffer size:**

```

int main()
{
    char buffer[100];
    snprintf(buffer, 10, "Hello123456789 %d", 1);
    printf("%s", buffer);
}

```

solution.c:8
iting 15 by
snprintf
solution.c:8
ze 10
snprintf
Hello1234

- iii. **F – file or stderr:** you can open a file and write to it then and there i.e. fprintf.

```

char buffer[100];
fprintf(stderr, "Hello123456789 %d", 1);

```

Hello123456789 1

- iv. **\_s : safer version of fprintf:** Used for safer file operations.

## Questions

**Q1:** what is the output of the following C program fragment:

```

#include <stdio.h>

int main() {
    int var = 0x43FF;
    printf("%x", var);
    return 0;
}

```

Answer: %x is used to print out hex

## Q2: what is the output of the following C program fragment:

```
#include <stdio.h>

static int i;
static int i = 27;
static int i;
int main() {
    static int i;
    printf("%d", i);
    return 0;
}
```

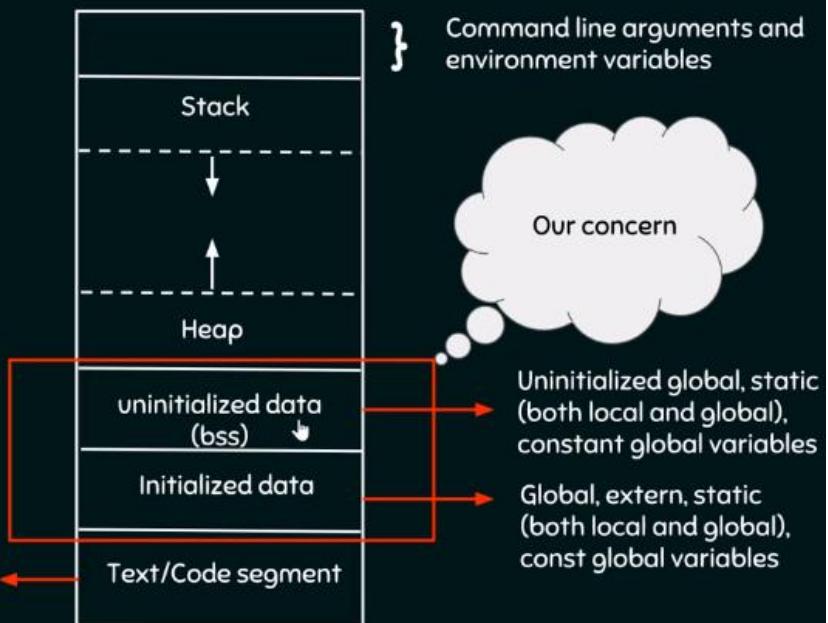
- a) 27
- b) 0
- c) No output
- d) None of the above

## MEMORY LAYOUT OF C PROGRAM

Two memory segments:

- 1) Text/code segment
- 2) Data segment
  - a) Initialized
    - i) Read only
    - ii) Read write
  - b) Uninitialized  
(bss - Block started by Symbol)
  - c) Stack
  - d) Heap

Contains machine code of the compiled program



## Operator in C:

Arithmetic Operators:

### ARITHMETIC OPERATORS

+ , - , \* , / , %

addition Subtraction Multiplication Division Modulus

All are binary operators → means two operands are required to perform operation

For example:

$A + B$

1. Modulus = remainder

# OPERATOR PRECEDENCE AND ASSOCIATIVITY

Precedence



Highest

Precedence	Operators	Associativity
Highest	* , / , %	Left to right
Lowest	+ , -	Left to right

2. Precedence is from top to bottom first. And from left to right
3. Remember MDM for multiplication, division and modulus.

## CODING EXAMPLE

```
#include <stdio.h>

int main() {
    int a = 2, b = 3, c = 4, d = 5;
    printf("a * b / c = %d\n", a*b/c);
    printf("a + b - c = %d\n", a+b-c);
    printf("a + b * d - c %% a = %d", a+b*d-c%a);
    return 0;
}
```

```
"C:\Users\jaspr\Downloads\C
a * b / c = 1
a + b - c = 2
a + b * d - c % a = 17
D ACADEMY"
```

1. Look at the 3<sup>rd</sup> one and tell why the

Increment and decrement Operators:

## INCREMENT AND DECREMENT OPERATORS

Pre-increment operator

`++a;`

Post-increment operator

`a++;`

Pre-decrement operator

`--a;`

Post-decrement operator

`a--;`

2. **Pre-increment (`++i`)** – Before assigning the value to the variable, the value is incremented by one.
3. **Post-increment (`i++`)** – After assigning the value to the variable, the value is incremented.

`x = ++a;`

`x = a++;`

x

a

6

~~5~~

x

a

5

~~5~~

6

## INCREMENT AND DECREMENT OPERATORS



You cannot use rvalue before or after increment/decrement operator.

Example:

$(a + b)++;$     error!

$++(a + b);$     error!

error: lvalue required as increment operand

1. Lvalue: should be a variable that has an identifiable location in the memory.
2. Can't be anything
3. Rvalue or right values are usually a expression or function that can return some value.

$a++;$

$a = a + 1;$

$(a+b)++;$

$(a+b) = (a+b) + 1;$

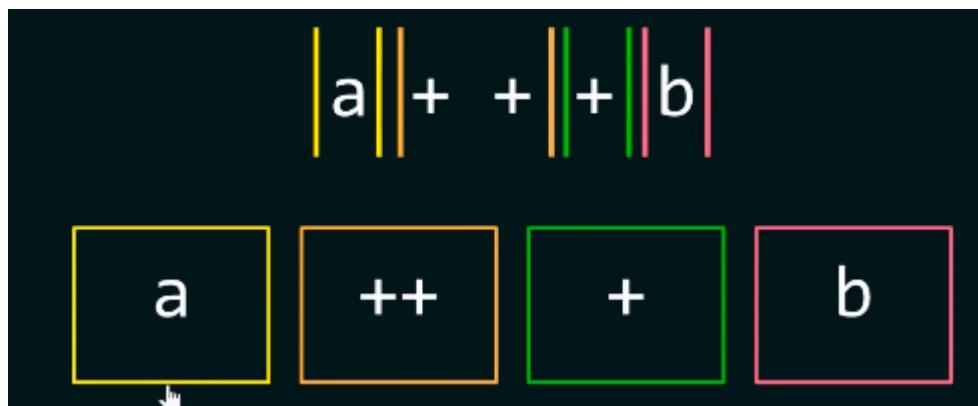
Questions:

**Q1:** What is the output of the following C program fragment:

```
#include <stdio.h>

int main() {
    int a = 4, b = 3;
    printf("%d", a+++b);
    return 0;    ↴
}
```

1. Lexem analyzer is analyzing lexemes from left to right.
2. Compiler will analyze valid lexemes and then assign them tokens



1. Post increment operator in terms of a equation : analyze the equation and then increment "a".

**Post increment/decrement  
in context of equation:**

First use the value in the  
equation and then  
increment the value

**Pre increment/decrement in  
context of equation:**

First increment the value  
and then use in the equation  
after completion of the  
equation.

**Q2:** What is the output of the following C program fragment:

```
#include <stdio.h>

int main() {
    int a = 4, b = 3;
    printf("%d", a + ++b);
    return 0;
}
```

Answer: 8

**Q3:** What is the output of the following C program fragment:

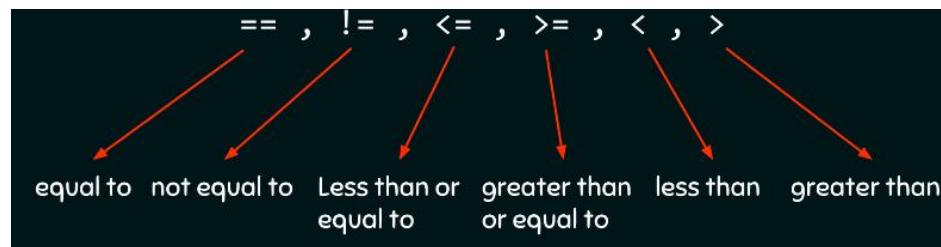
```
#include <stdio.h>

int main() {
    int a = 4, b = 3;
    printf("%d", a+++++b);
    return 0;
}
```

- a) 7
- b) 8
- c) 9
- d) Error

Answer: **Very important question.** Remember Lvalue concept.  $A++ = (a = a+1)++$ . **WRONG**

## Relational Operators:



## Logical Operators:

**Why logical operators:** Idea is to **optimize the code**. Maybe you are executing a lot of code and you want to decrease it. Then all you got to do is optimize by some sort of logical operation.

1. **And &&:** You are looking for conditions to be true by checking if **both operands** are **existing** or not. Any number that exists means it's true.

**Short circuit in case of &&:** simply means if there is a condition anywhere in the expression that returns false, then the rest of the conditions after that will not be evaluated.

<pre>#include &lt;stdio.h&gt;  int main() {     int a = 5, b = 3;     int incr;      incr = (a &lt; b) &amp;&amp; (b++);     printf("%d\n", incr);     printf("%d", b);     return 0; }</pre>	<pre>#include &lt;stdio.h&gt;  int main() {     int a = 5, b = 3;     int incr;      incr = (a &gt; b) &amp;&amp; (b++);     printf("%d\n", incr);     printf("%d", b);     return 0; }</pre>
---	---

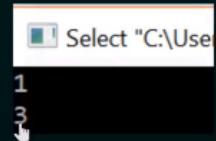
2. **OR ||:** You are looking for conditions to be true by checking if **anyone operands** is **existing** or not. Any number that exists means it's true.

**Short circuit in case of ||:** simply means if there is a condition anywhere in the expression that returns True, then the rest of the conditions after that will not be evaluated.

```
#include <stdio.h>

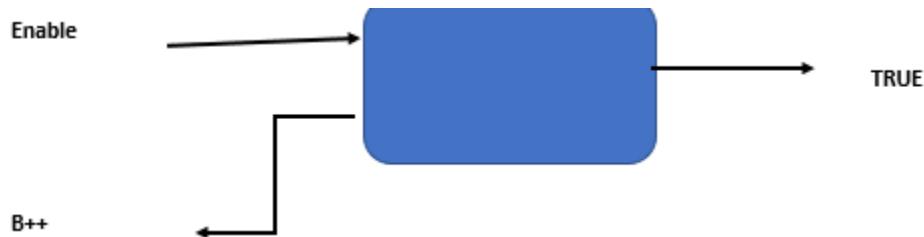
int main() {
    int a = 5, b = 3;
    int incr;

    incr = (a > b) || (b++);
    printf("%d\n", incr);
    printf("%d", b);
    return 0;
}
```



```
1
3
```

3. Look at the diagram below to understand short circuit for both && and ||



4. In case of && if Enable is high then B++ will BE looked at to produce true. If enable is low, then FALSE is the output.
5. In case of || if Enable is high then B++ will NOT be evaluated since, no need in case of OR. One high will produce TRUE.

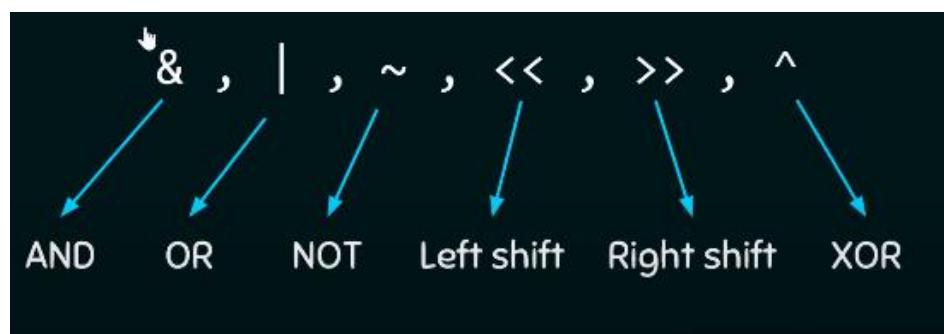
```

//short circuit in terms of AND and OR
int main()
{
    int a,b;
    printf("Enter A: \n");
    scanf("%d",&a);
    printf("Enter B: \n");
    scanf("%d",&b);

    if((a+b)|| (b++))
    {
        printf("Updated B: %d\n",b);
    }
    else
    {
        printf("Old B is : %d",b);
    }
}

```

### Bitwise Operators:



Bitwise AND:

Truth Table

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise AND:

7 → 0 1 1 1  
4 → & 0 1 0 0  
\_\_\_\_\_  
4 ← 0 1 0 0  
\_\_\_\_\_

7 & 4 = 4

Bitwise OR:

Truth Table

A	B	A B
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise OR:

7 → 0 1 1 1  
4 → | 0 1 0 0  
\_\_\_\_\_  
7 ← 0 1 1 1  
\_\_\_\_\_

7 | 4 = 7

Bitwise NOT:

Truth Table

A	$\sim A$
0	1
1	0

Bitwise NOT:

7 →  $\sim$  0 1 1 1  
\_\_\_\_\_  
8 ← 1 0 0 0  
\_\_\_\_\_

$\sim 7 = 8$

Bitwise XOR:

Truth Table

A	B	$A \wedge B$
0	0	0
0	1	1
1	0	1
1	1	0

7 → 0 1 1 1	
4 → ^ 0 1 0 0	
_____	
3 ← 0 0 1 1	

$7 \wedge 4 = 3$

#### Difference between Bitwise and Logical:

#### DIFFERENCE BETWEEN BITWISE AND LOGICAL OPERATORS

```
#include <stdio.h>

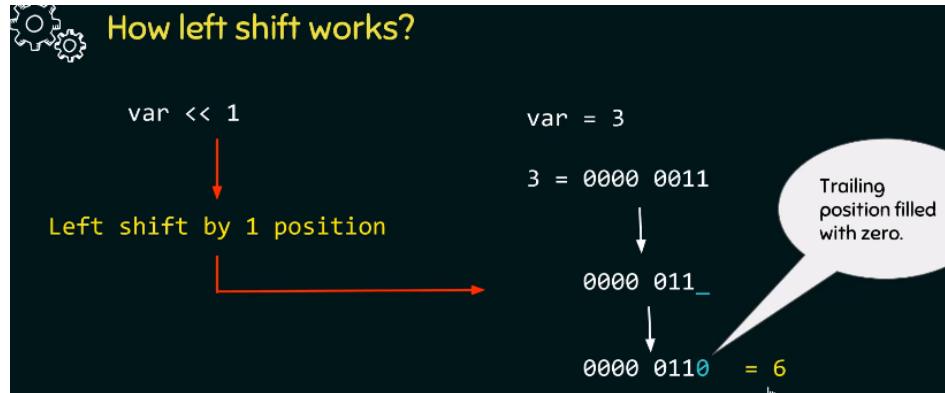
int main() {
    char x = 1, y = 2; //x = 1(0000 0001), y = 2(0000 0010)
    if(x&y)           //1&2 = 0(0000 0000)
        printf("Result of x&y is 1");
    if(x&&y)          //1&&2 = TRUE && TRUE = 1
        printf("Result of x&&y is 1");

    return 0;
}
```

#### Left shift operator:

```
#include <stdio.h>

int main() {
    char var = 3; //Note: 3 in binary = 0000 0011
    printf("%d", var<<1);
    return 0;
}
```



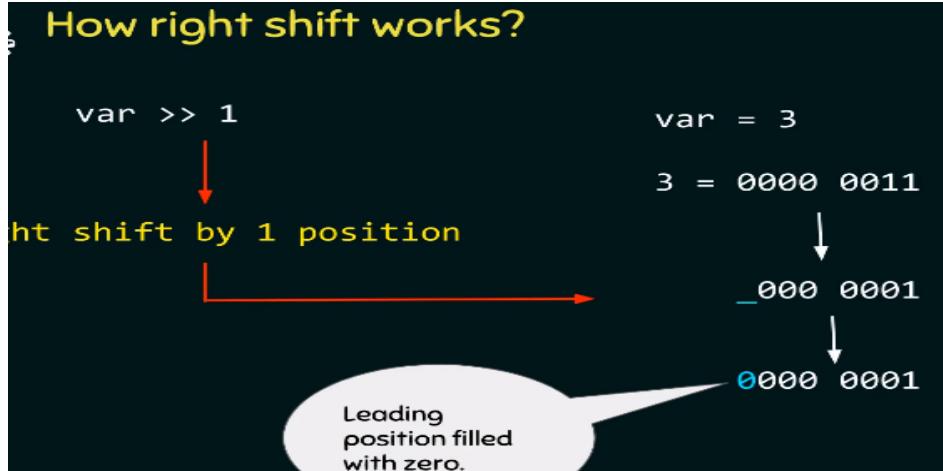
② Left shifting is equivalent to multiplication by  $2^{rightOperand}$

Example:

```
var = 3;
```

`var << 1`      Output: 6 [3 x 2<sup>1</sup>]

**Right shift operator:**



### IMPORTANT POINTS

- (2) Right shifting is equivalent to division by  $2^{right\ Operand}$

Example:

```

var = 3;
var >> 1      Output: 1 [3 / 21]

var = 32;
var >> 4      Output: 2 [32 / 24]

```

What is the output of the following program snippet?

```

#include <stdio.h>

int main() {
    int a = 4, b = 3;
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;

    printf("After XOR, a = %d and b = %d", a, b);
    return 0;
}

```

### Questions on Bit-wise operators:

**Question 1:** Check if a given number is a power of 2.

```
int main()
{
    int a;
    printf("Enter a number: ");
    scanf("%d",&a);

    if((a & (a-1)) == 0)
    {
        printf("\nNumber is power of 2");
    }
    else
    {
        printf("\nNumber is not power of 2");
    }

}
```

**Question 2:** Check what power of 2 is the number?

```
int main()
{
    int a,b;
    printf("Enter a number: ");
    scanf("%d",&a);
    int count = 0;
    b=a;
    do
    {
        if(a%2 == 0)
        {
            a= a>>1;
            count++;
        }
        else
        {
            printf("\nNumber is not power of 2");
            printf("\nCount : %d ",count);
            return 0;
        }
    }while(a!=1);

    printf("\n%d is 2 ^ %d ",b,count);
    return 0;

}
```

**Question 3:** Count the number of ones in the decimal.

```

int main()
{
    int a;
    printf("Enter a number: ");
    scanf("%d",&a);
    int count = 0;

    do
    {
        if(a%2 == 0)
        {
            a= a>>1;
        }
        else
        {
            a = a>>1;
            count++;
        }
    }while(a!=0);

    printf("Number of ones are: %d ",count);
    return 0;

}

```

**Question 4:** Check if the i-th bit is set.

```

int main()
{
    int n,b;
    char p;
    printf("Enter a number: ");
    scanf("%d",&n);
    printf("Enter the bit position: ");
    scanf("%d",&b);
    printf("Enter the 'l' for left and 'r' for right: ");
    scanf(" %c",&p);
    //getch();

    //printf("%c",p);

    if(p == 'l')
    {
        if((n>>(b-1))%2 == 1)
        {
            printf("The %d from left bit is SET" , b);
        }
        else
        {
            printf("The %d th bit is NOT SET" , b);
        }
    }
    else
    {
        if((n>>(31-b))%2 == 1)
        {
            printf("The %d from right bit is SET" , b);
        }
        else
        {
            printf("The %d th bit is NOT SET" , b);
        }
    }
}

return 0;
}

```

**Assignment Operators:** used to assign value to a variable.

<code>+=</code>	First addition than assignment	<code>%=</code>	First modulus than assignment
<code>-=</code>	First subtraction than assignment	<code>&lt;&lt;=</code>	First bitwise left shift than assignment
<code>*=</code>	First multiplication than assignment	<code>&gt;&gt;=</code>	First bitwise right shift than assignment
<code>/=</code>	First division than assignment	<code>&amp;=</code>	First bitwise AND than assignment
<b>Example:</b> <code>a += 1</code> is equivalent to <code>a = a + 1</code>			<code> =</code> First bitwise OR than assignment
Similar concept for other shorthand assignment operators as well			<code>^=</code> First bitwise XOR than assignment

**Conditional Operators:** used to assign value to a variable.

- `? :`

```
char result;
int marks;

if (marks > 33)
{
    result = 'p';
}
else
{
    result = 'f';
}
```

```
char result;
int marks;
```

```
result = (marks > 33) ? 'p' : 'f';
```

What is the output of the following C program fragment?

```
#include <stdio.h>

int main() {
    int var = 75;
    int var2 = 56;
    int num;

    num = sizeof(var) ? (var2 > 23 ? ((var == 75) ? 'A' : 0) : 0) : 0;

    printf("%d", num);
    return 0;
}
```

Answer: 65

**Comma Operators:** used to assign value to a variable.

```
int a = 3, b = 4, c = 8;
```

```
int a = (3, 4, 8);
```

```
printf("%d", a);
```

1. Comma operator will evaluate all the values but assign the right most one.
2. Below it will print HELLO and then assign VAR = 5 and later print it on screen.

```
int var = (printf("%s\n", "HELLO!"), 5);
```

```
printf("%d", var);
```

### HOMEWORK PROBLEM

What is the output of the following C program fragment?

```
#include<stdio.h>
int main()
{
    int var;
    int num;

    num = (var = 15, var+35);
    printf("%d", num);
    return 0;
}
```

- a) 15
- b) 50
- c) No output
- d) Error

Answer: **Very Important:** 50

### **Precedence and Associativity of Operators:**

1. Higher precedence operator will be given priority.

**Associativity:** Associativity comes in the picture when precedence of the operators is same.

## PRECEDENCE AND ASSOCIATIVITY TABLE

CATEGORY	OPERATORS	ASSOCIATIVITY
Parenthesis/brackets	( ) [ ] -> . ++ --	Left to right
Unary	! ~ ++ -- + - * & (type) sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Bitwise Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right

## PRECEDENCE AND ASSOCIATIVITY TABLE

CATEGORY	OPERATORS	ASSOCIATIVITY
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	? :	Right to left
Assignment	= += -= *= /= %= &= ^=  = <<= >>=	Right to left
Comma	,	Left to right

What is the output of the following C program fragment?

```
#include <stdio.h>

int main() {
    //code
    int a=10, b=20, c=30, d=40;
    if(a <= b == d > c)
    {
        printf("TRUE");
    }
    else
    {
        printf("FALSE");
    }
    return 0;
```

- a) TRUE
- b) FALSE

Ans: TRUE

Questions:

What is the output of the following C program fragment? Assume size of integer is 4 bytes.

```
#include <stdio.h>
int main() {
    int i = 5;
    int var = sizeof(i++);
    printf("%d %d", i, var);
    return 0;
}
```

Answer: is 54 not 64. Because according to C99 standard(not C11), sizeof(i++) will give size of variable type and will only solve the expression when it's a variable array type.

What is the output of the following C program fragment?

```
int a = 1;
int b = 1;
int c = ++a || b++;
int d = b-- && --a;

printf("%d %d %d %d", d, c, b, a);
```

- a) 1 1 1 1
- b) 0 1 0 0
- c) 1 0 0 1
- d) 1 1 0 1

Answer: **Very Important**, work again

**Rapid Fire:**

Go back to Video 37 and do the quiz.

## Conditionals:

If else:

Switch: Good replacement to long if else constructs

**WHAT IS SWITCH?**

Switch is a great replacement to long else if constructs.

**Example:**

```
int x = 2;

if (x == 1)
    printf("x is 1");
else if(x == 2)
    printf("x is 2");
else if(x == 3)
    printf("x is 3");
else
    printf("x is a number
other than 1, 2 and 3");
```

```
int x = 2;

switch(x)
{
    case 1: printf("x is 1");
              break;
    case 2: printf("x is 2");
              break;
    case 3: printf("x is 3");
              break;
    default: printf("x is a
number other than 1, 2, and 3");
              break;
}
```

SO ACADEMY

2. Why put break? If you don't put break, then substituent functions will be evaluated **once the condition is satisfied** until we reach the **break**.
3. Conditions of switch: You aren't allowed to duplicate cases.

```
int main() {
    int x = 1;
    switch(x)
    {
        case 1: printf("x is 1");
                  break;
        case 1: printf("x is 1");
                  break;
        case 2: printf("x is 2");
                  break;
    }
}
```

**Output:**

```
prog.c:9:6: error: duplicate case value
      case 1: printf("x is 1");
                  ^
prog.c:7:6: error: previously used here
      case 1: printf("x is 1");
```

4. Only those expressions are allowed those are integral **constant values** in switch statement. **No floats** allowed.

**ALLOWED**

```
int main() {
    int a = 1, b = 2, c = 3;
    switch(a+b*c)
    {
        case 1: printf("choice 1");
                  break;
        case 2: printf("choice 2");
                  break;
        default: printf("default");
                  break;
    }
}
```

**NOT ALLOWED**

```
int main() {
    float a = 1.15, b = 2.0, c = 3.0;
    switch(a+b*c)
    {
        case 1: printf("choice 1");
                  break;
        case 2: printf("choice 2");
                  break;
        default: printf("default");
                  break;
    }
}
```

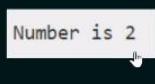
5. **Floats are not allowed in the CASE statement either.** However, **expressions** are allowed that will lead to integer value.

### FACTS RELATED TO SWITCH

**(3)** Float value is not allowed as a constant value in **case label**. Only integer constants/constant expressions are allowed in case label.

<p style="text-align: center;"><b>NOT ALLOWED</b></p> <pre>int main() {     float x = 3.14;     switch(x)     {         case 3.14: printf("x is 3.14");                     break;         case 1.1: printf("x is 1.14");                     break;         case 2: printf("x is 2");                     break;     } }</pre> <p style="font-size: small; margin-top: 10px;">         prog.c:7:6: error: case label does not reduce to an integer constant          case 3.14: printf("x is 3.14");            ^          prog.c:9:6: error: case label does not reduce to an integer constant          case 1.1: printf("x is 1.14");     </p>	<p style="text-align: center;"><b>ALLOWED</b></p> <pre>int main() {     int x = 23;     switch(x)     {         case 3+3: printf("choice 1");                     break;         case 3+4*5: printf("choice 2");                     break;         default: printf("default");                     break;     } }</pre> <p style="text-align: right; margin-top: 10px;">  </p>
---	--

6. **Variable expressions aren't allowed.** However, **macros** are allowed. Also, Integer.

<pre>int main() {     int x = 2, y = 2, z = 23;     switch(x)     {         case y: printf("Number is 2");                     break;         case z: printf("Number is 23");                     break;         default: printf("default case");                     break;     } }</pre> <p style="font-size: small; margin-top: 10px;">         prog.c:7:6: error: case label does not reduce to an integer constant          case y: printf("Number is 2");            ^          prog.c:9:6: error: case label does not reduce to an integer constant          case z: printf("Number is 23");     </p>	<pre>#include &lt;stdio.h&gt; #define y 2 #define z 23  int main() {     int x = 2;     switch(x)     {         case y: printf("Number is 2");                     break;         case z: printf("Number is 23");                     break;         default: printf("default case");                     break;     } }</pre> <p style="text-align: center; margin-top: 10px;"> <b>Output:</b>   </p>
--	---

7. **Default** can be put anywhere. And will only be evaluated once all the cases are evaluated.

**For while loop:**

**While loop:**

<pre>int i = 3 while( i &gt; 0 ) {     print ( i );     i--; }</pre>	<pre>int i = 3 while( TRUE ) {     print ( i );     i--; }</pre>
--	--

**For loop:**

```
for(initialization; condition; increment/decrement)
{
    Statements;
}
```

<pre>int i; i = 3; while(i &gt; 0) {     print (i);     i--; }</pre>	≡	<pre>for(i = 3; i &gt; 0; i--) {     print (i); }</pre>
--	---	---

**Do while loop:** Evaluate the arguments and then check the condition. Unlike while, where you check the condition first and then evaluate argument.

<p><b>While</b></p> <pre>int i = 0; while(i &gt; 0) {     printf("%d", i);     i--; }</pre>	<p><b>do-While</b></p> <pre>int i = 0; do {     printf("%d", i);     i--; } while(i &gt; 0);</pre>
---	--

**Output:** No Output

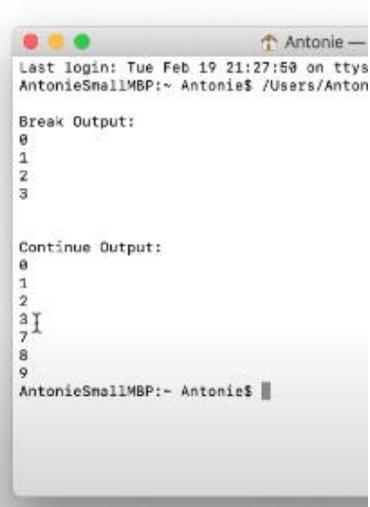
8. Write a program that allows user to enter a value, until he or she enters zero. Using while and do while genius.

## Break and continue:

Difference between Break and Continue: Break the loop and Skip the loop.

```
printf("\nBreak Output:\n");
int i;
for (i=0; i<10; i++)
{
    if ((i >= 4) && (i <= 6))
        break;
    printf("%d\n",i);
}

printf("\n\nContinue Output:\n");
for (i=0; i<10; i++)
{
    if ((i >= 4) && (i <= 6))
        continue;
    printf("%d\n",i);
}
```



```
Last login: Tue Feb 19 21:27:58 on ttys
Antonie$ /Users/Antonie$ 
Break Output:
0
1
2
3

Continue Output:
0
1
2
3
7
8
9
Antonie$
```

### Questions:

How many times will “Hello, World” be printed in the below program?

```
#include <stdio.h>
int main() {
    int i = 1024;
    for(; i; i >= 1)
        printf("Hello, World");
    return 0;
}
```

a) 10  
b) 11  
c) infinite  
d) compile time error

Answer: 11

What is the output of the following C program fragment?

```
#include <stdio.h>
int main()
{
    int i;
    for(i=0; i<20; i++)
    {
        switch(i)
        {
            case 0: i += 5;
            case 1: i += 2;
            case 5: i += 5;
            default: i += 4;
        }
        printf("%d ", i);
    }
}
```

ESO ACADEMY

- a) 5 10 15 20
- b) 7 12 17 22
- c) Compiler error
- d) 16 21

Answer: Do it on your own.

How many times will “Neso” be printed on the screen?

```
int i = -5;
while(i <= 5)
{
    if(i >= 0)
        break;
    else
    {
        i++;
        continue;
    }
    printf("Neso");
}
```

- a) 10 times
- b) 5 times
- c) Infinite times
- d) 0 times

Answer: Do it on your own

What is the output of the following C program fragment?

```
int main()
{
    int i = 0;
    for(printf("one\n"); i < 3 && printf(""); i++)
    {
        printf("Hi!\n");
    }
    return 0;
}
```

Answer: **Very Important Question.** Do it again.

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

int indicies( int array[], int target, int size);

int main()
{
    int i = 0;
    for(printf("one\n"); i<3 && printf("look here |"); i++)
    {
        printf("Hi!\n");
    }

    return 0;
}
```

```
one
look here Hi!
look here Hi!
look here Hi!
```

Also, look. ONE is only printed once.

What is the output of the following C program fragment?  
Assuming size of unsigned int is 4 bytes.

```
#include <stdio.h>
int main()
{
    unsigned int i = 500;
    while(i++ != 0);
    printf("%d", i);
    return 0;
}
```

a) Infinite loop  
b) 0  
c) 1  
d) Compiler error

Ans: Do it again.

What is the output of the following C program fragment?

```
#include <stdio.h>
int main()
{
    int x = 3;
    if(x == 2); x = 0;
    if(x == 3) x++;
    else x += 2;

    printf("x = %d", x);
    return 0;
}
```

- a) x = 4
- b) x = 2
- c) Compiler error
- d) x = 0

Answer: Do it again

## Pyramid of Stars Palindrome Armstrong

### Function:

Why functions?

- Reusability and abstraction

Function declaration as a prototype:

#### WHAT IS FUNCTION DECLARATION

Similarly, function declaration (also called **function prototype**) means declaring the properties of a function to the compiler.

For example: int fun(int, char);

##### Properties:

1. Name of function: fun
2. Return Type of function: int
3. Number of parameters: 2
4. Type of parameter 1: int
5. Type of parameter 2: char

#### IMPORTANT TAKEAWAY



It is not necessary to put the name of the parameters in function prototype.

For example: int fun(int var1, char var2);

Not necessary to mention  
these names

- Is it always necessary to **declare** a function before using it? No. You **do not** have to **declare**. Instead of declaration you can define the function too before the main function.

- What happens when you do not declare or define a function? Implicit declaration

```

1 #include <stdio.h>
2 int main()
3 {
4     char c = fun();
5     printf("character is: %c", c);
6 }
7
8 char fun()
9 {
10    return 'a';
11 }
12

```

Message

```

== Build file: "no target" in "no project" (compiler: unknown) ==
In function 'main':
warning: implicit declaration of function 'fun' [-Wimplicit-function-declaration]
error: conflicting types for 'fun'

```

- Two errors: Conflicting types for “fun” and Implicit declaration of function.
- **Conflicting types Error:** Since the function is not declared or defined, therefore, compiler assumes the return type of the function to be integer data type.
- **Implicit declaration warning:** Compiler implicitly assumed return type to be integer.
- If you declare an integer type than you can bypass function call and declaration.

## WHAT IS THE DIFFERENCE BETWEEN AN ARGUMENT AND A PARAMETER?

**Parameter:** is a variable in the declaration and definition of the function.

**Argument:** is the actual value of the parameter that gets passed to the function.

```

int add(int, int);

int main()
{
    int m = 20, n = 30, sum;
    sum = add(m, n);
    printf("sum is %d", sum);
}

int add(int a, int b)
{
    return (a + b);
}

```

## Call by value and call by reference: With example!

### CALL BY VALUE

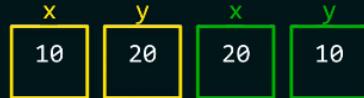
Here values of actual parameters will be copied to formal parameters and these two different parameters store values in different locations

```
int x = 10, y = 20;  
fun(x, y);
```

```
printf("x = %d, y = %d", x, y);
```

```
int fun(int x, int y)  
{  
    x = 20;  
    y = 10;  
}
```

Output: x = 10, y = 20



### CALL BY REFERENCE

```
int x = 10, y = 20;  
fun(&x, &y);
```

```
printf("x = %d, y = %d", x, y);
```

```
int fun(int *ptr1, int *ptr2)  
{  
    *ptr1 = 20;  
    *ptr2 = 10;  
}
```

Output: x = 20, y = 10



## Questions:

The output of the following C program is:

```
void f1(int a, int b)          int main()
{
    int c;                  {
    c = a; a = b; b = c;      int a=4, b=5, c=6;
}                                f1(a, b);
void f2(int *a, int *b)        f2(&b, &c);
{
    int c;                  printf("%d", c-a-b);
    c = *a; *a = *b; *b = c; return 0;
}
```

[GATE 2015 - Set 1]

Answer: -5

Consider the following C program:

```
int fun()
{
    static int num = 16;
    return num--;
}

int main()
{
    for(fun(); fun(); fun())
    printf("%d ", fun());
    return 0;
}
```

What is the output of the C program available in the LHS?

- a) Infinite loop
- b) 13 10 7 4 1
- c) 14 11 8 5 2
- d) 15 12 8 5 2

Answer: Very important. Do it again.

Static Function:

### BASICS



In C, functions are global by default.



This means if we want to access the function outside from the file where it is declared, we can access it easily.



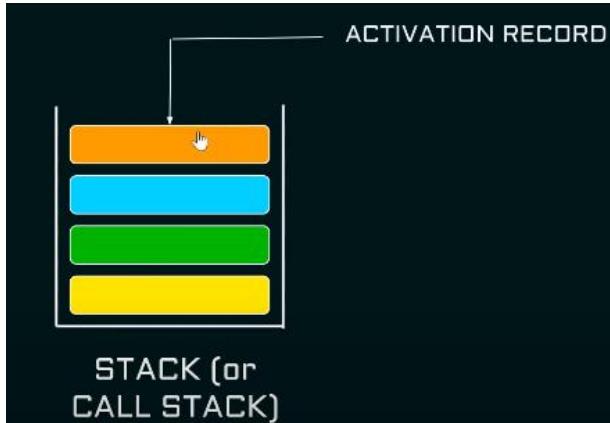
Now if we want to restrict this access, then we make our function static by simply putting a keyword **static** in front of the function.

## Static scoping and dynamic scoping in C:

- ★ Stack is a container (or memory segment) which holds some data.
- ★ Data is retrieved in Last In First Out (LIFO) fashion.
- ★ Two operations: `push` and `pop`.



- ★ Stack is a container (or memory segment) which holds some data.
- ★ Data is retrieved in Last In First Out (LIFO) fashion.
- ★ Two operations: `push` and `pop`.



Activation Record ( ) – is a portion of a stack which is generally composed of:

1. Locals of the callee
2. Return address to the caller
3. Parameters of the callee

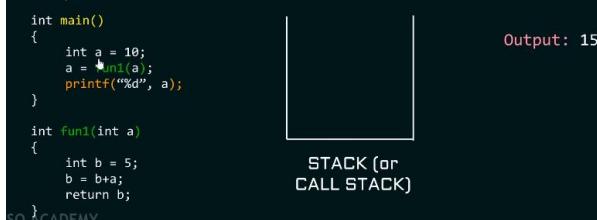
Example:



Activation Record ( ) – is a portion of a stack which is generally composed of:

1. Locals of the callee
2. Return address to the caller
3. Parameters of the callee

Example:



## Scoping:

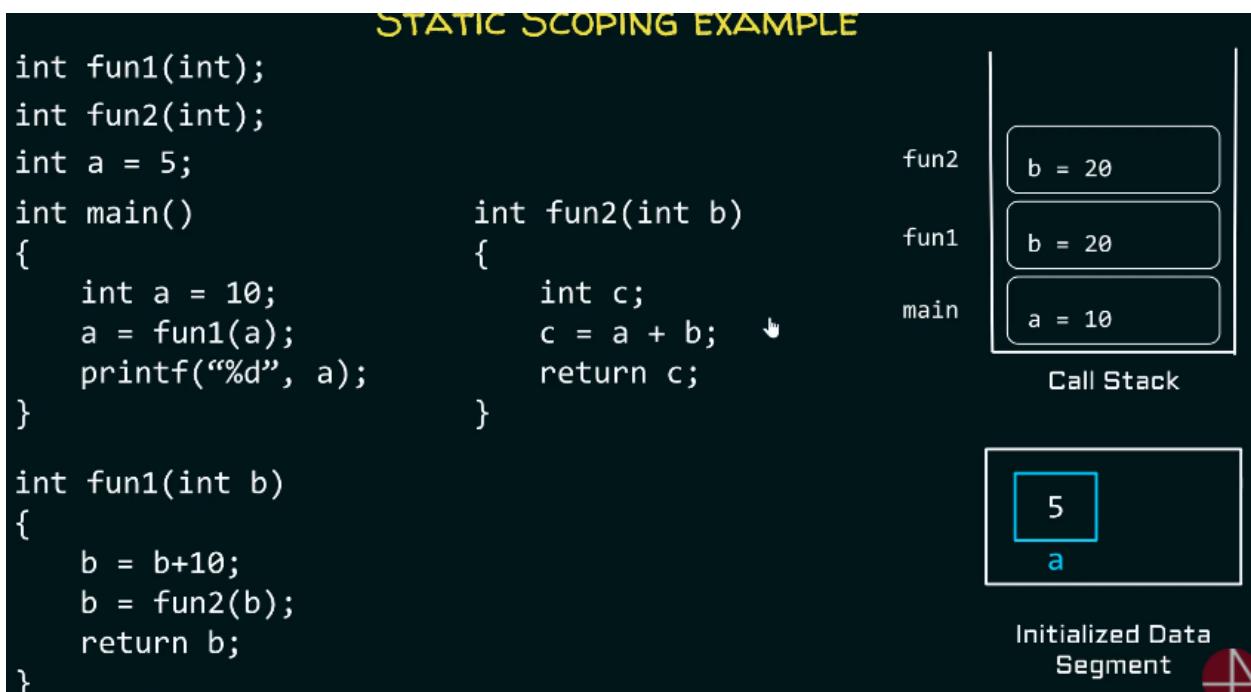
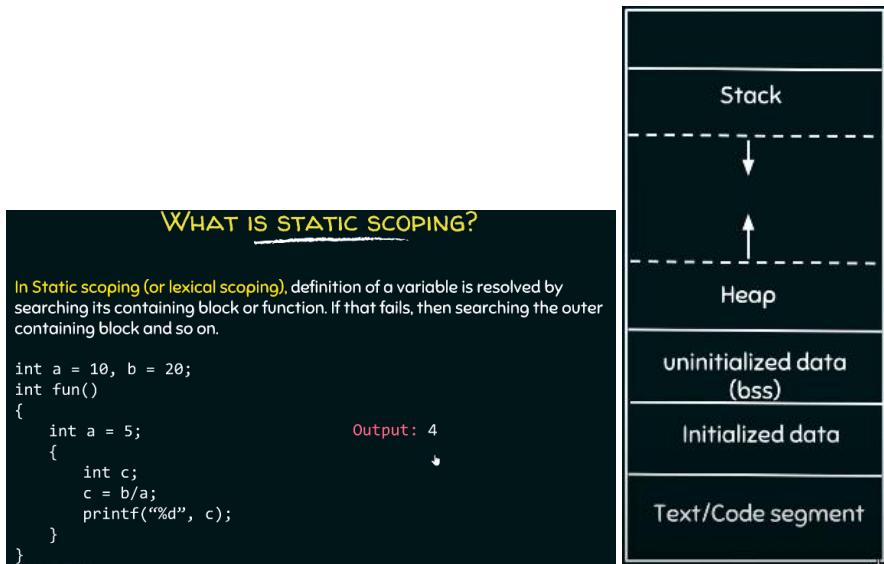
### WHY SCOPING?

Scoping is necessary if you want to reuse variable names

Example:

```
int fun1()
{
    int a = 10;
}

int fun2()
{
    int a = 40;
}
```



Dynamic scoping:

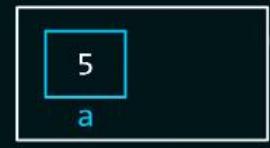
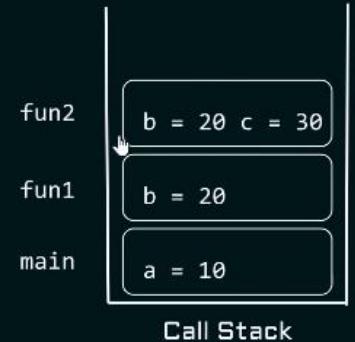
## WHAT IS DYNAMIC SCOPING?

In dynamic scoping, definition of a variable is resolved by searching its containing block and if not found, then searching its calling function and if still not found then the function which called that calling function will be searched and so on.

### DYNAMIC SCOPING EXAMPLE

```
int fun1(int);
int fun2(int);
int a = 5;
int main()
{
    int a = 10;
    a = fun1(a);
    printf("%d", a);
}

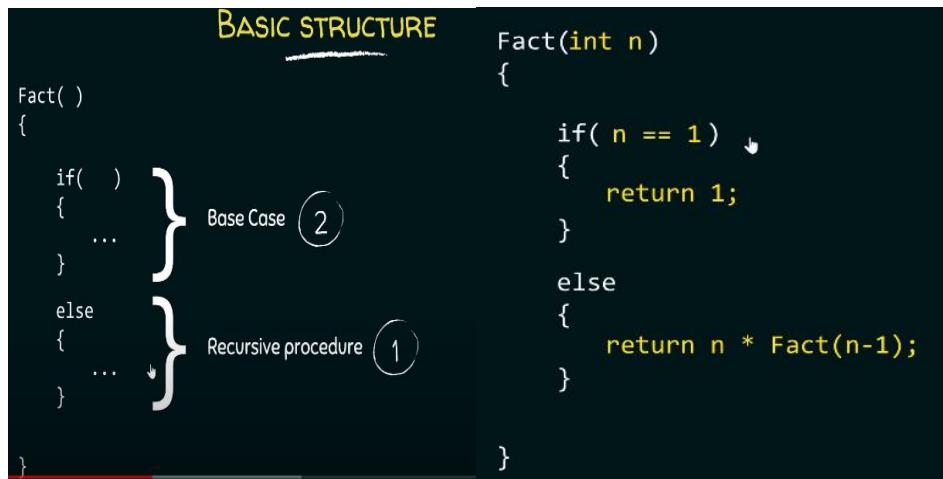
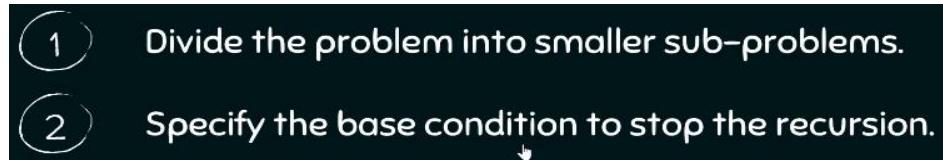
int fun1(int b)
{
    b = b+10;
    b = fun2(b);
    return b;
}
```



Initialized Data Segment

## Recursion:

1. **Recursion:** Function calling itself.
2. **How to write a recursive function:**
  - i. If – else construct
  - ii. If (base): return 1; terminate the recursion.
  - iii. Else : return the function itself.



3. How to find **recursive procedure**: Look at the picture below.

```
Calculate Fact(4)

Fact(1) = 1
Fact(2) = 2 * 1 = 2 * Fact(1)
Fact(3) = 3 * 2 * 1 = 3 * Fact(2)
Fact(4) = 4 * 3 * 2 * 1 = 4 * Fact(3)

Fact(n) = n * Fact(n-1)
```

4. How to find the **base condition**: Condition which doesn't require to call the same **function again**.

```
Calculate Fact(4)
```

```
Fact(1) = 1
```

```
Fact(2) = 2 * 1 = 2 * Fact(1)
```

```
Fact(3) = 3 * 2 * 1 = 3 * Fact(2)
```

```
Fact(4) = 4 * 3 * 2 * 1 = 4 * Fact(3)
```

Base condition is the one which doesn't require to call the same function again and it helps in stopping the recursion.

#### 5. Types of recursion:

### TYPES OF RECURSION

- 1 Direct recursion
- 2 Indirect recursion
- 3 Tail recursion
- 4 Non-tail recursion

#### 6. Direct recursion:

### 1 Direct recursion

A function is called direct recursive if it calls the same function again.

Structure of Direct recursion:

```
fun() {
    //some code

    fun();
    //some code
}
```

## 7. Indirect recursion:

2 **Indirect recursion**

A function (let say **fun**) is called **indirect recursive** if it calls another function (let say **fun2**) and then **fun2** calls **fun** directly or indirectly.

Structure of Indirect recursion:

```
fun() {  
    //some code  
  
    fun2();  
  
}  
  
fun2() {  
    //some code  
  
    fun();  
  
}
```

**Program to understand indirect recursion**

WAP to print numbers from 1 to 10 in such a way that when number is odd, add 1 and when number is even, subtract 1.

Output: 2 1 4 3 6 5 8 7 10 9

## 8. Tail recursion:

**DEFINITION**

A recursive function is said to be **tail recursive** if the recursive call is the last thing done by the function. There is no need to keep record of the previous state.

```
void fun(int n) {  
    if(n == 0)  
        return;  
    else  
        printf("%d ", n);  
        return fun(n-1);  
}  
int main() {  
    fun(3);  
    return 0;  
}
```

## 9. Non tail recursive:

**DEFINITION**

A recursive function is said to be **non-tail recursive** if the recursive call is not the last thing done by the function. After returning back, there is some something left to evaluate.

```
void fun(int n) {  
    if(n == 0)  
        return;  
    fun(n-1);  
    printf("%d ", n);  
}  
int main() {  
    fun(3);  
    return 0;  
}
```

**10. Iterative program vs recursive program:**

- i. Sometimes iterative code is too complex to write, hence recursion.

**11. Questions on recursion:**

- i. Write a **recursive program** to find **sum** from **1 to n**:

```
//  
int sum(int n)  
{  
    if(n == 1)  
    {return 1;}  
    else  
    {  
        return n+sum(n-1);  
    }  
}  
  
int main()  
{  
    int n,result;  
    printf("Enter n : ");  
    scanf("%d",&n);  
    result = sum(n);  
    printf("\nResult : %d",result);  
  
}
```

- ii. Print the array using recursion:

```
int arrayprint(int arr[], int size)  
{  
    if(size == 0 )  
    {return 1;}  
    else  
    {  
        printf("\nEnter arr[%d] %d: ",size-1, arr[size-1]);  
        return arrayprint(arr,size-1);  
    }  
}  
  
int main()  
{  
    int n;  
    printf("Enter size : ");  
    scanf("%d",&n);  
    int arr[n];  
  
    for(int i =0; i<n;i++)  
    {  
        printf("\nEnter arr[%d] : ",i);  
        scanf("%d",&arr[i]);  
    }  
  
    arrayprint(arr, n);  
}
```

## Array 1-D:

**What is Array:** Array is a **data structure** that contains **values of same type**.

**DATA TYPE OF ARRAY ELEMENTS**

a	5	6	10	13	56	76	1	2	4	8
b	‘a’	‘b’	‘c’	‘d’	‘e’					
c	‘a’	‘b’	1	5.6	‘e’	34	2	3		

(✓) (✓) (✗)

- ① How to declare and define one dimensional array?
- ② How to access the array elements?
- ③ How to initialize one dimensional array?

### 1. Declaring an array:

**DECLARATION AND DEFINITION OF 1D ARRAY**

Syntax: **data\_type name of the array [no. of elements];**

For example: an array of integers can be declared as follows:

```
int arr[5];
```

arr [ ] [ ] [ ] [ ] [ ]

↓

Compiler will allocate a contiguous block of memory = **5\*sizeof(int)**

### 2. Declaring size of an array: Can be declared using a **positive constant value** or a **constant expression**.

The length of an array can be specified by any **positive integer** constant expression.

```
int arr[5];
```

```
int arr[5+5];
```

```
int arr[5*3];
```

~~int arr[-5];~~

```
int a;  
int arr[a = 21/3];
```

3. **How to define 1-D array:** In method 2, size is defined by the compiler since you've initialized the array already. You can't create an array without specifying the size of it.

INITIALIZING 1D ARRAY	
<b>METHOD 1:</b> <pre>arr[5] = {1, 2, 5, 67, 32};</pre>	<b>METHOD 2:</b> <pre>arr[] = {1, 2, 5, 67, 32};</pre>
<b>METHOD 3:</b> <pre>int arr[5]; arr[0] = 1; arr[1] = 2; arr[2] = 5; arr[3] = 67; arr[4] = 32;</pre>	<b>METHOD 4:</b> <pre>int arr[5]; for(i=0; i&lt;5; i++){     scanf("%d", &amp;arr[i]); }</pre>

4. **Initializing the values in array:**

- By default, the value of an **array** is initialized to **garbage**.
- Partial initializing leads to reset of the elements being filled as **0s**.
- To initialize to all zeros, leave the curly bracket empty.

**Q**

What if number of elements are lesser than the length specified?

```
int arr[10] = {45, 6, 2, 78, 5, 6};
```

**A**

The remaining locations of the array are filled by value 0.

```
int arr[10] = {45, 6, 2, 78, 5, 6, 0, 0, 0, 0};
```

5. What to do if we want something like below?

Sometimes we want something like this:

```
int arr[10] = {1, 0, 0, 0, 0, 2, 3, 0, 0, 0};
```

```
int arr[10] = {[0] = 1, [5] = 2, [6] = 3};
```

This way of initialization is called **designated initialization**.

And each number in the square brackets is said to be a **designator**.

(2.)

No need to bother about the order at all.

```
int a[15] = {[0] = 1, [5] = 2};
```

```
int a[15] = {[5] = 2, [0] = 1};
```

Both are same

## WHAT IF I WON'T MENTION THE LENGTH?

- ★ Designators could be any non-negative integer.
- ★ Compiler will deduce the length of the array from the largest designator in the list.

```
int a[] = {[5] = 90, [20] = 4, [1] = 45, [49] = 78};
```

Because of this designator,  
maximum length of this  
array would be 50.

```
int a[] = {1, 7, 5, [5] = 90, 6, [8] = 4};
```

≡

```
int a[] = {1, 7, 5, 0, 0, 90, 6, 0, 4};
```

But, if there is a clash, then designated initializer will win.

```
int a[] = {1, 2, 3, [2] = 4, [6] = 45};
```

### Questions on Array 1-D:

**Question 1:** Find the number of elements in an array.

```
// To execute C, please define "int main()"  
int main() {  
    int arr[6];  
  
    printf("Size of whole array : %d\n", sizeof(arr));  
    printf("Size of single element in array : %d\n", sizeof(arr[0]));  
  
    printf("Number of elements : %d\n", sizeof(arr)/sizeof(arr[0]));  
    return 0;  
}
```

**Question 2:** Traversing through an array of limit size. Take user input from the user and fill those elements and then print them. Then change it to character array and float array.

```
#define limit 100  
  
int main() {  
    float arr[limit];  
    int n;  
  
    printf("Enter the number of elements: ");  
    scanf("%d", &n);  
  
    for(int i = 0; i<n; i++)  
    {  
        printf("\nEnter arr[%d]: ", i);  
        scanf("%f", &arr[i]);  
    }  
    for(int i = 0; i<n; i++)  
    {  
        printf("\nArr[%d]: %.2f", i, arr[i]);  
    }  
}
```

## Question 2: Insertion in an Array.

### i. Rules for insertion:

- $\text{Array}[\text{limit}-1] = \text{array}[\text{limit}-2]$  ;  $\text{limit} -$
- Till **position +1**;
- Insert new value at **position-1**

```
#define limit 10
int main()
{
    int arr[limit] = {0,1,2,3,4,5,6,7,8,9};
    int position, value;

    //print array
    printf("Array: ");
    for(int i = 0; i<limit; i++)
    {
        printf(" %d ",arr[i]);
    }

    //ask for position and value to insert
    printf("\nEnter the position: \n");
    scanf("%d", &position);

    printf("Enter the value: \n");
    scanf("%d", &value);

    //perform insertion
    for(int i = limit; i>=position+1; i--)
    {
        arr[i-1] = arr[i- 2];
    }

    arr[position -1] = value;

    //print the new array
    printf("Array: ");
    for(int i = 0; i<limit; i++)
    {
        printf(" %d",arr[i]);
    }
}
```

## Question 3: Deletion in an Array.

### ii. Rules for Deletion:

- $\text{Array}[\text{position}-1] = \text{array}[\text{position}]$ ;  $\text{position}++$
- Till **limit**;

```
#define limit 10
int main()
{
    int arr[limit] = {0,1,2,3,4,5,6,7,8,0};
    int position;

    //print array
    printf("Array: ");
    for(int i = 0; i<limit; i++)
    {
        printf(" %d ",arr[i]);
    }

    //ask for position and value to insert
    printf("\nEnter the position: \n");
    scanf("%d", &position);

    //perform deletion
    for(int i = position; i<limit; i++)
    {
        arr[i-1] = arr[i];
    }

    //print the new array
    printf("Array: ");
    for(int i = 0; i<limit; i++)
    {
        printf(" %d",arr[i]);
    }
}
```

**Question 3:** Reverse the elements in an Array. Then reverse first half of an array.

```
#define limit 5

int main() {
    int arr[limit] = {1,2,3,4,5} ;

    //check if its an odd array or even array
    if(limit%2 == 0)
    {
        int center = limit/2;
        int k = 1;
        int m = 0;
        for(int i = 0; i<(limit/2); i++)
        {
            arr[center - k] = arr[center - k] +arr[center+m];
            arr[center+m] = arr[center - k] - arr[center+m];
            arr[center - k] = arr[center - k] -
arr[center+m];
            k++;
            m++;
        }
    }else
    {

        int center = limit/2;
        int k = 1;
        for(int i = 0; i<(limit/2); i++)
        {
            arr[center - k] = arr[center - k] +arr[center+k];
            arr[center+k] = arr[center - k] - arr[center+k];
            arr[center - k] = arr[center - k] -
arr[center+k];
            k++;
        }
    }

    for(int i = 0; i<limit; i++)
    {
        printf("Arr[%d] : %d \n", i, arr[i]);
    }
}
```

**Question 4:** Convert decimal to binary then store it in an array.

```

#include <stdio.h>

// To execute C, please define "int main()"
#define limit 32

int main() {

    int arr[limit] = {};
    int n;
    printf("Enter a number: ");
    scanf("%d",&n);
    int i = 0;
    int count = 0;

    //convert it into binary
    while(n != 0)
    {
        arr[i] = n%2;
        n = n/2;
        i++;
        count++;
    }

    printf("Binary: ");
    for(int i = count-1; i>=0; i--)
    {
        printf(" %d", arr[i]);
    }
}

```

**Question 4:** Print odd and even number separately.

**Question 5:** 2<sup>nd</sup> largest element in array.

```

int main() {

    int limit;
    printf("Enter size: ");
    scanf("%d",&limit);

    int arr[limit];
    for(int i = 0; i<limit; i++)
    {
        printf("\nEnter array[%d] : ",i);
        scanf("%d",&arr[i]);
    }

    int l1,l2 = 0;
    for(int i = 0; i<limit; i++)
    {
        if(arr[i] > l1)
        {
            l2=l1;
            l1 = arr[i];
        }else if(arr[i]<l1 && arr[i]>l2)
        {
            l2 = arr[i];
        }
    }
    printf("2nd largest number : %d ",l2);
}

```

**Question 6:** How many times **each element repeats itself in the array.**

### Multi-dimensional Arrays: 2-D and 3-D:

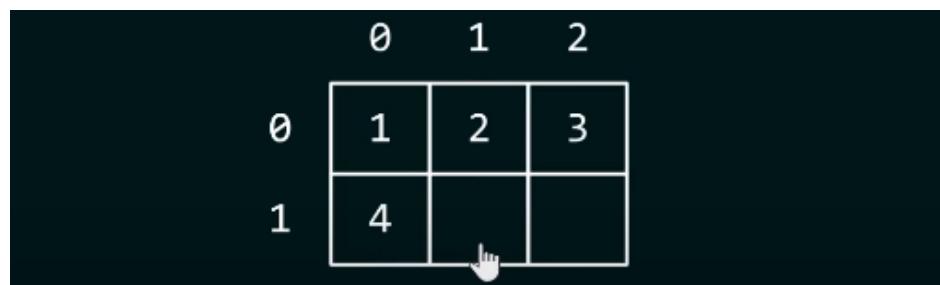
1. **What is a 2D array?** It is an array of array. Ex: `arr[2][3]` = 6 elements in total.
  - i. **Declaration:** datatype name [x][y]. X here being number of **rows** and Y being number of **columns**. **YOU HAVE TO DECLARE THE SIZE.**
  - ii. **How to visualize a 2D array?** Here is a 2D array. Basically, think of it as 4 -1D arrays stacked on top of each other.



- iii. **How to initialize a 2D array?** Initialize them like you will **initialize a normal 1-D array.**

```
int a[2][3] = {1, 2, 3, 4, 5, 6};
```

- iv. **How is a 2D array filled?** Think logically. **Left to right** and then **top to bottom**.



- v. But the last way is confusing. Let's see a better way of initializing a 2D array.

**Method 2:**

Row 1

Row 2

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

- vi. **How is an array filled in this way? Same as last time.**
  - vii. **How to access each element** of a 2D array: `arr[0][1]`.
2. Print each element of a 2-D array:

```

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

int size = sizeof(arr)/sizeof(arr[0][0]);

for(int i = 0; i<rows; i++)
{
    for(int j = 0; j<cols; j++)
    {
        printf("arr[%d][%d]: %d\n", i, j, arr[i][j]);
    }
}

```

3. Multiply two matrices and store the result in the third one:

<pre> #define rows 2 #define cols 3 #define z 2  //2D arrays int main() {      int arr1[rows][cols] = { {1,2,3},                             {1,2,3}                           };      int arr2[rows][cols] = { {1,2,3},                             {1,2,3}                           };      int arr3[rows][cols] = { };      //multiplication:     for(int i = 0; i&lt;rows; i++)     {         for(int j = 0; j&lt;cols; j++)         {             arr3[i][j] = arr1[i][j]*arr2[i][j];             printf("arr3[%d][%d]: %d\n", i, j, arr3[i][j]);         }     } } </pre>	Harsh Dubey ran arr3[0][0]: 1 arr3[0][1]: 4 arr3[0][2]: 9 arr3[1][0]: 1 arr3[1][1]: 4 arr3[1][2]: 9
--	---

4. 3-D array: Basically arr[z][rows][cols]: z denotes number of 2-D arrays.

- i. How to store an element in a 3-D array and then print it.

```

#define rows 2
#define cols 3
#define z 2

//Initializing and printing 3D arrays
int main()
{
    int arr[z][rows][cols] = {};
    //multiplication:
    for(int k = 0; k<z; k++)
    {
        for(int i = 0; i<rows; i++)
        {
            for(int j = 0; j<cols; j++)
            {
                arr[k][i][j] = j;
                printf("arr[%d][%d][%d]: %d\n", k, i, j, arr[k][i]
[j]);
            }
        }
    }

    int a[2][2][3] = {
        {{1, 2, 3}, {4, 5, 6}},
        {{7, 8, 9}, {10, 11, 12}}
    };

```

1	2	3
4	5	6

7	8	9
10	11	12

2 x 3

2 x 3

##### 5. Questions on multi-dimensional arrays: REMEMBER CRRC

- Write a program that will print sum of rows and cols in a 5x5 array.
- Matrix multiplication: Size of the resultant matrix is:
  - m1[rows][cols] x m2[rows][cols] = Matrix [rows of m1] [cols of m2].**
  - How to multiply? Rows x Cols. Then add all of them. Shown below.

1	2	3
1	2	1
3	1	2

3 x 3

1	2	3
1	2	1
3	1	2

3 x 3

1 x 1

1 x 2

3 x 3

##### iii. Rules for matrix multiplication: CRRC

- Cols of 1<sup>st</sup> = rows of 2<sup>nd</sup> must be equal.

- $m1[rows][cols] \times m2[rows][cols] = \text{Matrix } [rows \text{ of } m1] [cols \text{ of } m2].$
- $\text{Matrix } [0][0] = \text{Sum } (m1 \text{ row}0 \times m2 \text{ col}0)$

iv. Write a program that multiplies two matrices:

```
//logic for matrix multiplication
for(int i = 0; i< rows1; i++)
{
    for(int j = 0; j< cols2; j++)
    {
        for(int k = 0; k< cols1; k++)
        {
            matrix[i][j] = matrix[i][j] + (m1[i][k]*m2[k][j]);
        }
    }
}
```

### Constant arrays and variable length arrays:

1. **What's a constant array:** Look up table!! **Read-only table.** Remember from EE347. For sensor calibration.

i. **Why is it called const?** BC once initialized, then it can't be changed. Ex, below.

```
#define rows 3
#define cols 3

int main()
{
    const int lookup[rows] = {70,45,60};
    lookup[0] = 99;
```

solution.c: In function 'main':  
solution.c:18:13: error: assignment of read-only loc  
 lookup[0] = 99;  
 ^  
solution.c:16:13: warning: variable 'lookup' set but  
ut-set-variable]
 const int lookup[rows] = {70,45,60};  
 ^~~~~~

ii. **Look-up table** can be **1D or 2D.**

```
int main()
{
    const int lookup[rows] = {70,45,60};
    int ADC;

    printf("Enter the value of the ADC: ");
    scanf("%d",&ADC);

    printf("Temprature reading on the basis of ADC is:
%d", lookup[ADC]);
```

2. **Variable length array:** Length of the array is specified during the runtime.

- Variable length arrays can't be initialized and declared at the same time.**
- Size can be given in terms of an expression.

## Pointers:

1. **What are pointers? Address**, pointers are ADDRESS also called REFERENCE. Don't think of them as anything else.

- i. What happens when you pass in a pointer? You pass the address.
- ii. What happens when you return a pointer? You return the address.
- iii. Remember: Referencing: &address and dereferencing: \* Value.

2. Declaring and initializing pointers:

- i. **Declaring a pointer:** Specify which data type will it point to. Ex: char\*ptr will point to character and int\*ptr will point to an integer.
- ii. **Defining a pointer:** Specify which address will it point to. **int\* ptr = &x;** Look, we have assigned the **address of x** to ptr.
- iii. **What if try to assign the address of an integer to a char pointer?** We can. And if we dereference it then it will print the 1<sup>st</sup> byte of it. However, it will show a warning.

```
solution.c: In function 'main':  
solution.c:16:14: warning: initialization of 'char *' from incompatible p  
ointer type 'int *' [-Wincompatible-pointer-types]  
    char*ptr = &x;
```

To fix this. Just type cast the pointer into character pointer.

- iv. **What if a float pointer is assigned to an integer pointer:** Will print 0.000000 even though the addressing is incremented by 4 bytes?
- v. **What if a double pointer is assigned to an integer pointer:** Segmentation fault. But not in case of an integer array where 8 or more than 8 bytes of data is available.

```
int x[4] = {1,2,3,4};  
  
double*ptr = (double*)&x;  
printf("%f", ptr[0]);
```

- vi. Can we create a pointer to unsigned values? Since the purpose of

3. **Value of operator or dereferencing operator:**

- i. When you print the pointer, you print the address of the variable the pointer points to.
- ii. To get the value we have to de-reference a pointer using \* operation.

```
int num = 5;  
int *ptr = &num;  
  
printf("Address of num using '&' : %#x\n", &num);  
printf("Address of num using pointer : %#x\n", ptr);  
printf("Value of num by dereferencing : %d\n", *ptr);
```

Output: Address of num using pointer : 0x10cd81e0  
Address of num using pointer : 0x10cd81e0  
Value of num by dereferencing : 5

- iii. **Caution: Do not dereference a pointer** that is not been initialized i.e., a wild pointer.
- iv. What will happen if we **dereference a wild pointer?** Segmentation fault. Wild pointer may point to a memory location, but we don't have a legal right to access that memory location. Hence **segmentation fault**.

4. What is **segmentation fault**: Program trying to illegally read or write to a memory location.

5. Pointer assignment:

- i. **Address assignment:** When you assign one pointer to another then you are assigning the content of one pointer to another i.e., the address. Look at the example below to understand it better.

```
int num = 5;
int *ptr1, *ptr2;
ptr1 = &num;

ptr2 = ptr1;

printf("Address of num using '&' : %#x\n", &num);
printf("Address of num using pointer 1 : %#x\n",
ptr2);
printf("Address of num using pointer 2 : %#x\n",
ptr2);
printf("Value of num by dereferencing ptr2 : %d\n",
*ptr2);
```

solution.c:22:46: warning: format '%x' expects argument of type 'int', but argument 2 has type 'int \*' [-Wformat]
printf("Address of num using pointer 2 : %d\n",

Address of num using '&' : 0xce762340
Address of num using pointer 1 : 0xce762340
Address of num using pointer 2 : 0xce762340
Value of num by dereferencing ptr2 : 5

- ii. **We can also assign the value of one pointer:** `ptr2 = *ptr1`, by dereferencing it.

- iii. **Question:** Just by reading.

**Predict the output of the following program:**

```
int i = 1;
int *p = &i;
q = p;
*q = 5;
printf("%d", *p);
```

6. **Pointers and functions (Pass by value and pass by reference):** Write a function that uses pointers to find the min and max value of the array and returns the value.

- i. **Pass by value vs pass by reference:** Why pass by reference? You can't return two values in C from a function.

- **Pass by value:** You pass in the value of the variables and do computation and come back. **You do not change the value of the variable.**

```
int summation(int a, int b)
{
    return a+b;
}

int main()
{
    int a = 5, b = 6, sum = 0;
    sum = summation(a, b);

    printf("a : %d\n", a);
    printf("b : %d\n", b);
    printf("Sum : %d\n", sum);

}
```

a : 5  
b : 6  
Sum : 11

- **Pass by reference:** You pass in the address of variables. Do computation and return the sum.

```

int summation(int *a, int *b)
{
    return *a+*b;
}

int main()
{
    int a = 5, b = 6, sum =0;
    sum = summation(&a, &b);

    printf("a : %d\n", a);
    printf("b : %d\n", b);
    printf("Sum : %d\n", sum);

}

```

a : 5  
b : 6  
Sum : 11

- **Difference between pass by value and pass by reference:** In pass by reference, you can actually change the content of the register.

```

int summation(int *a, int *b)
{
    *a = *b = 10;
    return *a+*b;
}

int main()
{
    int a = 5, b = 6, sum =0;
    sum = summation(&a, &b);

    printf("a : %d\n", a);
    printf("b : %d\n", b);
    printf("Sum : %d\n", sum);

}

```

a : 10  
b : 10  
Sum : 20

- **Why else use pass by reference?** Try returning 2 values from a function.

```

int summation(int *a, int *b)
{
    *a = *b = 10;
    (*a)++;
    *b = *b+5;
    return 0;
}

int main()
{
    int a = 5, b = 6;
    summation(&a, &b);

    printf("a : %d\n", a);
    printf("b : %d\n", b);
}

```

a : 11  
b : 15

- Another example of changing values by passing them by reference:

```

void minmax(int arr[], unsigned int size, int*min, int*max );
int main()
{
    int arr[] = {1,2,3,4,5,6,7,8,9,100,0};
    int min, max;
    unsigned int size = sizeof(arr)/sizeof(arr[0]);

    minmax(arr, size, &min, &max );
    printf("My min: %d\n", min);
    printf("My max: %d\n", max);

    return 0;
}
void minmax(int arr[], unsigned int size, int*min, int*max )
{
    *min = *max = arr[0];
    for(int i = 1 ; i<size; i++)
    {
        if(*min>arr[i])
            *min = arr[i];
        if(*max<arr[i])
            *max = arr[i];
    }
    return;
}

```

7. **Accepting and Returning pointers from function:** Return the pointer to the midpoint of an array then change the 1<sup>st</sup> byte of it for an integer.

- i. **Accepting:** Accepting a pointer means accepting the address

```

int summation(int *a)
{
    (*a)++;
    return 0;
}

int main()
{
    int a = 5;
    int*ptr =&a;

    summation(ptr);

    printf("a : %d\n", a);
}

```

- ii. **Receiving a pointer:**

```

int* midval(int arr[],int size )
{
    return &arr[size/2];
}
int main()
{
    int arr[] = {300,2,3,300,5,6,7};
    //find give the the address of the mid value
    int *ptr =NULL;
    int size = sizeof(arr)/sizeof(arr[0]);

    ptr = midval(arr,size );
    //print me the value
    printf("Mid value is: %d\n", *ptr);
    printf("Mid value address is: %#x\n", ptr);
    printf("Address of arr is: %#x\n", &arr);

    //take in 2nd byte of mid value and set it to zero
    char *byte = (char*)ptr;
    printf("Value of byte is: %#X\n", *byte);
    *byte = ~*byte;
    printf("Value of byte is: %#X\n", *byte);
    printf("New Mid value is: %d\n", *ptr);

    return 0;
}

```

8. Redo this using **float** and **double** precision. Print out mantissa, exponential and sign bit. For float: 4.25. And the figure out the **endianness of the system**.

- i. **Big endian:** MSByte in the smallest address.

ii. Little endian: LSByte in the smallest address.

```
#include <stdio.h>
//Number is 4.25 and -4.25
double *midpoint(double arr[], unsigned int size)
{
    return &arr[size/2];
}

int main()
{
    double arr[] = {1.2, 2.5, 3.8, -4.25, 5.689, 6.75, 31.14};

    unsigned int size = sizeof(arr)/sizeof(arr[0]);

    double*ptr = NULL;

    ptr = midpoint(arr, size);

    //print
    char *pc = (char*)ptr;
    for(int i = 0; i<8; i++)
    {
        printf("Double precession of -4.25 BYTE[%d] is
%#x\n", i, *(pc+i));
    }
}
```

Harsh Dubey ran 27 lines of C (finished in 1.20s):

```
Double precession of 4.25 BYTE[0] is 0
Double precession of 4.25 BYTE[1] is 0
Double precession of 4.25 BYTE[2] is 0
Double precession of 4.25 BYTE[3] is 0
Double precession of 4.25 BYTE[4] is 0
Double precession of 4.25 BYTE[5] is 0
Double precession of 4.25 BYTE[6] is 0x11
Double precession of 4.25 BYTE[7] is 0x40
```

Harsh Dubey ran 27 lines of C (finished in 696ms):

```
Double precession of -4.25 BYTE[0] is 0
Double precession of -4.25 BYTE[1] is 0
Double precession of -4.25 BYTE[2] is 0
Double precession of -4.25 BYTE[3] is 0
Double precession of -4.25 BYTE[4] is 0
Double precession of -4.25 BYTE[5] is 0
Double precession of -4.25 BYTE[6] is 0x11
Double precession of -4.25 BYTE[7] is 0xfffffff0
```

9. Convert floating point into binary:

```
int main()
{
    float num = 4.25;
    int lv = num;
    float rv = num - lv;

    //converting right value
    unsigned int count = 0;
    int templv = lv;
    while(templv!=0)
    {
        count++;
        templv>>=1;
    }int arrlv[count];
    for(int i = count-1; i>=0; i--)
    {
        arrlv[i] = lv%2;
        lv>>=1;
    }

    //converting left value
    int trunc = 0;
    float temprv = rv;
    unsigned int countrv = 0;
    if(rv>0){

        do{
            temprv = temprv*2;
            countrv++;
            if(temprv!=1){
                trunc = temprv;
                temprv = temprv-trunc;
            }
        }while(temprv!=1);

        int arrrv[countrv];
        for(int i = 0; i<countrv; i++){
            rv = rv*2;
            arrrv[i] = rv;
            rv = rv - arrrv[i];
        }
    }
}
```

10. Questions on pointers:

Question 1: Consider the following two statements

```
int *p = &i;
p = &i;
```

First statement is the declaration and second is simple assignment statement.  
Why isn't in second statement, p is preceded by \* symbol?

Answer:

```

void fun(const int *p)
{
    *p = 0;
}

int main() {
    const int i = 10;
    fun(&i);
    return 0;
}

```

Answer: Error. Assignment to read only location.

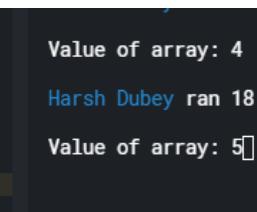
### 11. Pointer arithmetic:

#### i. Adding integers to pointer:

```

int main()
{
    int arr[] = {1,2,3,4,5};
    int *ptr = arr;
    printf("Value of array: %d",*(ptr+4));
}

```



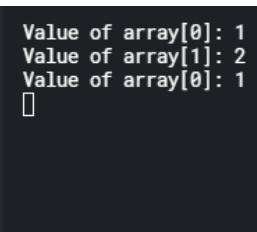
Value of array: 4  
Harsh Dubey ran 18  
Value of array: 5

#### ii. Post increment and pre-increment:

```

int main()
{
    int arr[] = {1,2,3,4,5};
    int *ptr = arr;
    printf("Value of array[0]: %d\n",*(ptr++));
    printf("Value of array[1]: %d\n",*(ptr));
    printf("Value of array[0]: %d\n",*(--ptr));
}

```



Value of array[0]: 1  
Value of array[1]: 2  
Value of array[0]: 1

#### iii. Comparing the pointers: Only possible if both pointers point to the same array? No.

### 12. Find the sum of the elements of an array using pointers:

```

int main()
{
    int arr[] = {1,2,3,4,5,6};

    unsigned size = sizeof(arr)/sizeof(arr[0]);
    int*ptr = &arr[0];
    int sum = 0;

    for(; ptr<=&arr[size-1]; ptr++)
    {sum+=*ptr;}

    printf("Sum: %d", sum);

}

```

### 13. Arrays and pointers: Array's name is just a pointer to the first element of the array.

#### i. Referencing and dereferencing array name: arr[1] = \*(arr + 1)

- ii. **Note:** we can't assign a new address to the pointer pointing to the base of an array.

```
int main()
{
    int arr[] = {1,2,3,4,5,6};

    unsigned size = sizeof(arr)/sizeof(arr[0]);
    int*ptr = arr;
    int sum = 0;

    for(; ptr<=arr + (size-1); ptr++)
    {sum+=*ptr; }

    printf("Sum: %d", sum);

}
```

- iii. **Passing array as an argument to a function:**

- When we pass an array to function, **we are not passing the whole array**, we are **passing the base address** of the array.
- **Question:** why when we pass a variable to function, we don't change its value but when we pass an array, we change its value. Let us see an example: **BC when we pass in array, we pass in base address of the array.**

<pre>void change(int arr[], unsigned size) {     for(int i = 0; i&lt;size; i++)     {         arr[i] = 2*arr[i];     }     size = 2*size; }  int main() {     int arr[] = {1,2,3,4,5,6};     unsigned size = sizeof(arr)/sizeof(arr[0]);      printf("Size: %d\n", size);     change(arr, size);      for(int i = 0; i&lt;size; i++)     {         printf("%d\n", arr[i]);     }     printf("Size: %d", size);  }</pre>	<pre>Size: 6 2 4 6 8 10 12 Size: 6</pre>
---	--

- **Can we pass array elements as values to the function:** Yes, it will behave exactly as pass by value? Look at the example below.

```

void chageup(int a)
{
    a = 2*a;
}

int main()
{
    int a[5] = {1,2,3,4,5};

    chageup(a[1]);
    printf("%d",a[1]);
}

```

2  
Harsh Dubey  
2

- iv. Pointers to 2D arrays: Row major order vs column major order. C stores multidimensional arrays in Row major order.

- What is row major order: It is a convention that C uses to store multidimensional arrays.

**Row major order:** Elements are stored row by row



**Column major order:** Elements are stored column by column



- Reassign the values of a 2D array and then print them by using a single for loop: We can use pointers and exploit the row major order and access the elements of a 2D array in 1D fashion.

```

int main()
{
    int arr[rows][cols] = { {1,2,3},
                           {4,5,6},
                           {7,8,9},
                           };

    //2D arrays in C : row major order
    //{{1,2,3}{4,5,6}{7,8,9}}
    int*ptr = &arr[0][0];

    for(;ptr<= &arr[rows-1][cols-1]; ptr++)
        *ptr = 1;

    ptr = &arr[0][0];
    for(;ptr<= &arr[rows-1][cols-1]; ptr++)
        printf("%d ",*ptr);
}

```

Harsh Dubey ran 32  
1 1 1 1 1 1 1 1  
Harsh Dubey ran 32  
1 1 1 1 1 1 1 1

- Then investigate 3D arrays using pointers: Even 3D array follows row major order.

```

int arr[z][rows][cols] = { {{1,2,3},{4,5,6},{7,8,9}},  

                           {{21,22,23},{24,25,26}},  

                           {27,28,29} }  

};  

//2D arrays in C : row major order  

//{1,2,3}{4,5,6}{7,8,9}  

int*ptr = &arr[0][0][0];  

//ptr = &arr[0][0];  

for(;ptr<= &arr[z-1][rows-1][cols-1]; ptr++)  

    printf("%d ",*ptr);

```

- **Caveat:** Look at the code above and see every time we use **pointers**, we initialize it with the **starting address of the first element of the 2D array**. But why? **Can't we initialize it with just the array name?** Since, that also contains the address of the first element of the array. Explore.
- **Ask yourself what makes a 2D array a 2D array? Or a 3D arrays a 3D array?**  
Aren't they in memory stored the same way a 1D array is? You **add pointers to add dimensions!!**

Harsh Dubey  
1  
□

```

int arr[2][rows][cols] = {{{1,2},{3,4}},  

                           {{5,6},{7,8}}}  

};  

int*ptr = **arr;  

printf("%d \n",*(ptr));

```

#### 14. Questions on arrays and pointers:

i.

Consider the following declaration of two dimensional array in C  
`char a[100][100]`

Assuming that the main memory is byte addressable and that the array is stored starting from the memory address 0, the address of `a[40][50]` is:

- a) 4040  
b) 4050  
c) 5040  
d) 5050

[GATE 2002: 2 Marks]

**Careful.** This could be an integer array too.

ii.

What is the output of the following C code? Assume that the address of x is 2000 (in decimal) and an integer requires four bytes of memory.

```
#include <stdio.h>
int main()
{
    unsigned int x[4][3] = {{1, 2, 3}, {4, 5, 6},
                           {7, 8, 9}, {10, 11, 12}};
    printf("%u, %u, %u", x+3, *(x+3), *(x+2)+3);
}
```

- a) 2036, 2036, 2036
- b) 2012, 4, 2204
- c) 2036, 10, 10
- d) 2012, 4, 6

[GATE 2015 (SET 1)]

**VERY** important.

### 15. Pointer to an entire array:

- i. **Array name and pointer to an array:** Array name is the **pointer** to the 1<sup>st</sup> element.

Example with memory layout. However, it's a self-pointer. That is if we create a **pointer to the array**, and it's not a self-pointer, as shown below.

```
int main()
{
    int a[6] = {1,2,3,4,5,6};
    int*ptr = a;

    //array name
    printf("Address of array pointer: %p \n",a);
    printf("Address of array pointer: %p \n\n",&a);
    //pointer to the array name
    printf("Address of array pointer: %p \n",ptr);
    printf("Address of array pointer: %p \n",&ptr);
```

Address of array pointer: 0x7fff96aba920  
 Address of array pointer: 0x7fff96aba920  
 Address of array pointer: 0x7fff96aba920  
 Address of array pointer: 0x7fff96aba8e0  
 □

106	6	
105	5	
104	4	
103	3	
102	2	
101	1	
Breakage		
101	101	<b>a</b>
51	101	*p

- ii. Let's analyze the 8 results:

```
int main()
{
    int a[6] = {1,2,3,4,5,6};
    int*ptr = a;

    //array name
    printf("1. Address of 'a': %p \n",a);
    printf("2. Address of 'a+1': %p \n",a+1);
    printf("3. Address of '&a': %p \n",&a);
    printf("4. Address of '&a+1': %p \n\n",&a+1);
    //pointer to the array name
    printf("5. Address of 'ptr': %p \n",ptr);
    printf("6. Address of 'ptr+1': %p \n",ptr+1);
    printf("7. Address of '&ptr': %p \n",&ptr);
    printf("8. Address of '&ptr+1': %p \n\n",&ptr+1);
```

1. Address of 'a': 0x7ffc45ac94c0  
 2. Address of 'a+1': 0x7ffc45ac94c4  
 3. Address of '&a': 0x7ffc45ac94c0  
 4. Address of '&a+1': 0x7ffc45ac94d8  
 5. Address of 'ptr': 0x7ffc45ac94c0  
 6. Address of 'ptr+1': 0x7ffc45ac94c4  
 7. Address of '&ptr': 0x7ffc45ac9480  
 8. Address of '&ptr+1': 0x7ffc45ac9488  
 □

- a) 'a' i.e. array name, is an integer pointer to the 1<sup>st</sup> element of the integer array.

So, 'a+1' will be address of a + 4 bytes.

- b) 'a+1' is address of a+ 4 bytes.
- c) **Address of '&a' is same as 'a'**: Self pointer. However, they differ in property.
- d) '&a' when incremented with '+1' it increments with **24 bytes**. Because 'a' is a **integer pointer** and '&a' is a **(\*)[size] pointer**. We will analyze a **(\*)[size] pointer** later.
- e) 'ptr' contains the value of 'a'. Which is the **address of the 1<sup>st</sup> element**.
- f) Just like 'a', 'ptr' is an **integer pointer**. Hence, **incremented by one byte**.
- g) Address of 'ptr', in this case **it's not a self-pointer** hence different address than content of 'ptr'.
- h) **IMPORTANT**. To the **address of any pointer**, if added 1, will **move 8 bytes** if it's a **64 bits machine**. Hence, pointers are very useful when it comes to knowing the machine.

- iii. **(\*)[size] pointers**: Create a **Double Pointer** to whatever it points to. Image below will prove that.

```
int x = 12316757;
unsigned char(*ptr)[2] = (unsigned char(*)[2])&x;

//character pointer to an integer using (*)[size]
operator
printf("1. '&x': %p\n", &x);
printf("2. '&ptr': %p\n", &ptr);
printf("3. 'ptr': %p\n", ptr);
printf("4. '*ptr': %p\n", *ptr);
printf("5. '**ptr': %d\n", **ptr);
```

1. '&x': 0x7ffd40727c10
2. '&ptr': 0x7ffd40727c90
3. 'ptr': 0x7ffd40727c10
4. '*ptr': 0x7ffd40727c10
5. '**ptr': 85

- a. '&x' will give the address of x.
- b. '&ptr' will give address of ptr.
- c. **Address of '&ptr' and content of 'ptr' is different. IMPORTANT.**
- d. '\*ptr' != 'ptr', NOT EQUAL. Even though the address is same, but behavior is different.
- e. You dereference again and you get the value stored in the memory.

- iv. **(\*)[size] pointers in case of array name**: In case of array name, **(\*)[size]** pointer is a **self-pointer**. Look above, everything is same, except C. Address of '&arr' will be equal to the content of 'arr'. Example, below.

```
int main() {
    int arr[9] = {1,2,3,4,5,6,7,8,9};

    printf("'%&arr' : %p\n", &arr);
    printf("'%arr' : %p\n", arr);
```

Harsh Dubey ran 49 lines
'&arr' : 0x7ffeaa146a00
'arr' : 0x7ffeaa146a00

- v. **Make a 1D array into a 2D array**:

```
#define rows 3
#define cols 3

int main() {
    int arr[9] = {1,2,3,4,5,6,7,8,9};

    //changing this to 3x3 2D array
    int(*ptr)[3] = (int(*)[3])&arr;
    for(int i = 0; i< rows; i++)
    {
        for(int j = 0; j<cols; j++)
        {
            printf("%d ",ptr[i][j]);
        }
    }
}
```

1 2 3 4 5 6 7 8 9
-------------------

## 16. Size of pointer to array:

- i. **Sizeof()**: returns **long unsigned int** or **size\_t**. Run the statement below and you'll get answer as 8 bytes. Because it returns **size\_t** and that's an **unsigned long integer**.

```
// size_t  
printf("%llu\n", sizeof(sizeof(int)));
```

- a. **Let's investigate array pointer and size of operator:** Below program is very important, I'll explain it in the next bullet point.

```
int main() {  
    int arr[9] = {1,2,3,4,5,6,7,8,9};  
    //changing this to 3x3 2D array  
  
    printf("Size of array: %lu\n", sizeof(arr));  
    printf("Size of array &arr: %lu\n", sizeof(&arr));  
    printf("Size of array *&arr: %lu\n", sizeof(*&arr));
```

Size of array: 36  
Size of array &arr: 8  
Size of array \*&arr: 36  
[]

- b. **Dereferencing the pointer below shows the true size of the pointer:**

```
int main() {  
    int arr[9] = {1,2,3,4,5,6,7,8,9};  
    //changing this to 3x3 2D array  
    int (*ptr)[3] = (int(*)[3])&arr;  
    printf("Size of 'ptr': %lu\n", sizeof(ptr));  
    printf("Size of '&ptr': %lu\n", sizeof(&ptr));  
    printf("Size of '*ptr': %lu\n", sizeof(*ptr));
```

Size of 'ptr': 8  
Size of '&ptr': 8  
Size of '\*ptr': 12  
[]

- ii. **Array subscript operator:** Why is 4. Working? Because **addition is commutative**.

**\*(arr+1) = \*(1+arr)**

```
int main() {  
    int arr[9] = {10,2,3,4,5,6,7,8,9};  
  
    printf("1. '*arr' : %d\n", *arr);  
    printf("2. '(*arr+0)' : %d\n", *(arr+0));  
    printf("3. 'arr[0]' : %d\n", arr[0]);  
    printf("4. '0[arr]' : %d\n", 0[arr]);
```

1. '\*arr' : 10  
2. '(\*arr+0)' : 10  
3. 'arr[0]' : 10  
4. '0[arr]' : 10  
[]

## 17. Kinds of pointer:

- i. **Void pointer:** Property less pointer. There is **no referencing property** and **no dereferencing property**. You must **type cast a property to it** before referencing or dereferencing.

As shown below, referencing property of void pointer is to refer memory byte by byte.

```
int a = 300;  
void*ptr = &a;  
  
printf("%p\n", ptr);  
printf("%p", ptr+1);
```

0x7ffd8f20d540  
0x7ffd8f20d541

However, you can't de-refer it.

```
int a = 300;  
void*ptr = &a;  
  
printf("%p\n", *ptr);  
printf("%p", *(ptr+1));
```

solution.c:11:16: error: invalid use of void expression  
solution.c:12:14: warning: dereferencing 'void \*' po  
printf("%p", \*(ptr+1));  
^~~~~~  
solution.c:12:14: error: invalid use of void expression

Why use **void pointer**? That's what malloc returns.

- ii. **Null pointers:** Doesn't point to any **memory location** or **points to an invalid memory location**. **NULL** in case of pointers is 0. Don't confuse it with the integer 0.

We can assign a pointer to NULL, to be safe.

```
int*ptr = NULL;                                (nil)  
printf("%p\n", ptr);
```

Why you use NULL? BC that's what malloc returns.

- iii. **Dangling pointers:** Pointer that **points to a non-existing memory location**.

In this example, we have a pointer and we allocated memory and \*ptr has the memory address of the 1<sup>st</sup> byte. Then we free the memory and \*ptr becomes a dangling pointer because it still points to an invalid memory location.

```
int main()  
{  
    int *ptr = (int *)malloc(sizeof(int));  
    ...  
    ...  
    free(ptr);  
    ptr = NULL;  
    return 0;  
}
```

Now, ptr is no more dangling.

In this example, we have automatic deallocation of memory. Remember automatic variable, they get automatically destroyed. Hence, the pointer becomes a dangling pointer.

```
int* fun()  
{  
    int num = 10;  
    return &num;  
}  
  
int main() {  
    int *ptr = NULL;  
    ptr = fun();  
    printf("%d", *ptr);  
    return 0;  
}
```

OUTPUT: Segmentation fault

- iv. **Wild pointers:** No address assigned to the pointer.

```
int main()  
{  
    int *p;  
    *p = 10;  
    return 0;  
}
```

## 18. Memory manipulations using pointers and arrays: #include<string.h>

- i. **Memcmp():** It compares memory. How does it compare memory? **Byte by byte**. The result of comparison is the %ld. **Int x = memcmp( a, b, sizeof(a));**
- If (x == 0): Arrays are same.
  - If(x > 0): a[] > b[]
  - If(x < 0): a[] < b[]

The third parameter is **sizeof()**, takes in the number of bytes to be compared. Then compare BYTE BY BYTE. Lower order BYTE 1<sup>st</sup>.

```
int main() {
    int a = 300;
    char b = 44;
    long int x = 0;

    x = memcmp(&a,&b, sizeof(char));

    if(x == 0)
        printf("Array are same!!");
    else if (x > 0)
        printf("B is less than A!!");
    else if (x < 0)
        printf("A is less than B!!");
}
```

Array are same!!

That means given any kind of data set, we can compare it and see what's missing and what's not. We can use memcmp() to show that **integer and float are actually different**.

```
int a = 300;
float b = 300;
long int x = 0;

x = memcmp(&a,&b, sizeof(float));

if(x == 0)
    printf("Array are same!!");
else if (x > 0)
    printf("B is less than A!!");
else if (x < 0)
    printf("A is less than B!!");

```

B is less than A!!  
Harsh Dubey ran 28 1  
B is less than A!!

Let's make our own memcmp function:

```
int memcompare(void*ptr1, void*ptr2, long unsigned int size)
{
    char*byte1 = (char*)ptr1;
    char*byte2 = (char*)ptr2;

    for(int i = 0; i < size; i++)
    {
        if(i[byte1] == i[byte2])
        {
            i++;
            continue;
        }
        if(i[byte1] > i[byte2])
            return 1;
        if(i[byte1] < i[byte2])
            return -1;
    }
    return 0;
}

int main() {

    int a = 300;
    char b = 44;

    int x = memcompare(&a,&b, sizeof(b));

    if(x == 0)
        printf("Memory Equal!!\n");
    else if(x > 0)
        printf("A > B!!\n");
    else
        printf("A < B!!\n");
}
```

- ii. **Memcpy():** It copies a **chunk of memory** from **destination to source**. Syntax:

```
memcpy(void*destination, void*source, Quantity);
```

**Let's make our own memcpy function.**

```
void memcpy(void*ptr1, void*ptr2, long unsigned int size)
{
    char*byted = (char*)ptr1;
    char*bytes = (char*)ptr2;

    for(int i = 0; i < size; i++)
    {
        i[byted] = i[bytes];
    }
}

int main() {

    int a[] = {1,2,3,4,5};
    int b[] = {300,302,303};

    memcpy(&a,&b, sizeof(char));

    for(int i = 0; i < sizeof(a)/sizeof(a[0]); i++)
    {
        printf("%d ", a[i]);
    }
}
```

```
1 2 3 4 5  
1 5 8 4 5  
Harsh Dubey ran  
300 302 303 4 5  
Harsh Dubey ran  
44 2 3 4 5  
Harsh Dubey ran
```

- iii. **Memset():** It takes a **chunk of memory** and **sets it to a desired value byte by byte**.

**Syntax:** memset(arr1, integer value to set it to, size\_t )

**Let's make our own upgraded memset() function. In which now you can index the byte you want to change.**

```
int memsetx(void*ptr, int value, unsigned long size,
int index )
{
    char*byte = (char*)ptr;

    //check for overflow and underflow
    if(value > 127)
        return 1;
    if(value < -128)
        return -1;
    //
    for(int i = 0; i < size; i++)
    {
        i[byte+index] = value;
    }
    return 0;
}

int main() {

    int a[] = {300,1,2}, x = 0;

    x = memsetx(&a,-9, 1, 4 );

    if(x == -1)
        printf("Underflow!! ");
    else if(x == 1)
        printf("Overflow!! ");
    else
        printf("Successful!!\n");

    printf("A: %d ", a[1] );
}
```

```
Successful!!
A: 247
```

In the code above, look at the **answer carefully**. We changed the byte one of 2<sup>nd</sup> element of an array to -9. Why is the answer 247 then? Because we only changed the 1<sup>st</sup> byte. We need sign extension.

- iv. **Memchr:** Goes to the chunk of the memory and finds the byte that contains the specified value. And returns the pointer to that byte. If byte is not found, then it returns **NULL**.

**Syntax:** `memchr(address of data, value you are searching for, number of bytes you want to check )`

The C library function **void \*memchr(const void \*str, int c, size\_t n)** searches for the first occurrence of the character **c** (an unsigned char) in the first **n** bytes of the string pointed to, by the argument **str**.

**Let's make our own memchar()**

## 19. Pointers and Data Structures:

- i. **Stack using pointers:** Grows up. Last element empty.

```
#define limit 10
int stack[limit];
int*stackptr = &stack[0];

void push(int val)
{
    if(stackptr == &stack[limit-1])
    {
        fprintf(stderr, "Stack full !\n");
        return;
    }
    *(stackptr++) = val;
}

int pop(){
    if(stackptr == &stack[0])
    {
        fprintf(stderr, "Stack empty !\n");
        return -1;
    }
    return *(--stackptr);
}

int main()
{
    for(int i = 0; i < limit-1; i++){
        push(i);
    }

    for(int i = 0; i < 15; i++){
        printf("Stack: %d \n", pop());
    }
}
```

```
Stack: 4
Stack: 3
Stack: 2
Stack: 1
Stack: 0
Stack empty !
Stack: -1
[]
```

## 20. Queue using pointers: pushptr and popptr

- i. Pushptr > popptr:
- ii. Pusptr reaches the top:
- iii. Pushptr == popptr : we can't pop
- iv. Popptr reaches the top
- v. Popptr > pushptr:

```
#define limit 10
int queue[10];
int *pushptr = queue;
int *popptr = queue;
//endcases
int *endofqueue = &queue[limit-1];
int *reset = &queue[0];

void push(int val)
{
    //end condition of pusptr: reset the pushptr to address 0 and only push if
    //atleast 2 memory location available: one to store data and other for pointer to point to
    if(pushptr == endofqueue)
    {
        if(popptr > &queue[1])
        {
            pushptr = reset;
            *(pushptr++) = val;
            return ;
        }
        //if nothing is popped
        fprintf(stderr, "Queue is full !\n");
        return;
    }
    //if push is smaller than pop: only push if memory is available for complete push
    // i.e. atleast 2 memory location available: one to store data
    //and other for pointer to point at
    if(pushptr < popptr)
    {
        if(pushptr == (popptr -1))
        {
            fprintf(stderr, "Queue is full !\n");
            return;
        }
        else
        {
            *(pushptr++) = val;
            return;
        }
    }
    //default condition
    *(pushptr++) = val;
    return;
}
```

## Strings:

1. **What are pointers? Address**, pointers are **ADDRESS** also called **REFERENCE**. Don't think of them as anything else.

## Struct: Data Structure

**What kind of data structure is a struct:** It's your kind of data structure?

1. **Data type:** Are one unit of data structure.
2. **Arrays:** are multiple units of same data type joint together and can be indexed using a pointer.
3. But what if I want to **create a data structure that can store multiple different data types:** use **struct**. What if we **want to return two or more same or different value from a function?** Use **structs**.

```
//typedef
struct harsh{
    int x;
    double y;
    char c;
};

int main()
{
    struct harsh p;

    p.x = 10;
    p.c = 'a';
    p.y = 20.22;

    printf("int: %d, double:%.2lf, char: %c\n",
p.x,p.y,p.c);

}
```

4. How to declare and define the struct at the same time as other data types:

- Caveat: Initialize the data type in the same order as in definition of struct, or else undefined behavior.

```
//typedef
struct harsh{
    int x;
    double y;
    char c;
};

int main()
{
    struct harsh p ={ 
        p.x = 10,
        p.y = 20.22,
        p.c = 'a'

    };

    printf("int: %d, double:%.2lf, char: %c\n",p.x,p.y,p.c);

}
```

5. How to use it like other data types:

```

//typedef
typedef struct harsh{
    char c;
    int x;
    double y;

}harsh;

int main()
{
    harsh p ={
        p.c = 'a',
        p.y = 20.22,
        p.x = 10
    };

    printf("int: %d, double:%.2lf, char: %c\n",p.x,p.y,p.c);
}

```

i. We can also declare them individually without the order restriction.

```

int main()
{
    harsh p;
    p.x = 10;
    p.c = 'a';
    p.y = 20.22;

    printf("int: %d, double:%.2lf, char: %c\n",p.x,p.y,p.c);
}

```

6. What is type def doing: Let's break down the structure.

i. If we look at **typedef structure**, then we can see that it is **typedef – datatype – rename**.

```

typedef struct point
{
    int x;
    double y;
    char c;
}

```

ii. Basically, we can rename any other datatype.

<pre> typedef int harsh; int main() {     harsh x = 10;      printf("int: %d",x); } </pre>	Harsh Dubey int: 10
--	------------------------

**Question:** Check if you can declare a structure as static.

**Question:** Pointers to struct.

7. **How to pass in structure to a function:** Why? We want to return two or more values from the function.

### Link List:

What are Link Lists:

```
typedef struct node{
    int x;
    struct node*next;
    struct node*base;
}node;

int main()
{
    node root;
    root.x = -1;
    root.next = malloc(sizeof(node));
    root.base = root.next;
    root.next->x = -2;
    root.next->next = malloc(sizeof(node));
    root.next->base = root.next->next;
    root.next->next->x = -3;
    root.next->next->next = NULL;
    root.next->next->base = NULL;

    node*curr = &root;
    while(curr!=NULL)
    {
        printf("%d\n", curr->x);
        curr = curr->next;
    }
    free(root.next->next);
    free(root.next);

}
```

## C Interview questions:

1. **Null pointer:** Pointer that points to nothing. Why we need it? So that there is no garbage value assigned to our pointers.
2. **Void pointer and limitations of void pointer:** It can point to any data type. Ex, can hold address of integer, float, char. But can't be used without doing type casting.
3. **Interrupt latency and how to reduce interrupt latency.:** Time b/w last instruction of interrupted task and first instruction of ISR or interrupt handler.
4. **What is interrupt dispatch latency?**
5. **Volatile keyword.**
6. **Can we make a pointer volatile?**
7. **Printf and malloc inside an ISR?**
8. **Can we use breakpoint in an ISR?** You can surely debug the code using it but no guarantee for real time performances.
9. **Nested interrupt? Interrupt inside another interrupt. High in low.**
10. **What is embedded C?**
11. **What are real time systems and RTOS?** Logical correctness of the system within the time constraints. OS that can support real time operations.
12. **Types of RTOS:** Hard (Air bags), Firm (Quality decreases) **and soft** (can miss the deadline).
13. **Features of RTOS:**
14. **RTOS overhead time:**
15. **Inline function: C++ not in C.**
16. **Static variable:** Block and Automatic variable
17. **Why can't we debug the static function?** They become inline since the compiler knows more about them and hence becomes really hard to debug.
18. **Priority inversion.**
19. **IPC mechanism:** Pipes, FIFO, Message-queue, shared memory, common files, socket
20. **FIFO vs PIPE:**
  - III. Inter process communication vs Communication between the processes that have a same ancestor.
  - IV. Can independently read and write, read and write happen at the same time.
  - V. Can give u control and ownership, no control or ownership.
  - VI. Half duplex vs simplex.
  - VII. Can exists after process dies and can't exist after process dies.
21. ## operator or Token pasting: takes two arguments and concatenate two arguments together.
22. **Forward referencing wrt to pointers:** Memory for pointer is created but it won't know which data type it will point to.
23. **How to code infinite loop:** while (1) and for (;;)
24. **What is static linking:** Linker will link all the libraries and link create an executable image. This can port across system where the libraries might not be present.
25. **What is Dynamic linking:** You just have the name of the libraries and they are linked when the execution starts.
26. **Wild pointer:** Pointer which is not initialized.

- 27. Dangling pointer:** Pointing to object that doesn't exist.
- 28. Near pointer:** Bit address up to 16 bits only.
- 29. Far pointer: 32 bits** or typically a pointer that can access information outside a given memory segment.
- 30. Register storage class:**
- 31. Malloc() and calloc():** Malloc() is used to allocate one block of memory and calloc is used to allocate multiple blocks of memory.
- 32. When to use realloc() and any specific condition available for the same:** When you want to reallocate the memory but keep the data intact.
- 33. Why segmentation fault happens:** Memory violations: Stack overflow, try to write to read-only memory, improper use of scanf.
- 34. Segmentation fault vs core dump:** Core dump is a file that is written when a program crashes or a segmentation fault happens. So that the programmer.

## Dynamic memory allocation:

**Malloc and realloc:** Malloc allocates a **chunk of memory** and **returns void pointer** to the starting address of memory location. Why void pointer? Bc it doesn't know what it's pointing to.

```
int *ptr = (int*)malloc(4)
```

```
int *ptr = (int *)malloc(sizeof(int));
```

```
ptr = (int *)realloc(ptr, 2*sizeof(int));
```

 This will allocate memory space of  $2 * \text{sizeof}(\text{int})$ .

 Also, this function moves the contents of the old block to a new block and the data of the old block is not lost.

 We may lose the data when the new size is smaller than the old size.

 Newly allocated bytes are uninitialized.

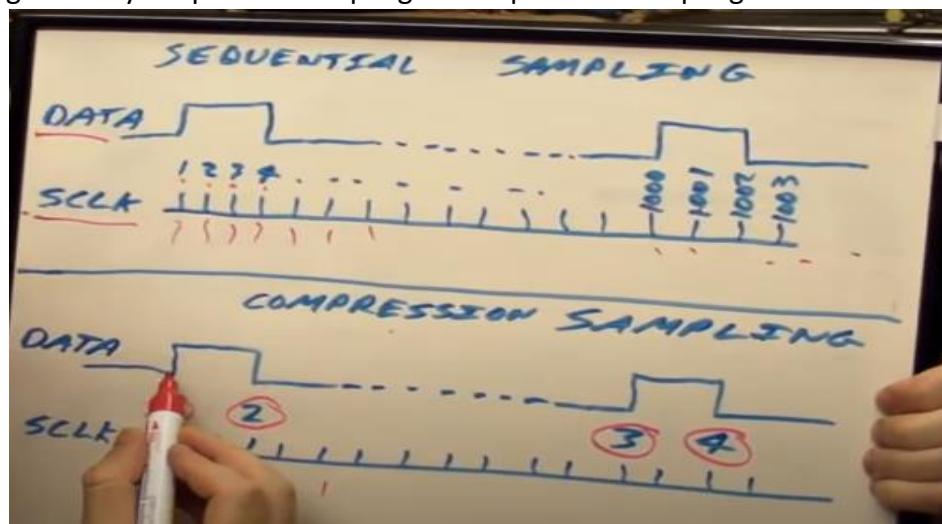
```
int main()
{
    int *ptr = (int *)malloc(4*sizeof(int));
    ...
    free(ptr);
}
```

## Oscope basics:

1. **Edge Trigger:** Trigger when to collect the data. You can bring the trigger knob down to stop the running waveform or you can trigger on rising or falling edge.
  2. **Pulse trigger:** Trigger if the pulse is less than, equal to , or greater than the entered pulse width.
  3. **Mode:** Auto and normal.
  4. **Auto mode:** The oscilloscope will trigger even though it's not in triggering setting. (knob is not on the wave form)
  5. **Normal mode:** Last captured waveform.

## Logic analyzer:

- 12.** Hate logic analyzer: Doesn't show what's actually happening.
  - 13.** Timing analysis vs State analysis.
  - 14.** Timing analysis: Have to provide internal clock. Asynchronous mode.
  - 15.** State analysis: Have to provide external clock. Synchronous mode.
  - 16.** Sampling memory: sequential sampling vs compression sampling.



17.

## **Take home challenge:**

### **1. Readme file:**

- i. **Contributors:**
- ii. **Brief description and Introduction:**
- iii. **Planning:** Whatever planning went before the actual code.
- iv. **Technologies used for the code:**
- v. How to do running and debugging.
- vi. Future improvement. What if there was more time?

### **2. Unit test and integration test:**