# C programming and Data Structures

# Kernighan and Ritchie 2nd 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 who's 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. |
| ULONG_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**?

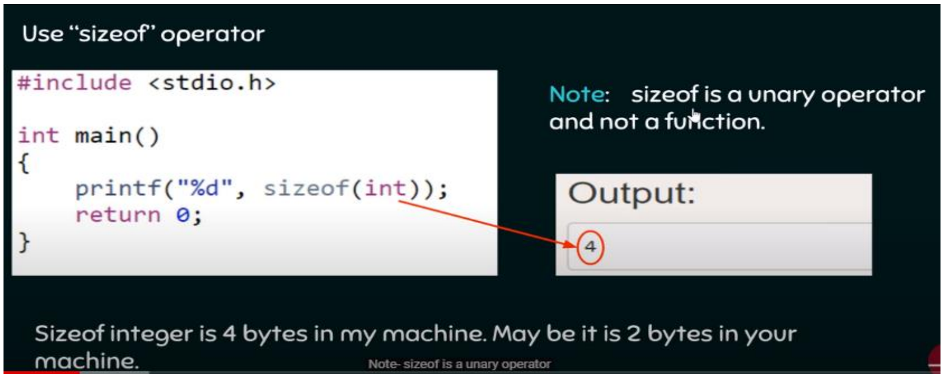
**Question: 2.1:**

**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;
}
```

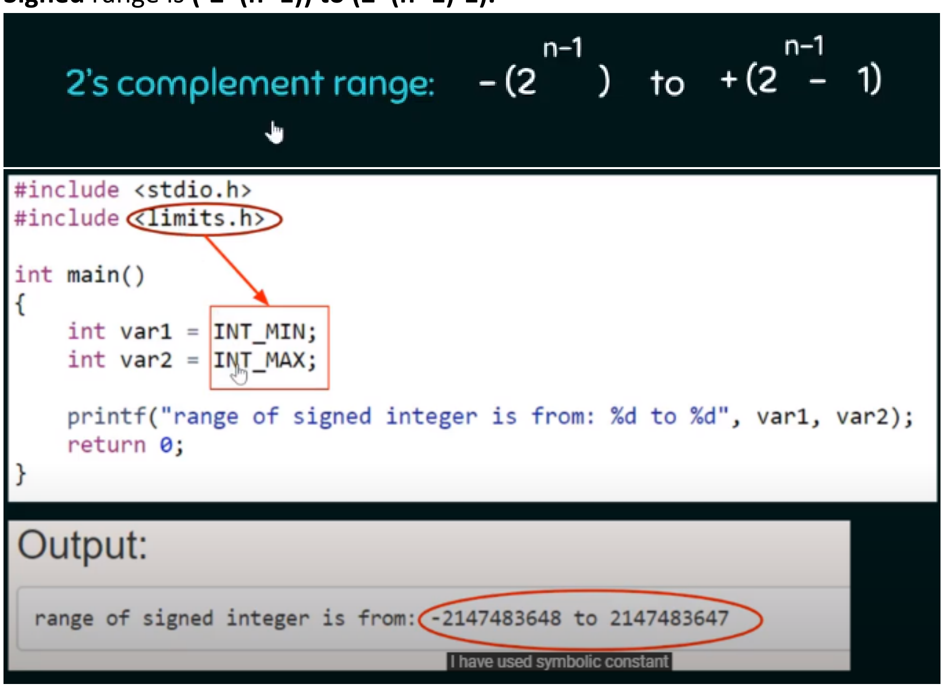
# Integer:

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




4. Find **range using <limits.h> .**
   - **Unsigned** range is **0 to ((2^n) -1).**


   - **Signed** range is **(-2^(n -1)) to (2^(n -1)-1).**



5. Print using **%d** for **signed** and for **unsigned** print using **%u.**
6. 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
```

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

**SUMMARY**

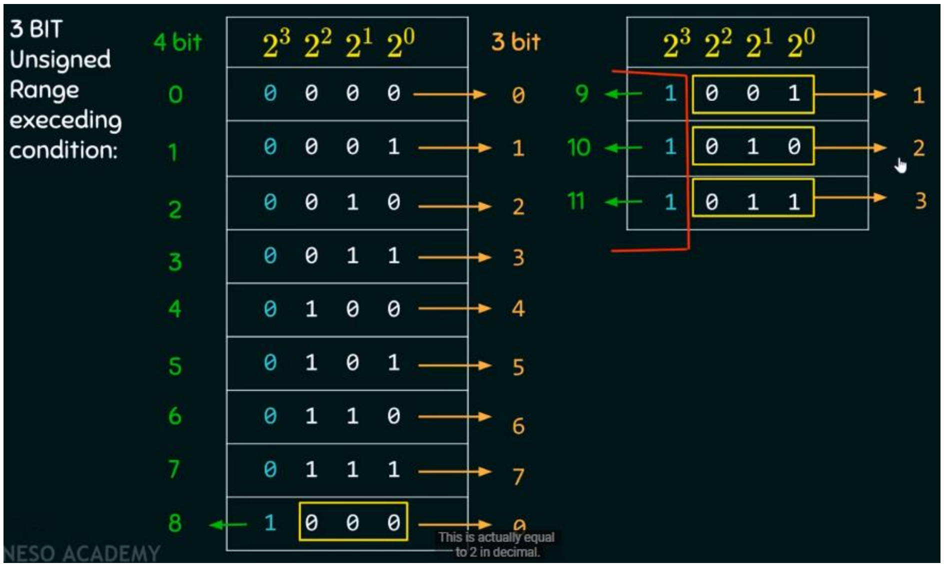
1. sizeof (short) <= sizeof (int) <= sizeof (long).
2. Writing signed int some_variable_name; is equivalent to writing int some_variable_name;
3. %d is used to print "signed integer"
4. %u is used to print "unsigned integer"
5. %ld is used to print "long integer" equivalent to "signed long integer"
6. %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"

## Character:

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

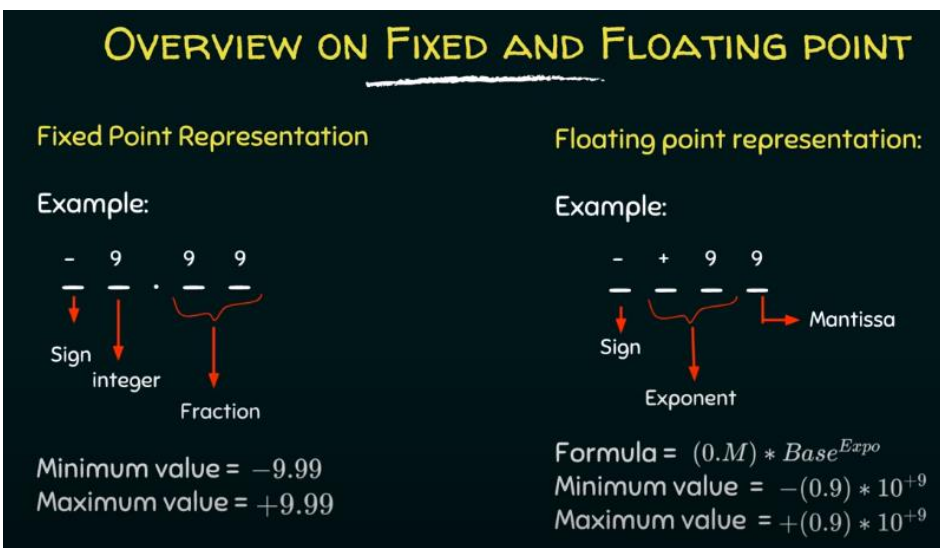

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |

1. 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.
2. Difference between **unsigned char and signed char**

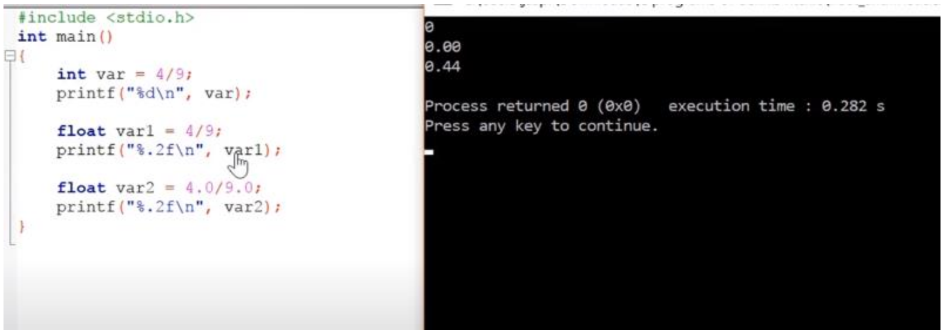



## Float, Double and long double:

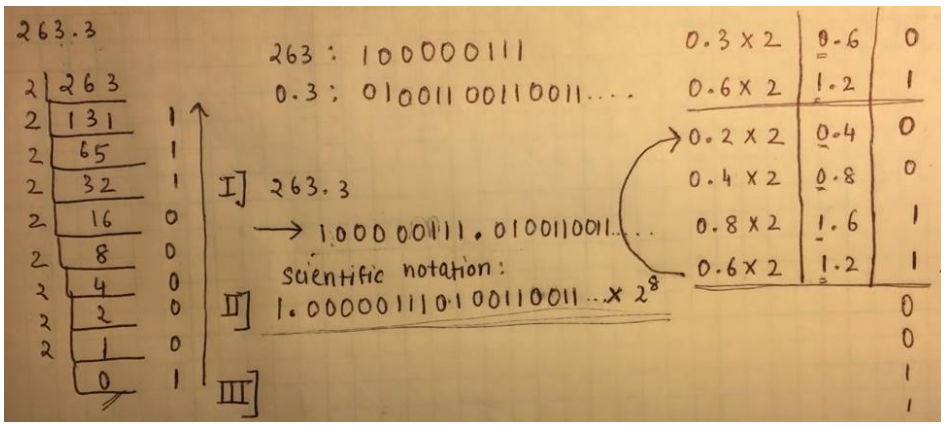
1. **Float: 4 bytes**; use %f to print it; by default %f and lf will print 7 digits. **%.16f** will print it upto 16 decimal places; however, float can only print upto 7 digits starting from before the decimal point.




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




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.

13.

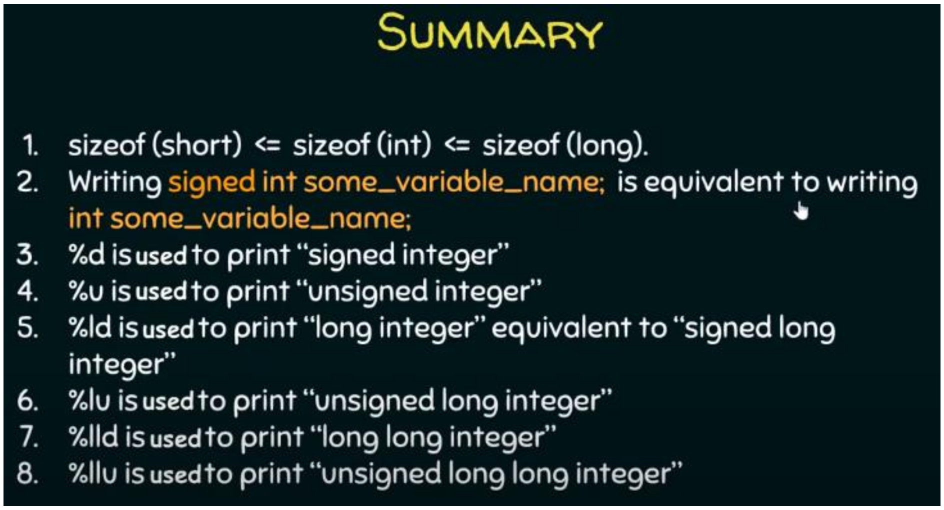
```
//Range of signed integer -2^31 to 2^31 - 1

  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);


}
```

```
  long int num;

  printf("Enter a signed number: \n");
  scanf("%ld", &num);


  //count the number of digits
  long 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
  long 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 : %ld", finalnum);

}
```

14.

```
//prototype
long int reversenum(long int );

int main()

{
  long int num, rnum;
  printf("Enter num: ");
  scanf("%ld",&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;}

}
```

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

16. 


**17.**

**18.**

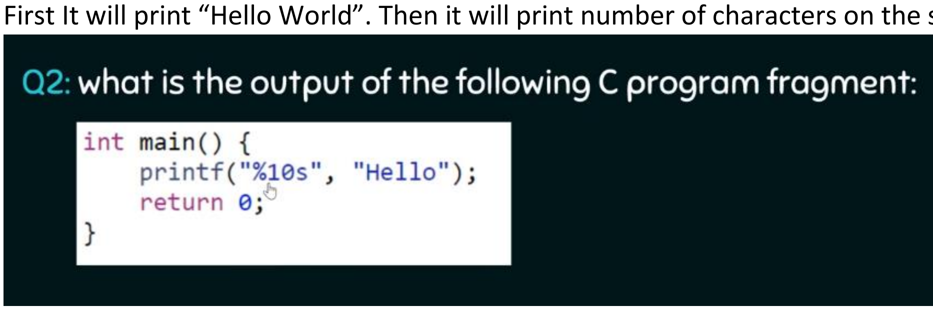
**19.**

**20.**

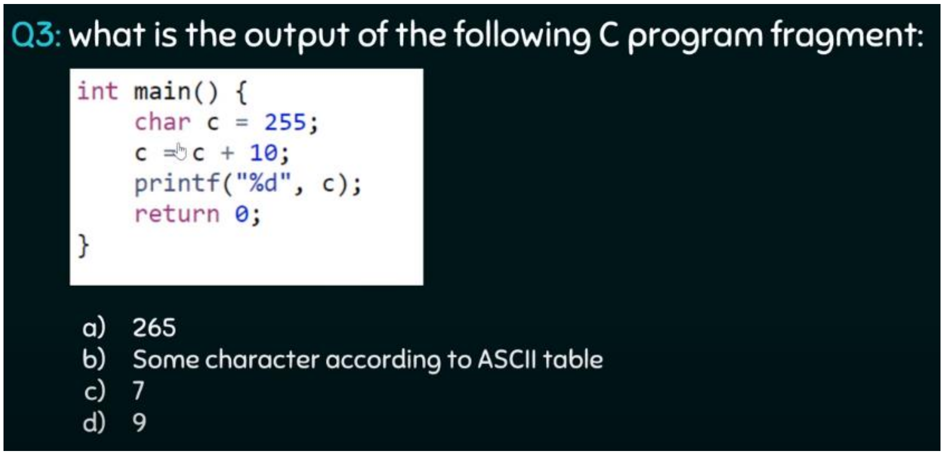
21. **Questions on datatype:** Neso Academy

22. 

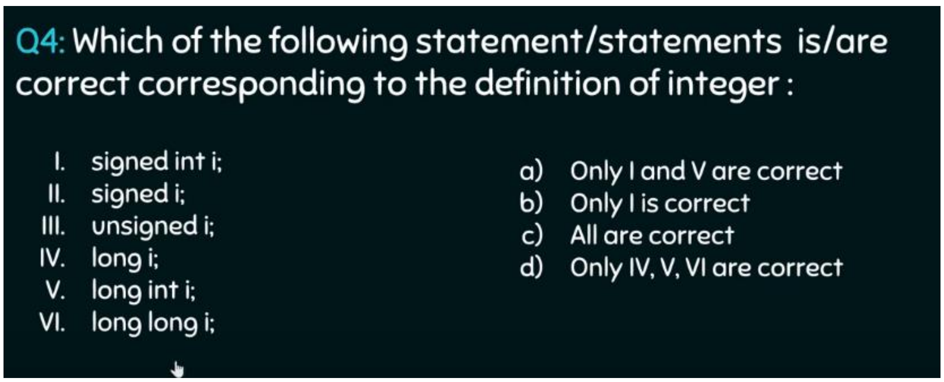

23. **Answer:** First It will print "Hello World". Then it will print number of characters on the screen.

24. 

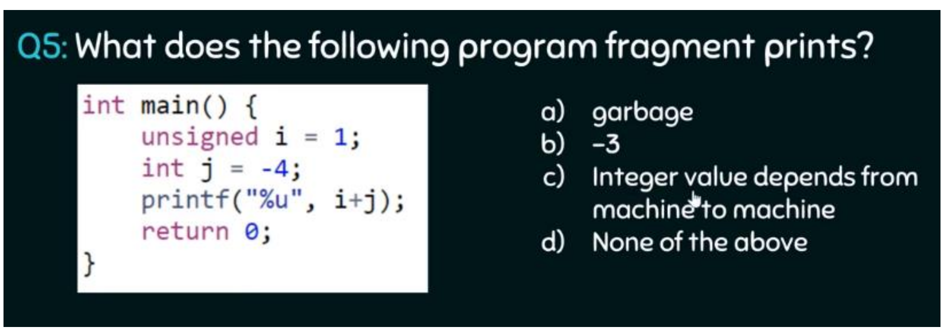

25. **Answer:** %10s means, print up to 10 characters wide. Therefore, 5 spaces+ Hello = 5

26. 


27. **Answer:** 9

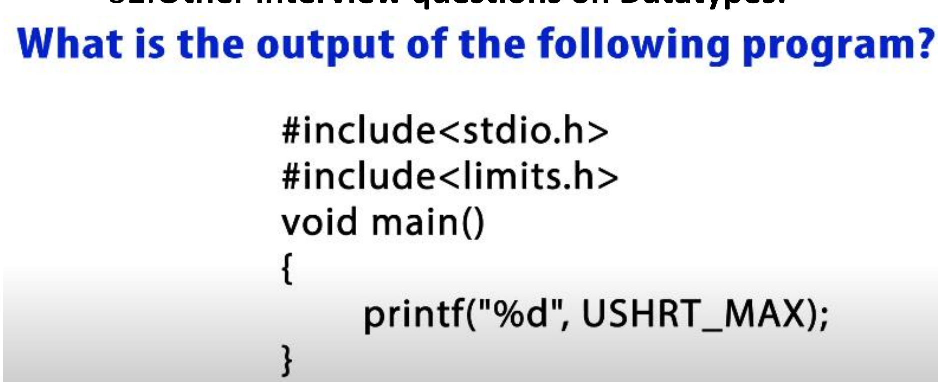
28. 


29. **Answer:** All are correct

30. 


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

32. **Other interview questions on Datatypes:**

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); }
```
**42.** a) 173  b)173.243  c)compile error  d)none

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
44. d) None of these

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 interger 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.

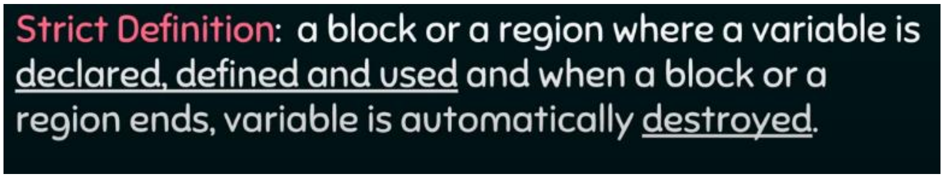
```
int main()
{
  int a = 3, b = 2;

  printf("%lf",(double)a/b);

}
```

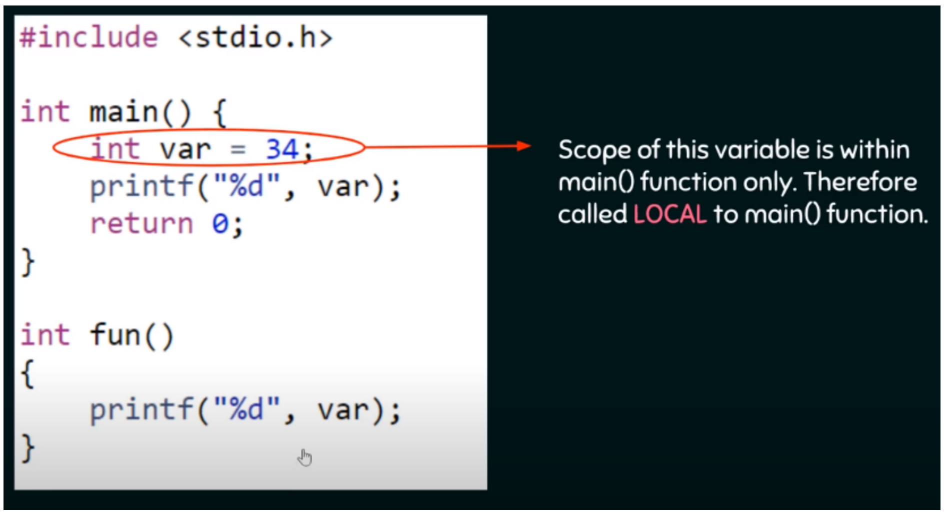
```
Harsh Dubey

1.500000
```

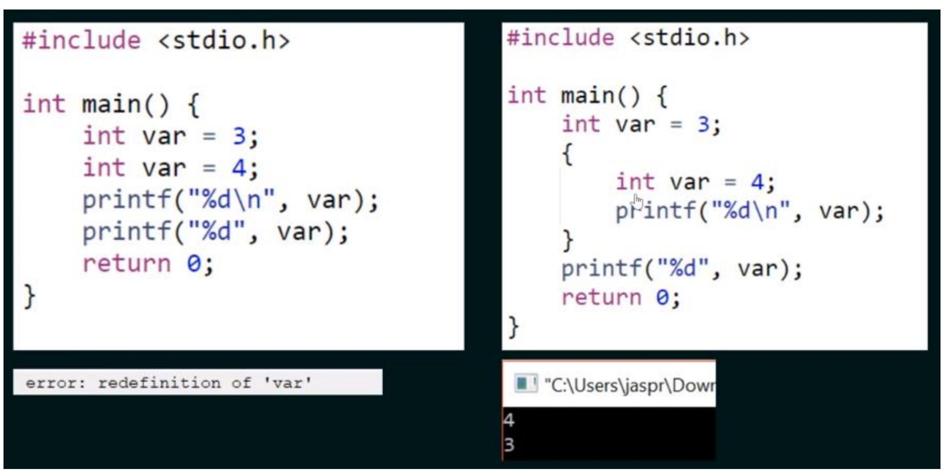
   vi. **Rules for type casting:** Smaller into bigger or same size into same size. Can type cast bigger in smaller. But then loss of data occurs.
2. **Use of typecasting and an example:**

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

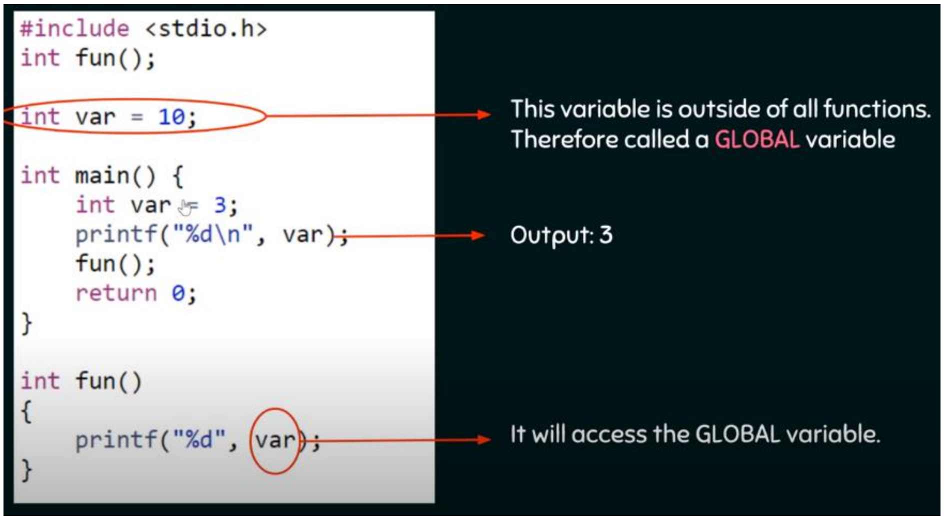
## Scope of Variable:



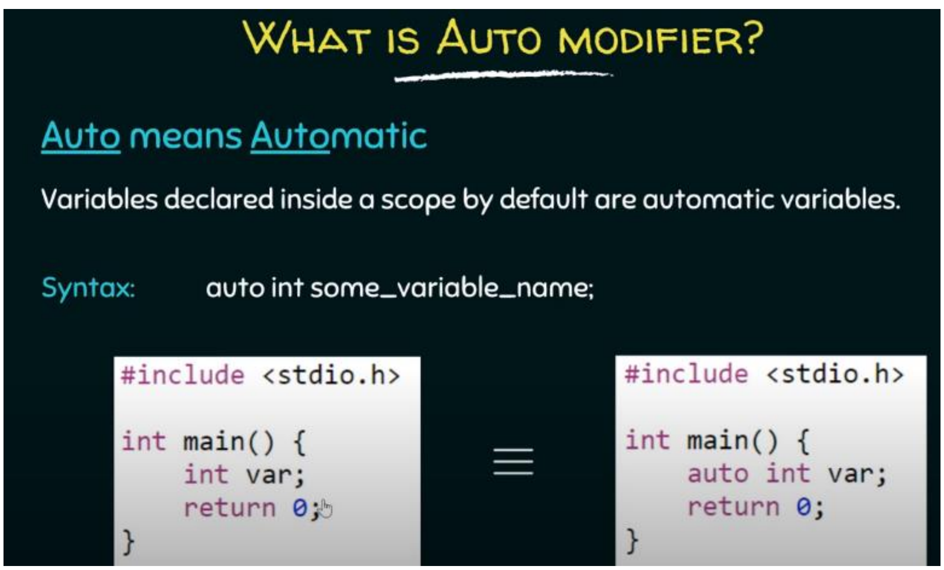

1. Block of code is not just separate functions but also can be created by using **curly brackets**.
2. **Global Variable:** Is also called external variable.



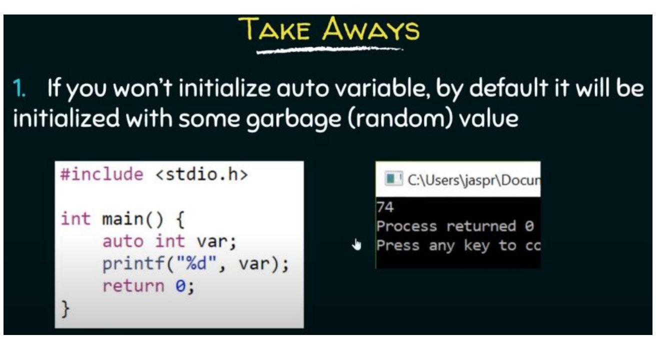

## Modifier:

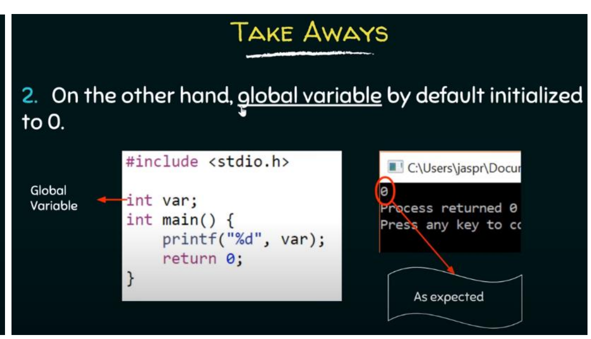
**Auto Modifier:** Automatically **destroyed.**

1. Every variable is a auto variable by default.




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.**



# 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.

```c
extern int a;

int main()

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

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

6. Let us define it out of scope now.

```c
extern int a;

int main()

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

a = 10;
```

```c
int a;

int main()

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

a = 10;
```

Output is 10 in both cases.

7. Let us define a global variable with extern.

```c
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.

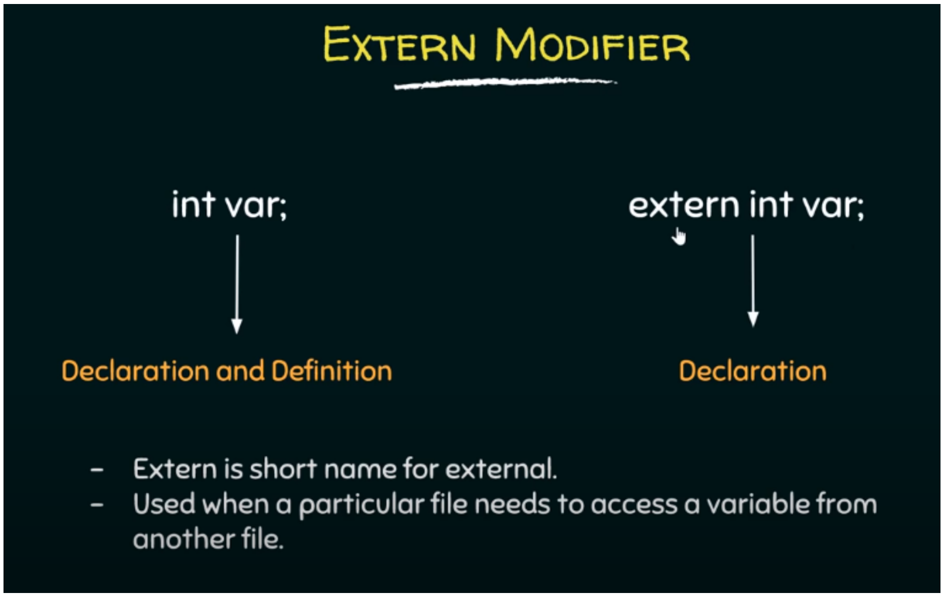
*in file1*:

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

void push(double f) { ... }

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

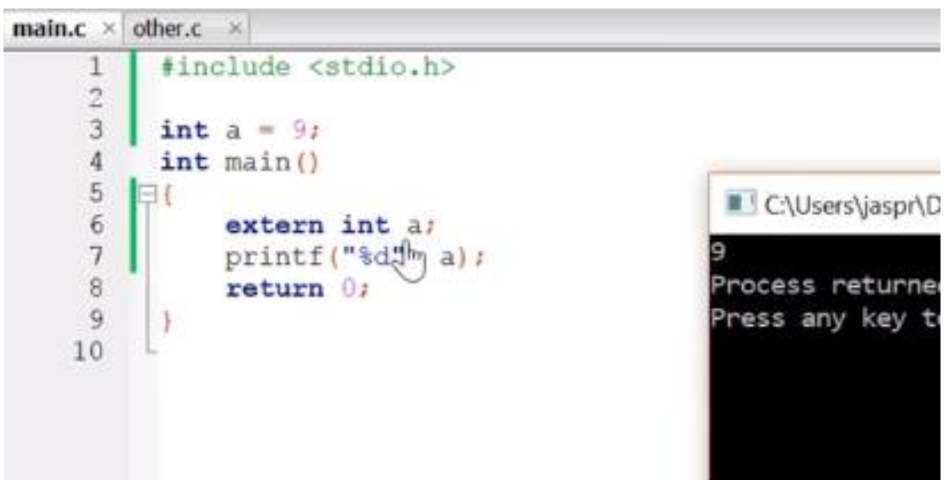
*in file2*:

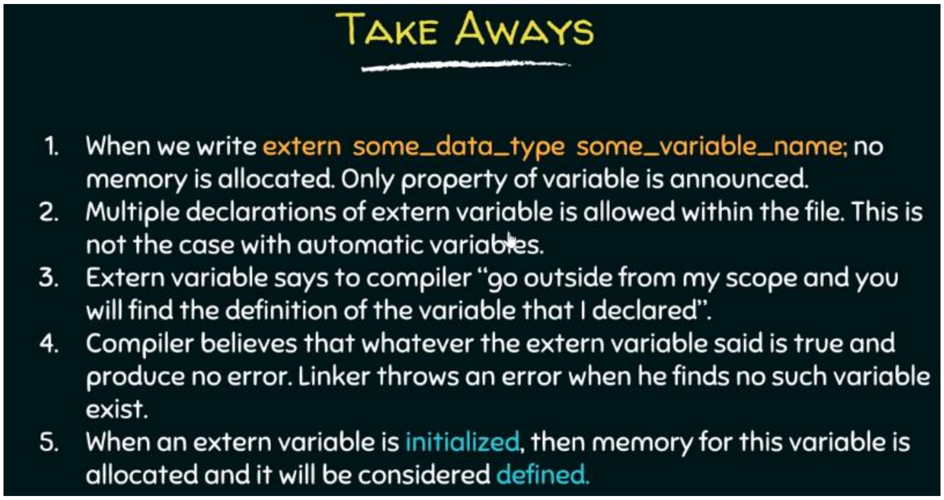
```c
int sp = 0;
double val[MAXVAL];
```

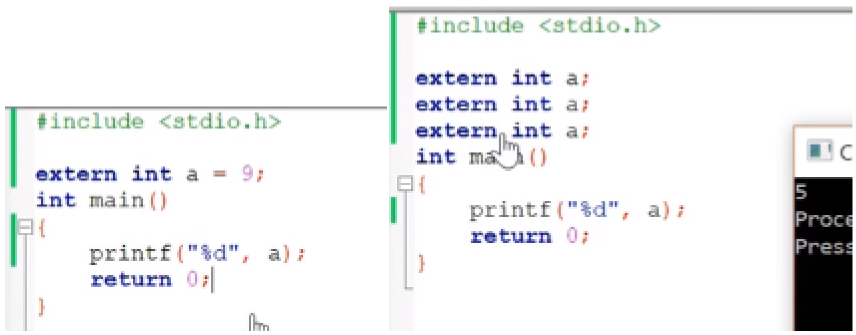
9. **Declaration means variable exists**, and **definition means book a memory space for** it.
10. Extern modifier: The variable is externally declared (out of scope) and defined in some other file. Therefore, don't define it.
11. Who can modify the variable? - **Linker links all the files together**.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



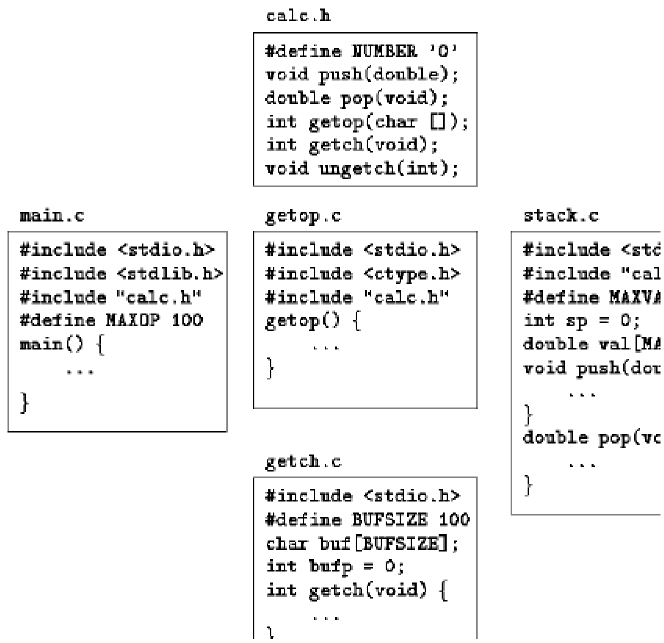



13. Multiple **declarations are allowed** but **multiple definitions are not allowed**.
14. Once the **extern variable is initialized** then memory for it is allocated and will be **considered defined.**

## Header file:

Keep things centralized. Below picture explains it very nicely.

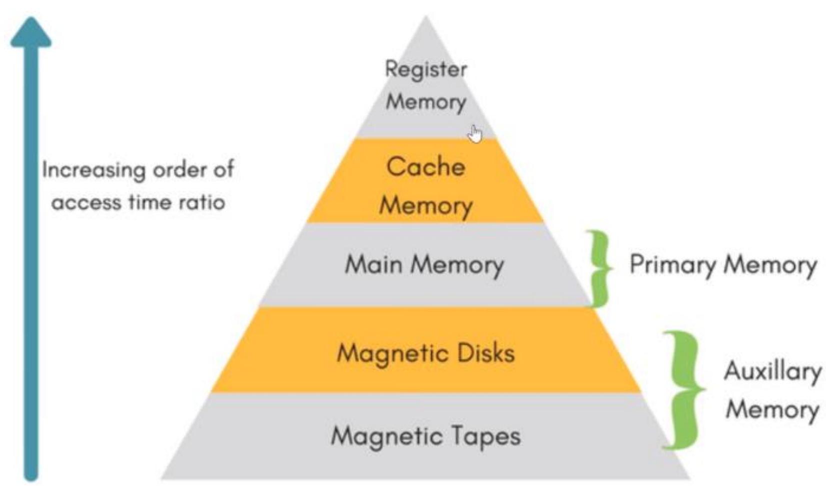


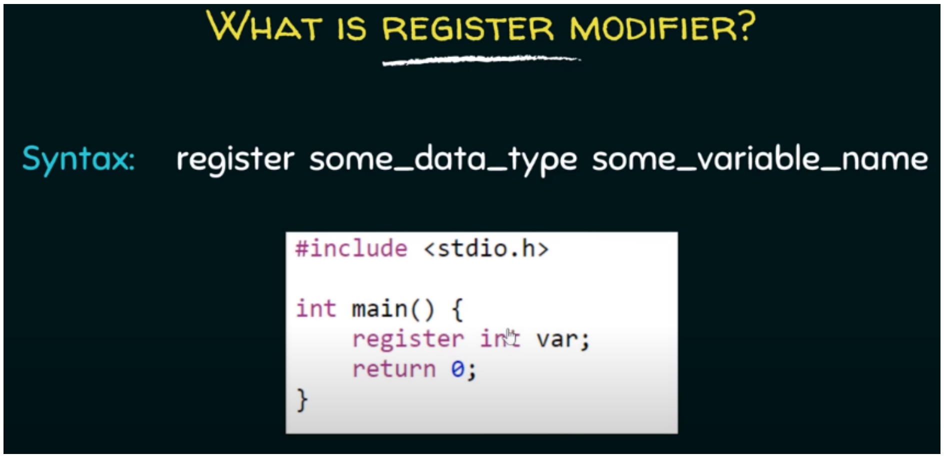

## Register Modifier:

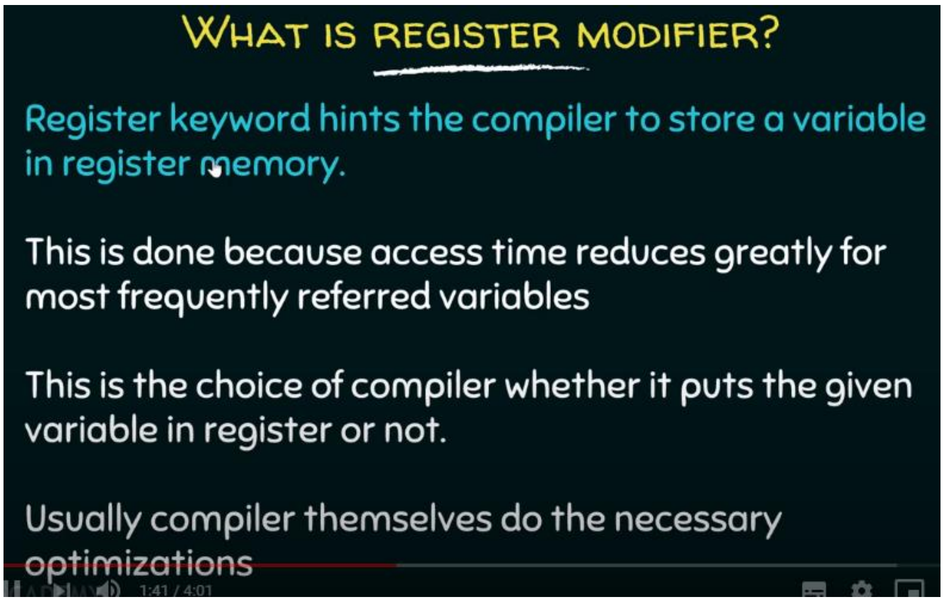
- A `register` declaration advises the compiler that the variable in question will be heavily used. The idea is that `register` variables are to be placed in machine registers, which may

  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;
      ...
  }
  ```

- 



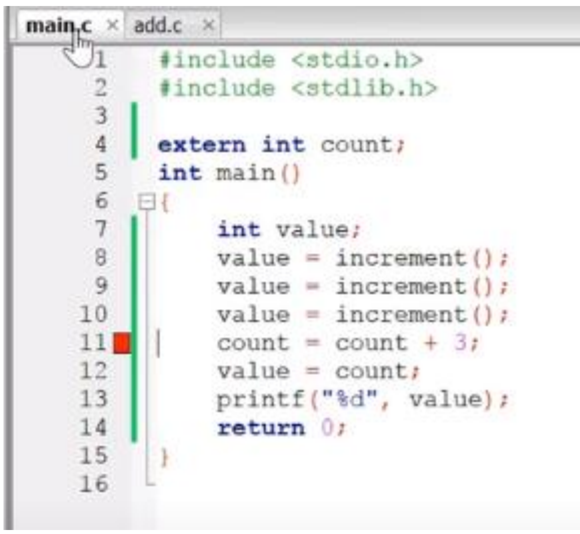

1. Whichever **variable needs to be accessed multiple times,** put that in register file. Since **access time is fastest** in it.

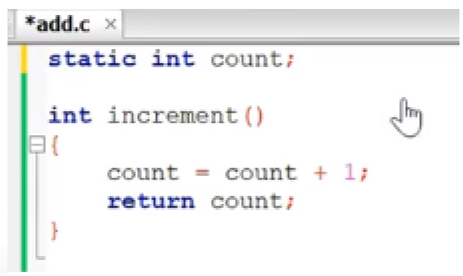
# Static modifier:

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.

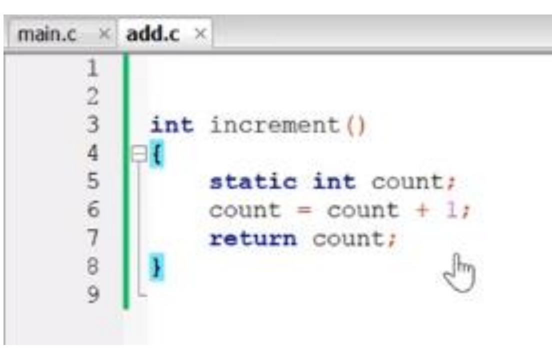
The `static` declaration can also be applied to internal variables. Internal `static` variables are local to a particular function just as automatic variables are, but unlike automatics, they remain in existence rather than coming and going each time the function is activated. This means that internal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

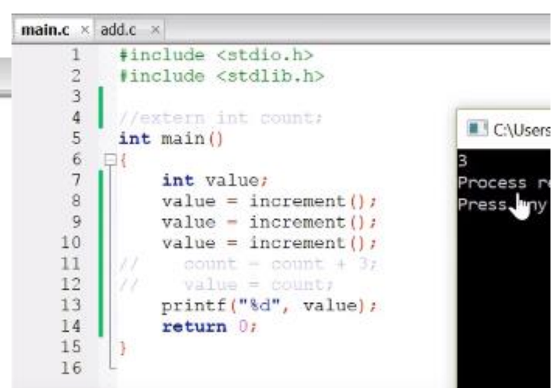
1. How to protect a global variable from other files? – Declare it static




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.



## 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;
}
```

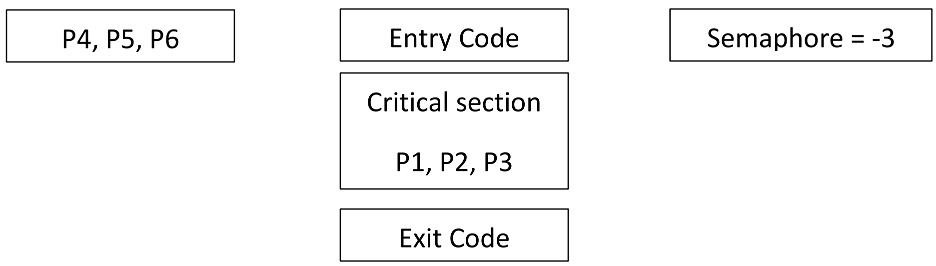
1. **Drawback of volatile keyword:** It can affect the variable if it is in the loop.

## 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.



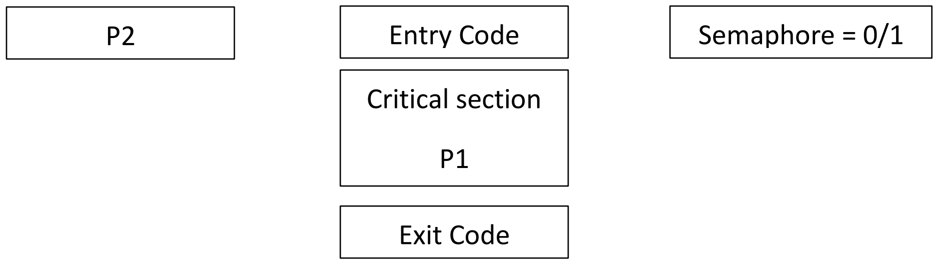

I. **Entry code:** P – wait – down.

```
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.

```
//Intially 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. **Before we use fork():** Let's talk about the header file and attributes. : Int fork()
2. **Header file: #include <unistd.h>**
3. **Attributes:** takes in **void** and returns an **integer. Positive value** is returned to **the parent processes**, **zero is returned** to the **child** and negative number is returned if **child** can't be **created.**
4. **Fork:** Create a new timeline from the **line of fork instantiation**. Two **fork () will create 2 to power 2 processes timeline.**
5. **Fork** returns an **integer ID**. **Baby timeline** has an **id** of **0. Father timeline** has a **non-zero ID**.
6. A child process **uses the same program counter**, **CPU register, same files** that are used by the **parent process.**

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

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

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

```
Harsh Dubey r

Hello World!
Hello World!
Hello World!
Hello World!
```

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

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

```
Harsh Dubey r

Hello Father
Hello baby
```

7. **How to create 3 processes?**

```c
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

Harsh Dubey ran 26 lines

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

8. 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() and exit().**
   i. **Wait():** pid_t wait(int *stat_loc)
   ii. **Exit():** void exit(int status)
   iii. 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. O**therwise it will wait for it to fill the buffer and then print every thing.

```c
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 li

6 7 8 9 10 1 2 3 4 5
```

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

```c
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);
  }

}
```

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

v. Now why wait? Child and parent can have different execution time.

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

  **vi.** Child of child = **0,0**

  **vii.** Main: **1,1**

  **viii.** Whatever **Id the child is born of will be Zero**.

  **ix.** **Rest whatever is left the child will inherit**.

10. **Pipe** the process: Transfer **data between the processes.**

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

//pipoe
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);
}
```

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

```c
#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);
   }

}
```

```c
#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);
}
```

**12. Piping with more than two processes:**

```c
//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.**

```c
#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 2nd thread is create the first thread is already done with its work.

```c
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);

}
```

```
Harsh Dubey ran 43

Mails: 160476

Harsh Dubey ran 43

Mails: 175432

Harsh Dubey ran 43

Mails: 174558

Harsh Dubey ran 43

Mails: 200000
```

| | 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 ques. 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 3rd 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 2nd 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:**

```c
#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.

```c
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.

```c
#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. **We** can print random numbers to a certain range: **(0 to +ive)**

```c
#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: (+ive to +ive)**

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

```c
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: (-ive to +ive)**

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

ix. **(-ive to 0):**

x. **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.

```c
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
```

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

```c
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;
}
```

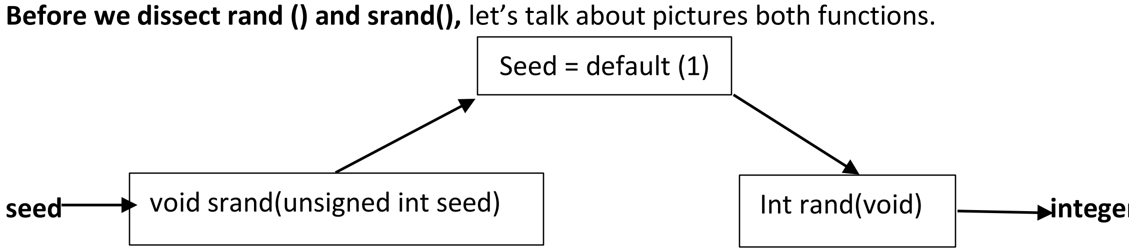
```
Random number: 0.573666
Harsh Dubey ran 28 lines of C
Enter a number: 10
Random number: 2.551377
Harsh Dubey ran 28 lines of C
Enter a number: 20
Random number: 0.930941
Harsh Dubey ran 28 lines of C
Enter a number: 25
Random number: 17.573496
```

**xii. 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.

```c
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:**

```c
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 srandom(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

  srandom(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
```

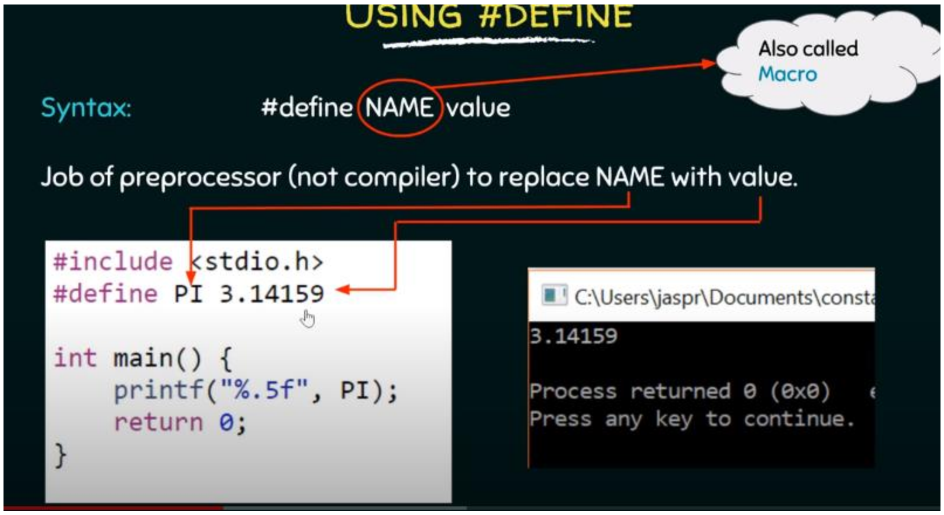
7. **Let's write a simple function to generate random numbers without using rand():**
8. **Let's change the upper limit of rand():** Very important.

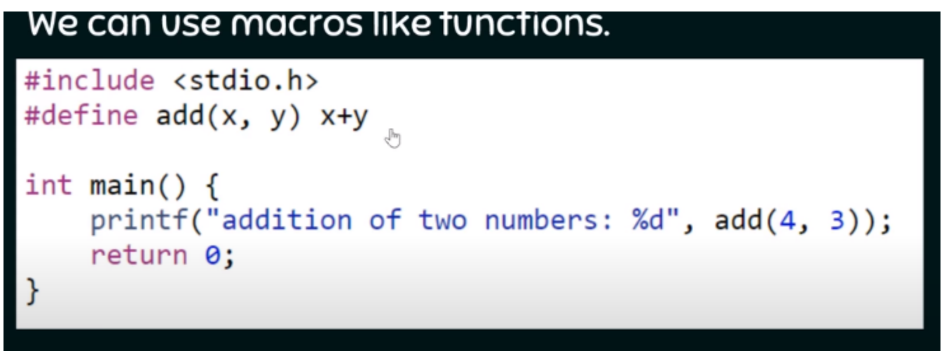
```c
unsigned int seed = 1;

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 srandom(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

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

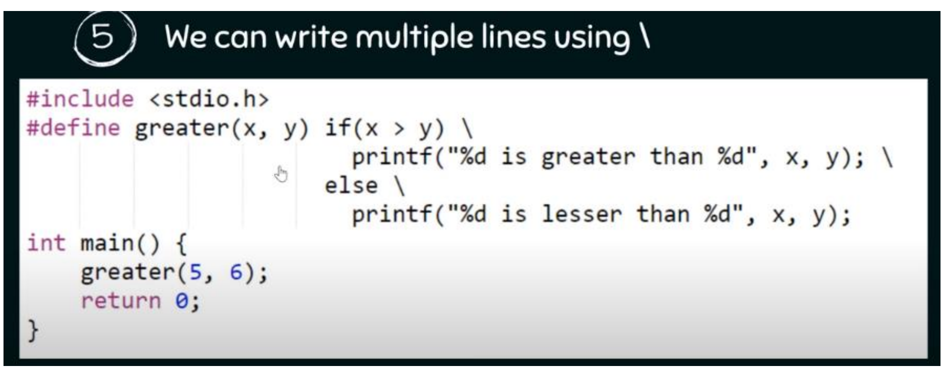
```
2103257819266719230
7801538317933176189
4375123187282855042
```

# Const keyword and Pre-processors (#define)

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




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

3.

```c
#define MulDiv(x,y)  if(x>y)\
                          ( x*y);\
                     else \
                           (x/y);\

int main()
{
    printf("%f",MulDiv(2.9,2.0));

    return 0;
}
```

4. Below is allowed.

```c
#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;
}
```

## 6 First expansion then evaluation.

```c
#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

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

```c
#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:**

```c
#include <stdio.h>

int main()
{
    const int var = 97;
    var = 57;
    printf("%d", var);
}
```

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

```c
const int baby = 3;

int main()
{

    baby = 1;
    printf("%d\n\n", baby);
```

**9.** But can be reinitialized.

```c
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.

```c
const int baby = 3;

int main()
{

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

    const int baby = 177;
    printf("%d\n\n", baby);

```

**Questions:**

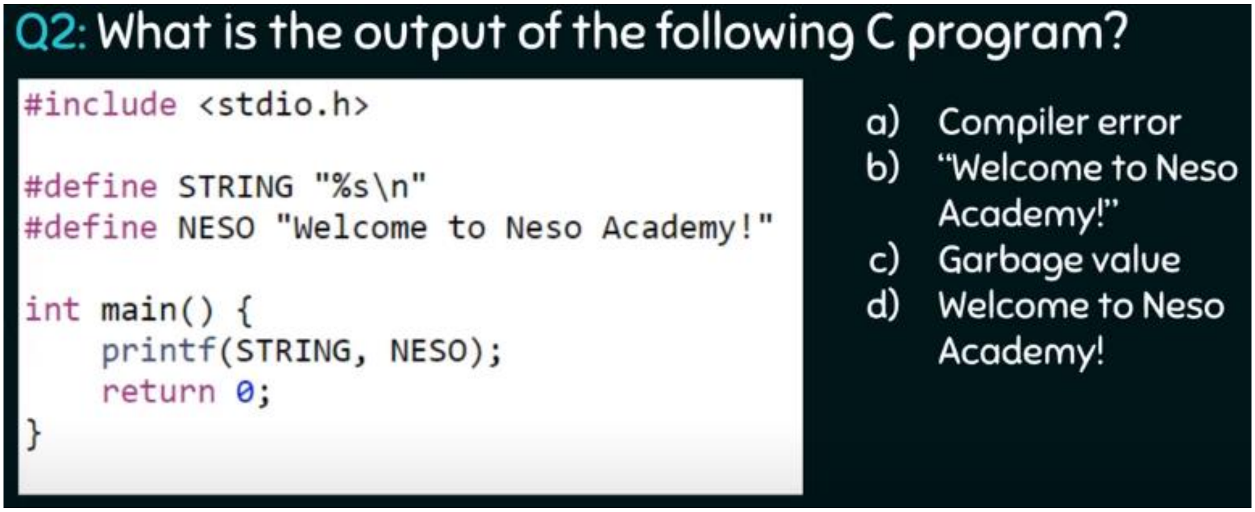
**Q1: What is the output of the following C program?**

```c
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.

$$\begin{array}{cc} 8^1 & 8^0 \\ 5 & 2 \end{array} \qquad 5 * 8 + 2 * 1 = 40 + 2 = 42$$




Answer: D, no double quotes

# Scanf and Printf and File terminal I/O

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.

```c
#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 o

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.

```c
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 "Enter key press" from the display buffer is not taken out. The [^\n] reads the \n or enter key press from last string and the terminates.

```c
#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.

```c
#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.

```c
#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:
aaaaaaaaaaaaaaaaaaaaaaa

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.

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

int main()
{
  char c[10];
  printf("Enter string: \n");
  fgets(c,10,stdin);
  printf("\nEntered 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.

```c
int main()
{
  char c[10];
  printf("Enter string: \n");
  fgets(c,10,stdin);
  c[strlen(c)-1] = 0;
  printf("\nEntered 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.**

```c
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.**

```c
//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: 0xhex

```c
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:**

```c
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:**

```c
#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:**

```c
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 form the screen and returns it.
   **iii. Getchar():** is basically getc(stdin)
7. **Putc, putch, putchar():**
   **i. Putchar():** Takes and integer and then converts that to a character and prints it. Also, takes in character and prints it.

```c
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.

```c
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:**

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

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

```
hh4
hh4
40
```

**10. Printf: 32 different functions**

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

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

```
Harsh Dubey

Hello 1
```

ii. **N- buffer size:**

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

```
solution.c:8
iting 15 byt
   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.

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

```
Hello123456789 1
```

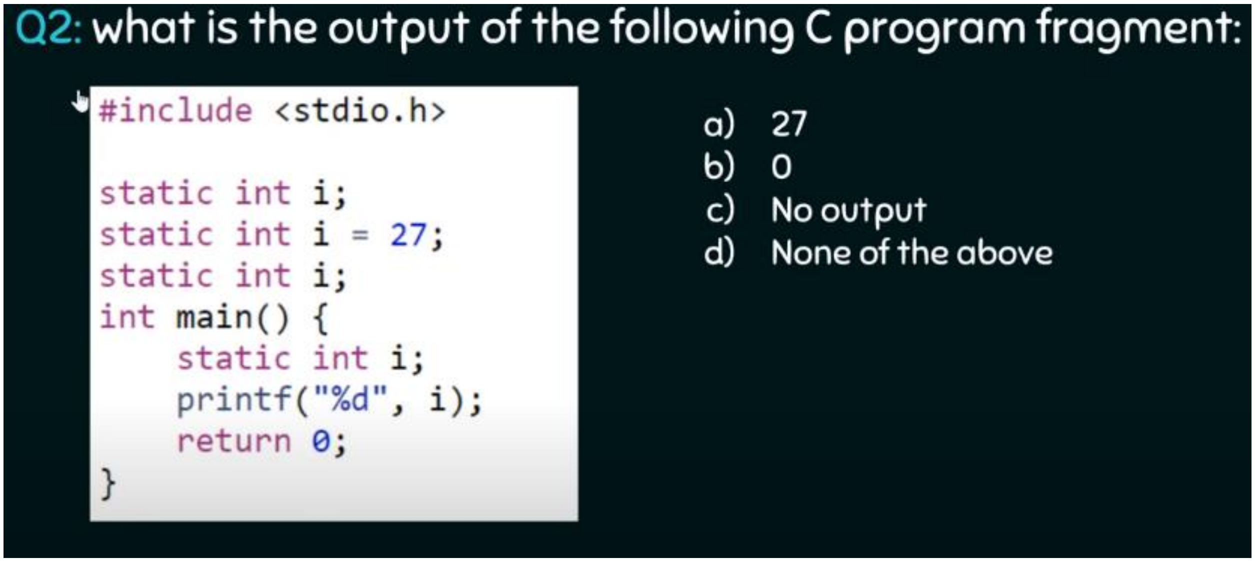
iv. **_s : safer version of fprintf:** Used for safer file operations.

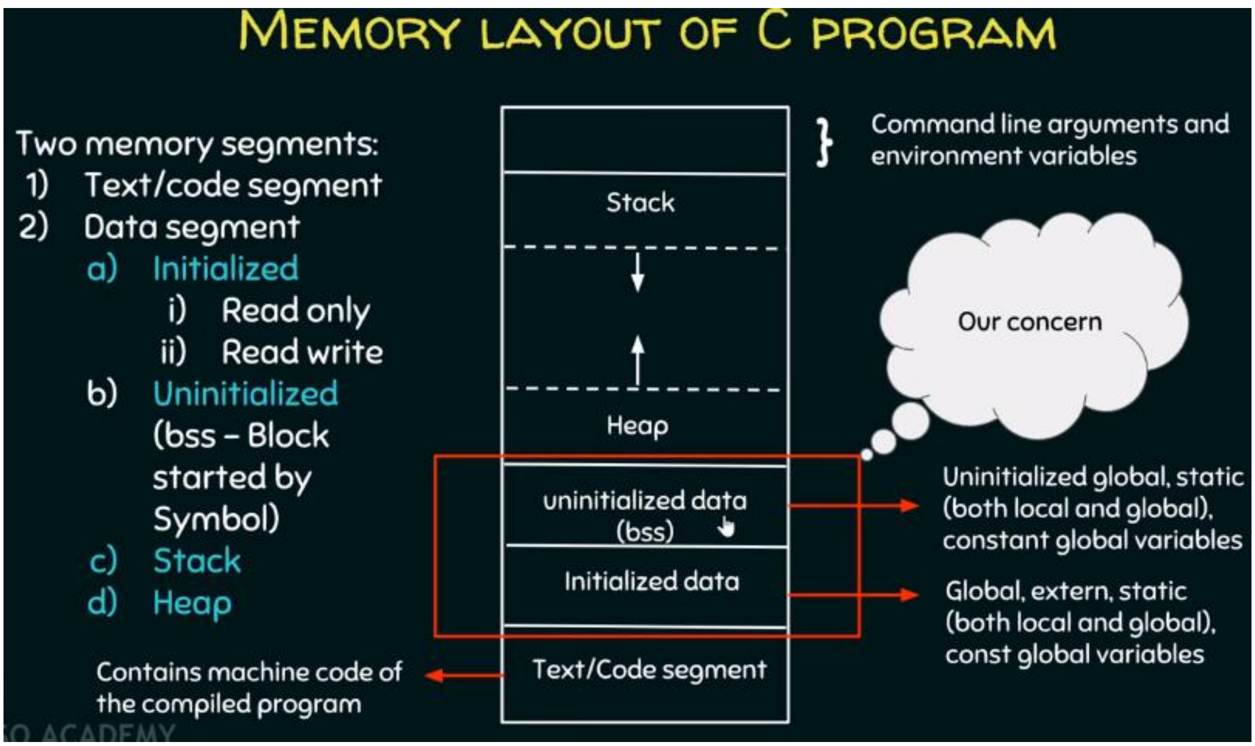
## Questions

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

```c
#include <stdio.h>

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

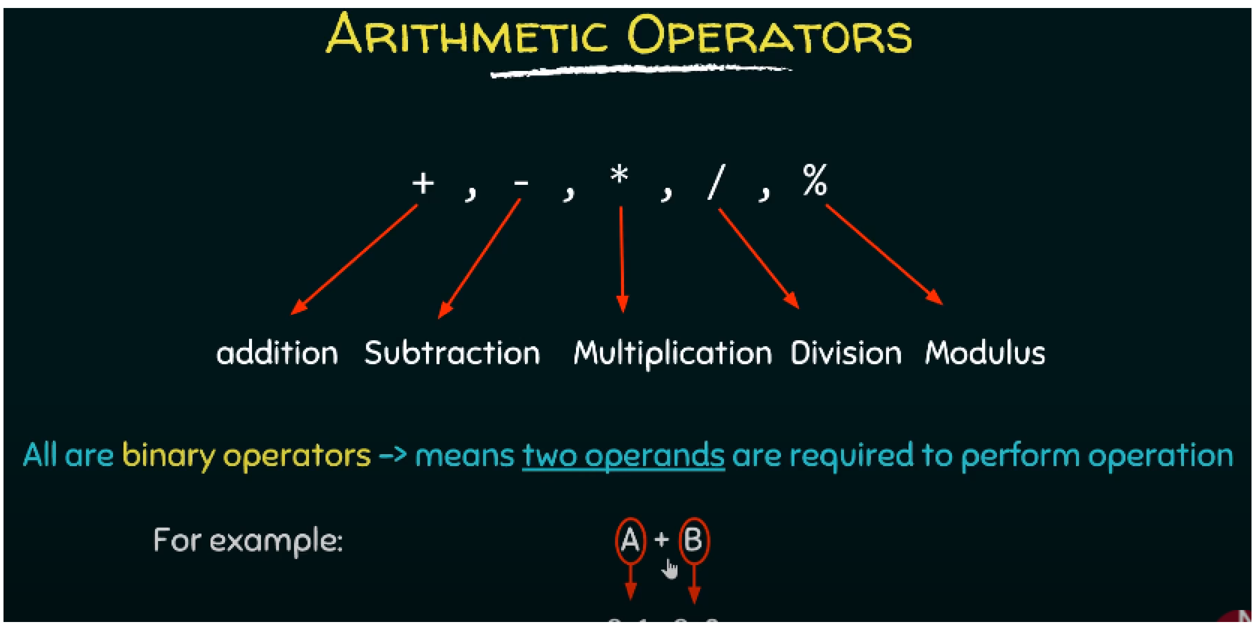
Answer: %x is used to print out hex



# Operator in C:

**Arithmetic Operators:**




1. Modulus = remainder

## OPERATOR PRECEDENCE AND ASSOCIATIVITY

| 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

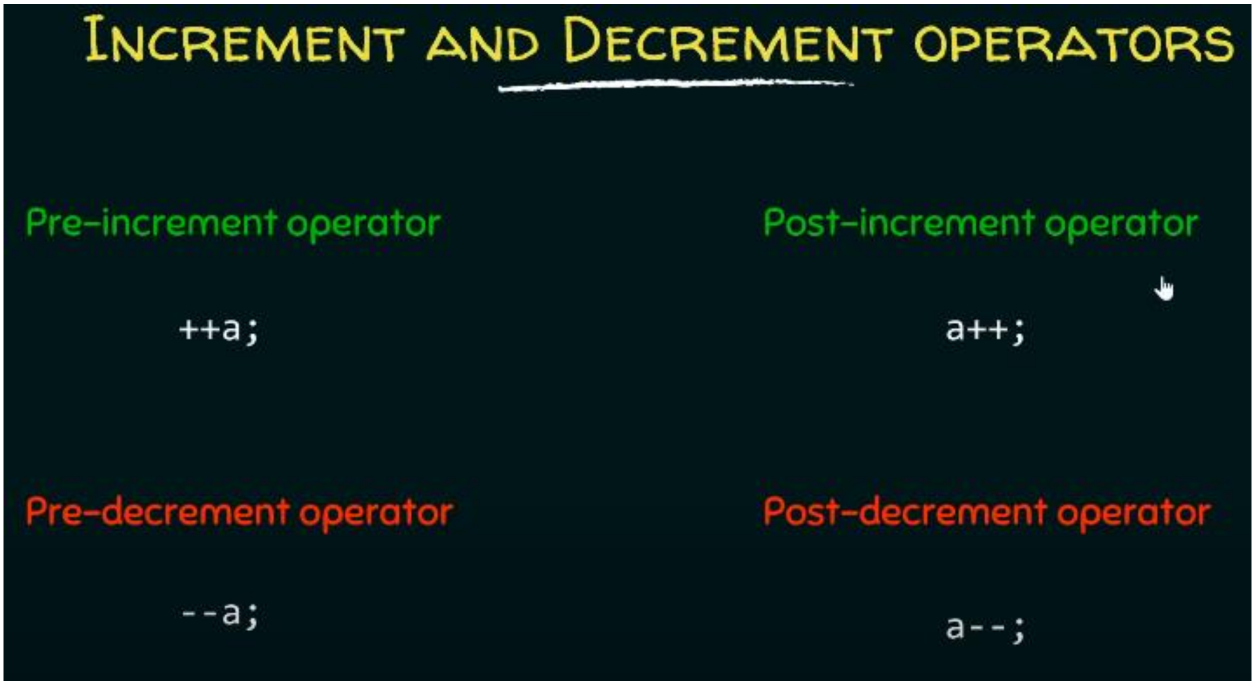
```c
#include <stdio.h>

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

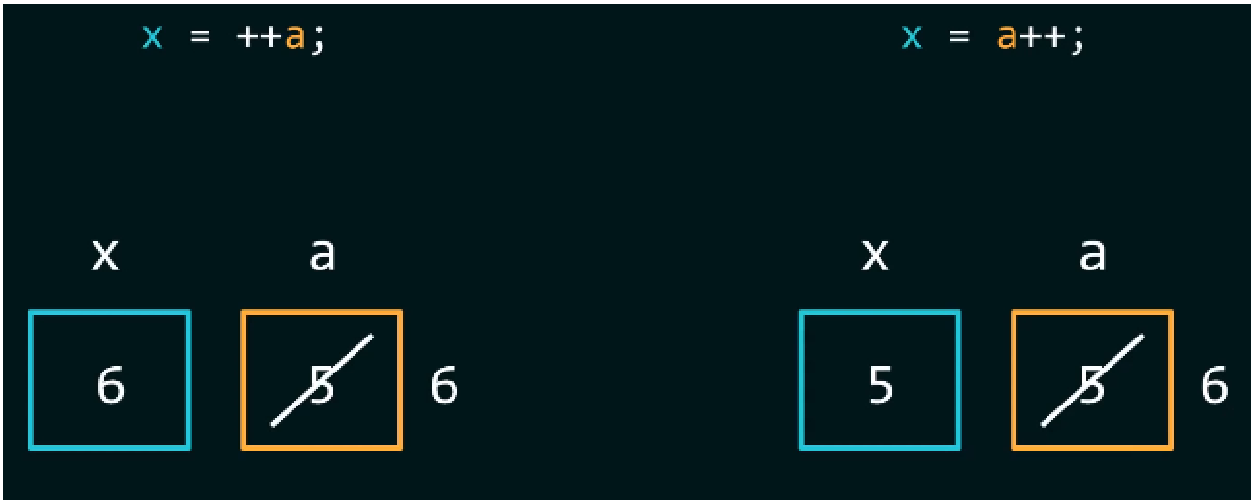
```
a * b / c = 1
a + b - c = 1
a + b * d - c % a = 17
```

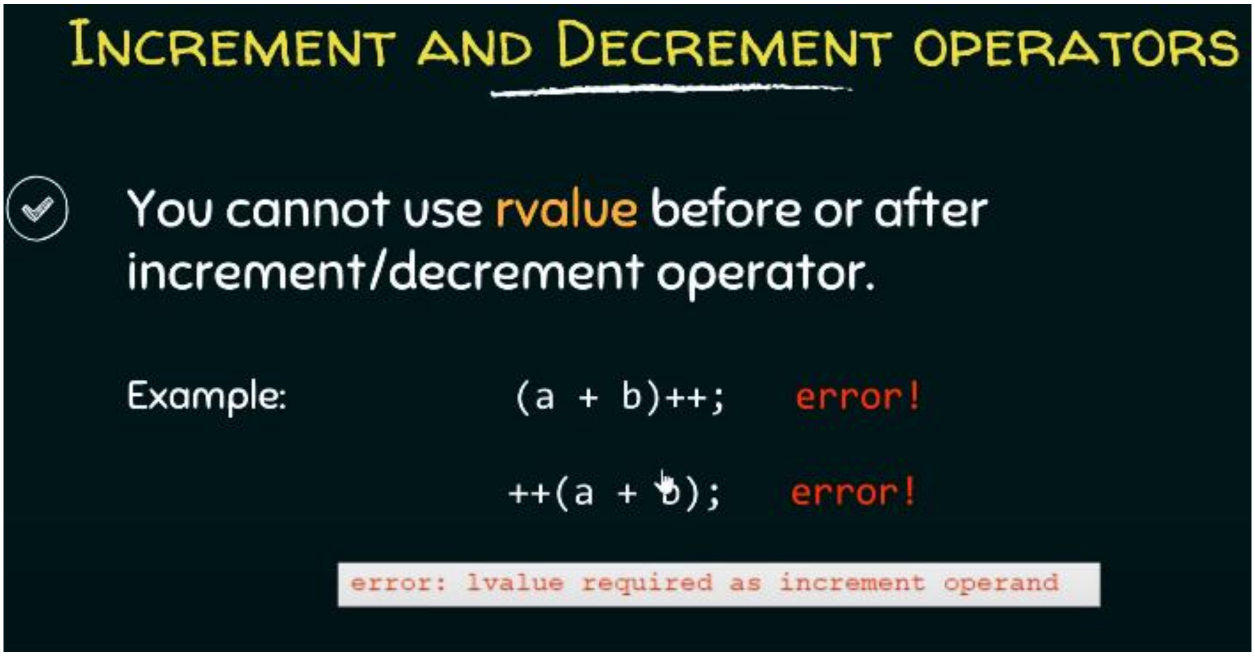
1. Look at the 3rd one and tell why the

**Increment and decrement Operators:**

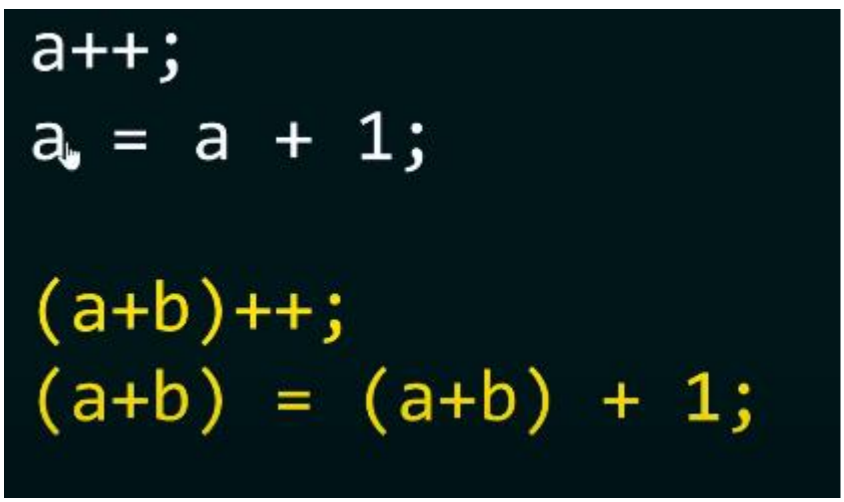



2. **Pre-increment (++i) −** Before assigning the value to the variable, the value is incremented by one.
3. **Post-increment (i++) −** After assigning the value to the variable, the value is incremented.

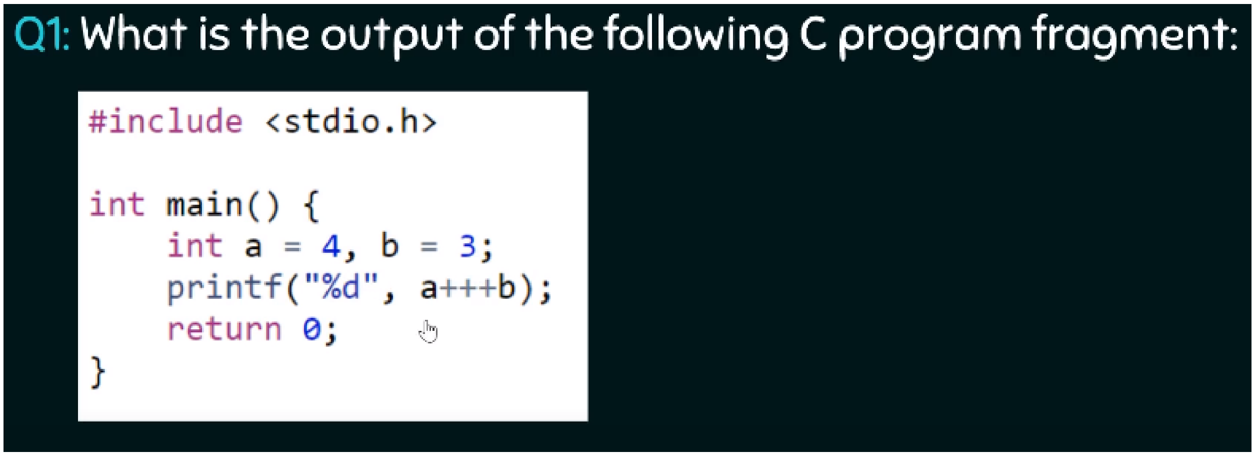




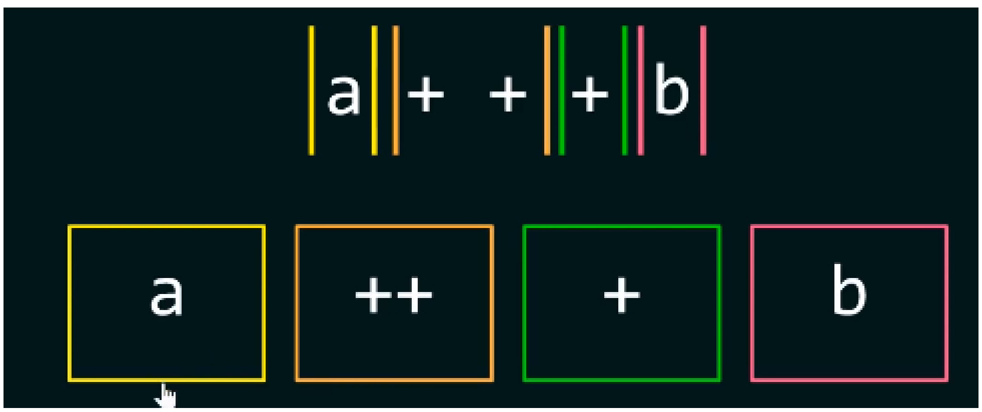

1. Lvalue: should be a variable that has an identifiable location in the memory.
2. Can't be anything
3. Rvalue or right values are usually a expression or function that can return some value.




**Questions:**




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

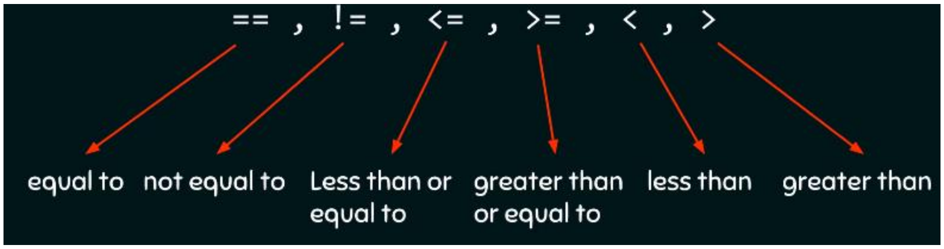
**Q3:** What is the output of the following C program fragment:

```
#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.

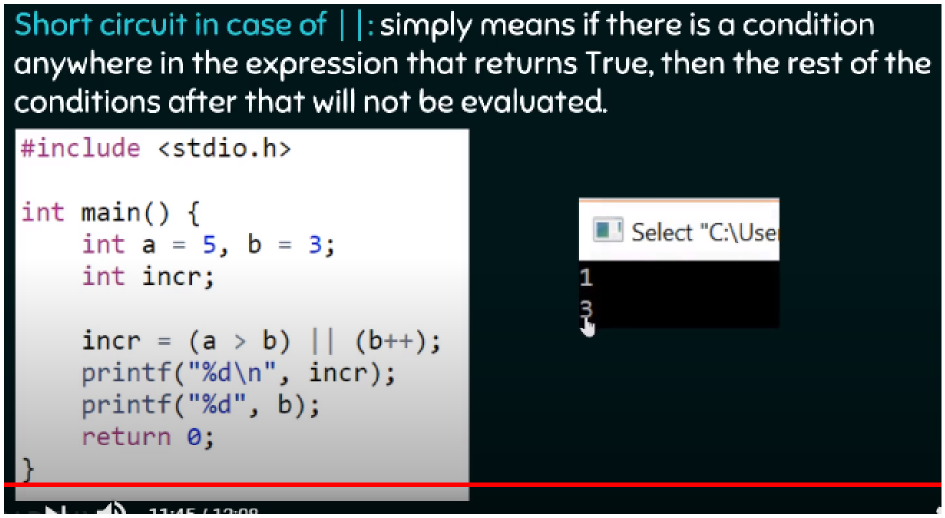
1. **And &&:** You are looking for conditions to be true by checking if **both operands** are **existing** or not. Any number that exists means it's true.

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.

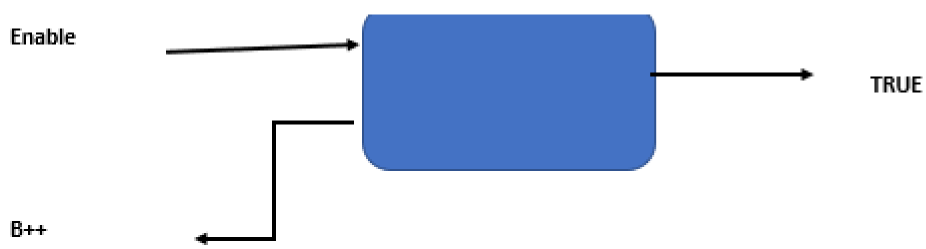
```c
#include <stdio.h>

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

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

```c
#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;
}
```

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.



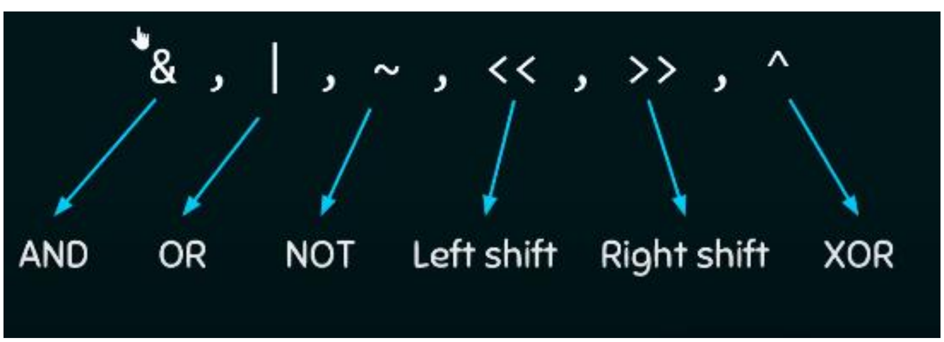

3. Look at the diagram below to understand short circuit for both && and ||



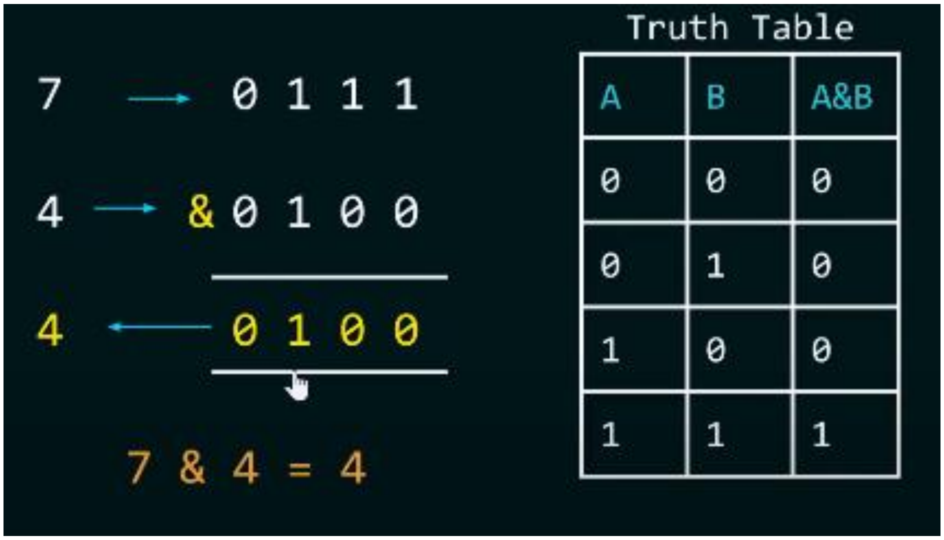

4. **In case of &&** if **Enable** is **high then B++ will BE looked** at to produce true. If enable is low, then FLASE is the output.
5. **In case of ||** if **Enable** is **high then B++ will NOT be evaluated** since, no need in case of OR. One high will produce TRUE.

```
//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);
  }
}
```

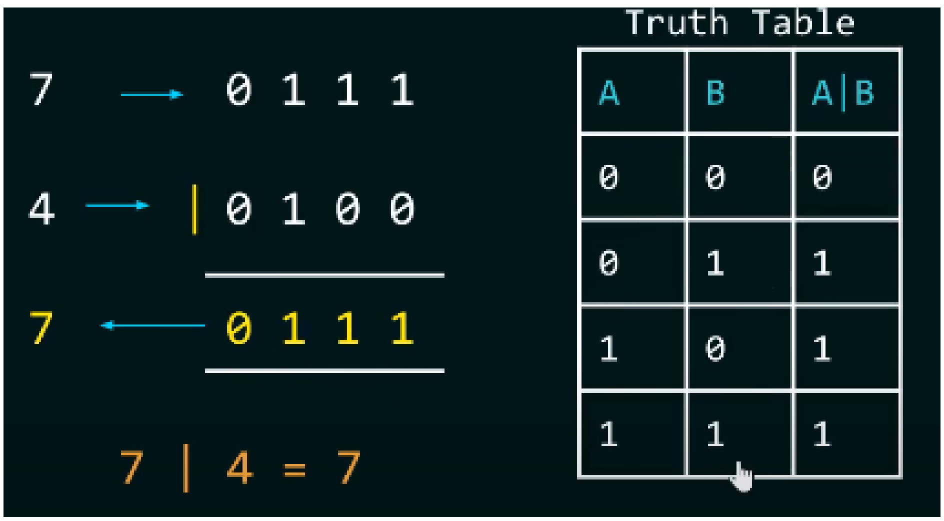
## Bitwise Operators:



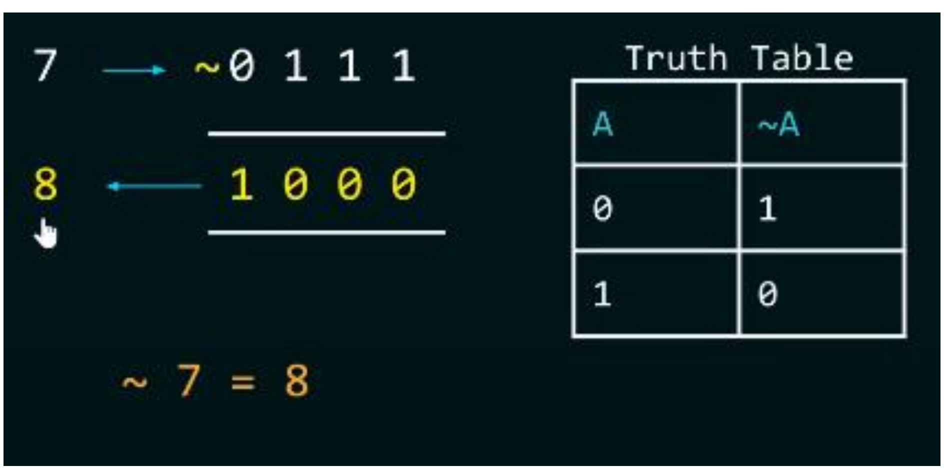

**Bitwise AND:**



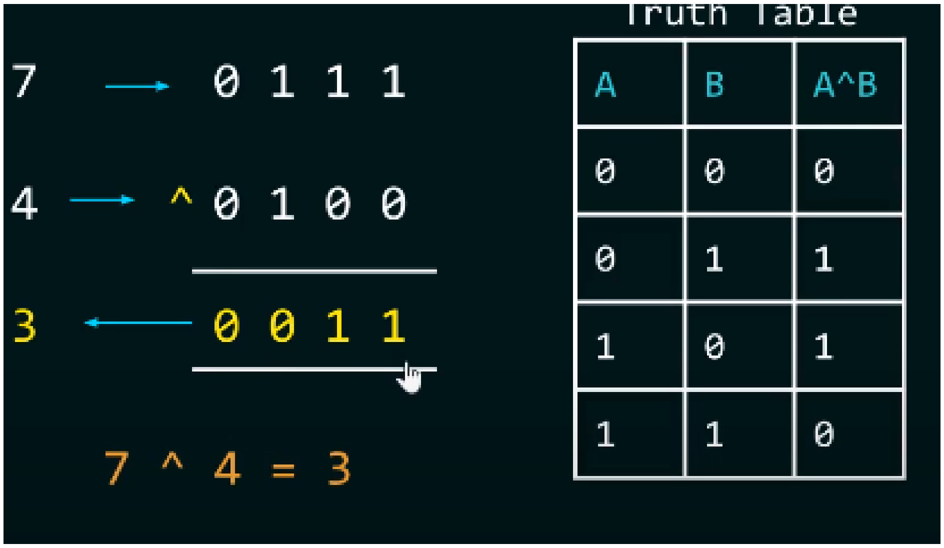

**Bitwise OR:**



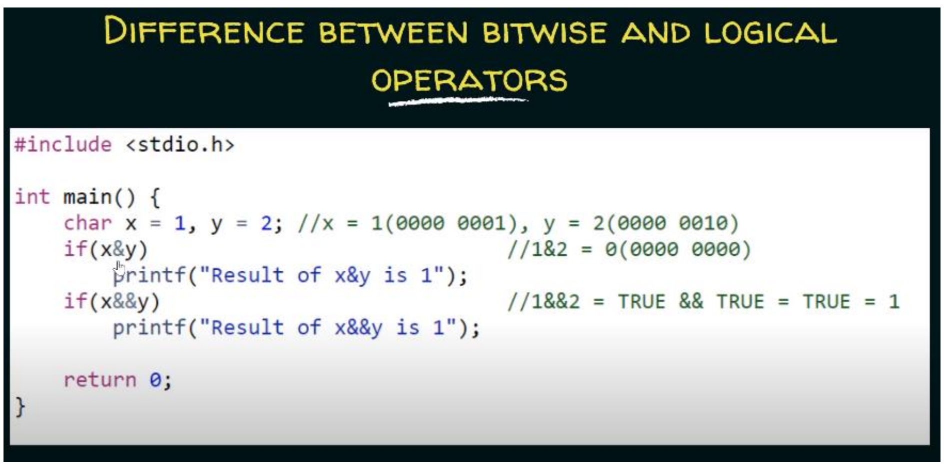

**Bitwise NOT:**




**Bitwise XOR:**

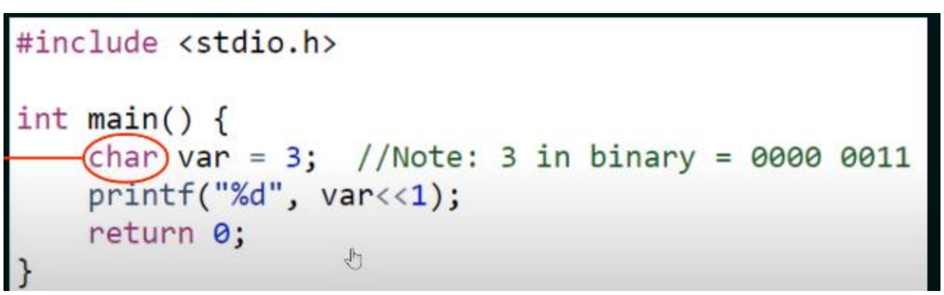



**Difference between Bitwise and Logical:**

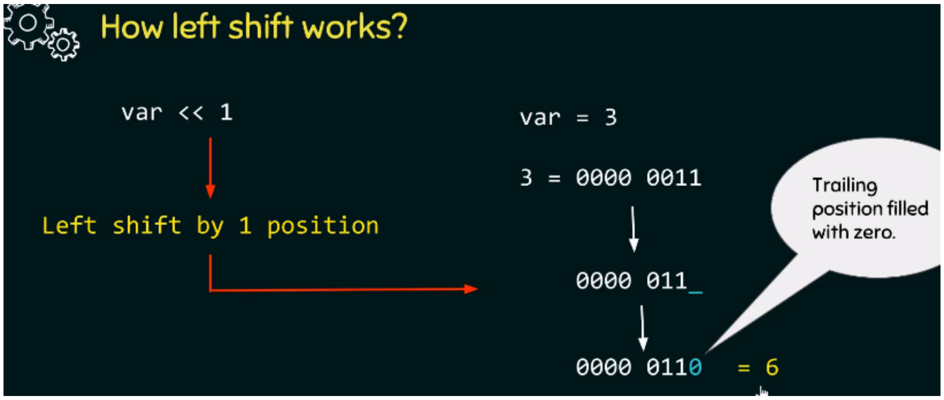


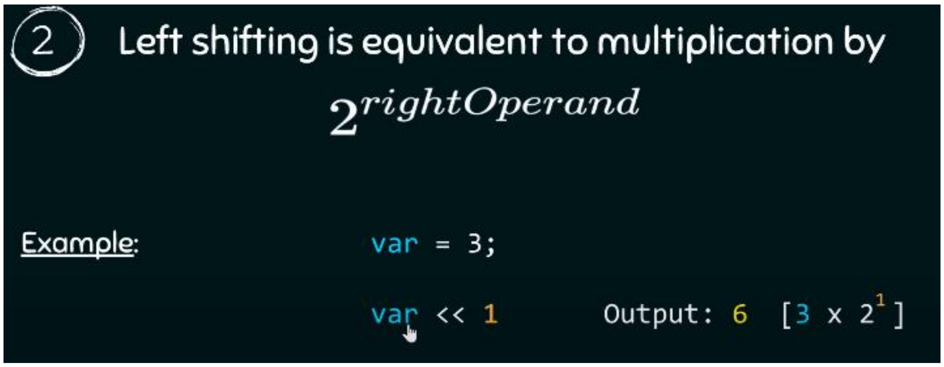

```
#include <stdio.h>

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 = TRUE = 1
        printf("Result of x&&y is 1");

    return 0;
}
```

## Left shift operator:

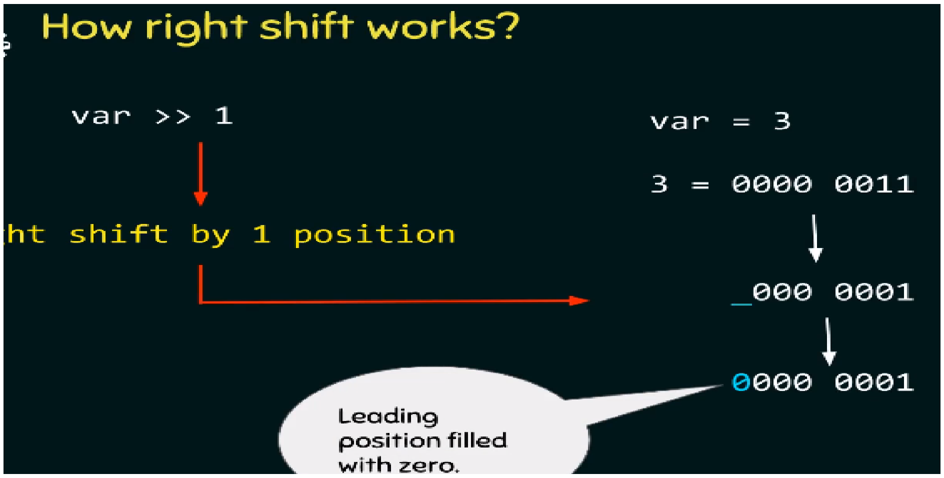


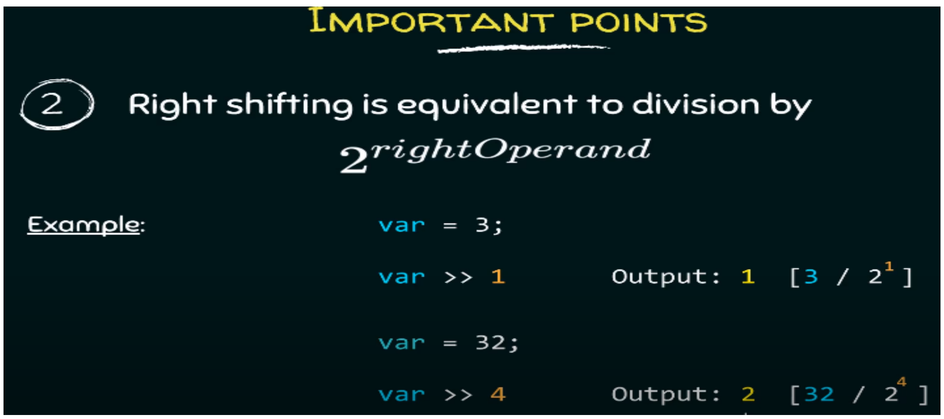
```
#include <stdio.h>

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

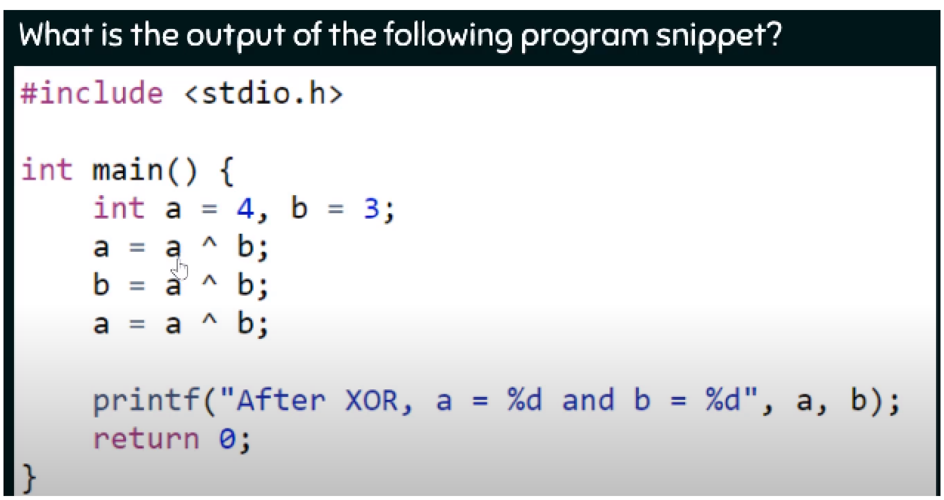



**Right shift operator:**




## Questions on Bit-wise operators:

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

```c
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?

```c
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.

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

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

- ? :

```c
char result;
int marks;

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

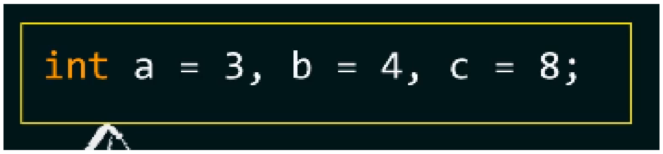
```c
char result;
int marks;

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

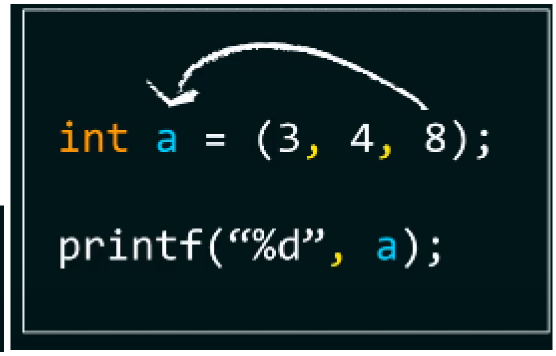
What is the output of the following is C program fragment?

```c
#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;
}
```

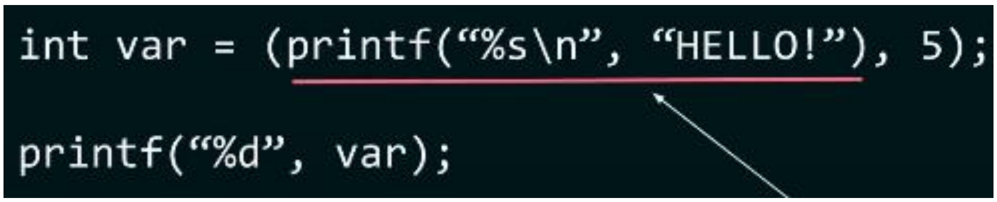
Answer: 65

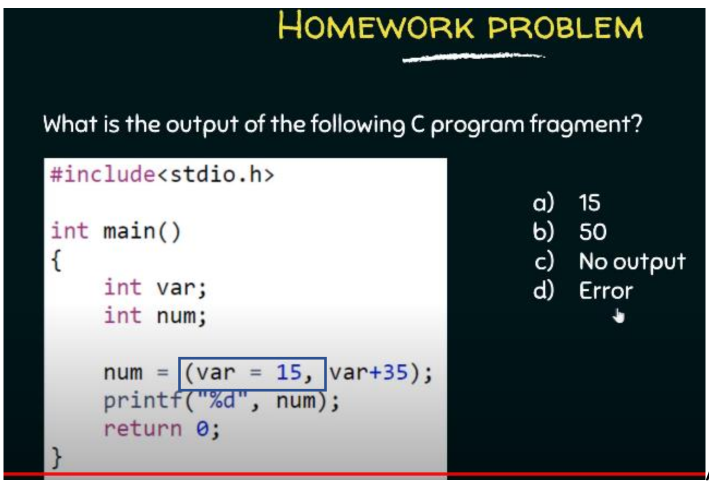
**Comma Operators:** used to assign value to a variable.




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.




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?

```c
#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.

```c
#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?

```c
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

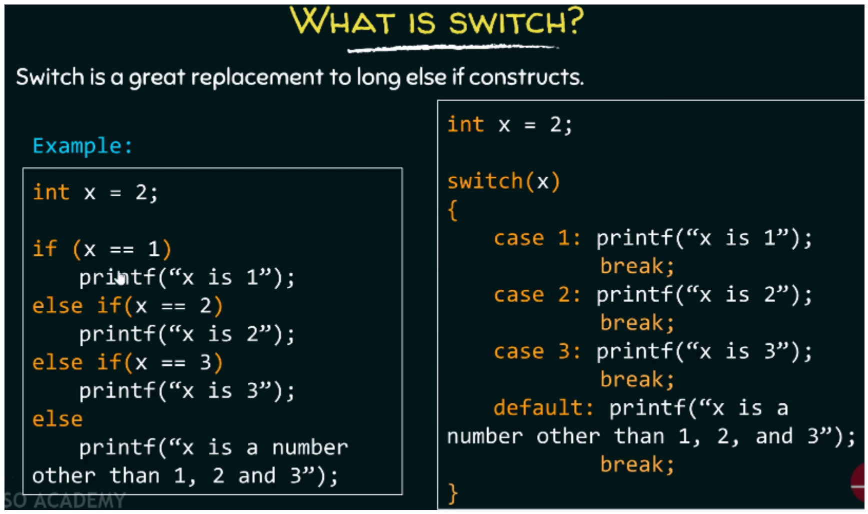
**Rapid Fire:**

Go back to Video 37 and do the quiz.

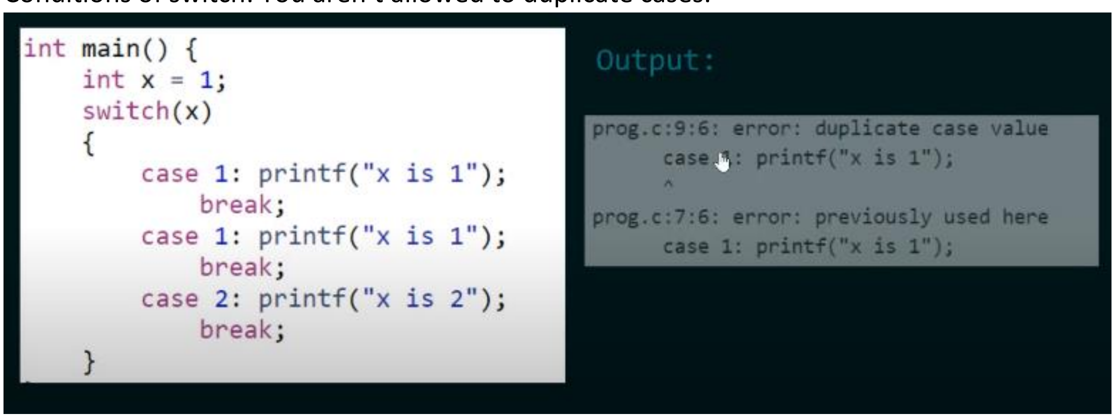
# Conditionals:

**If else:**

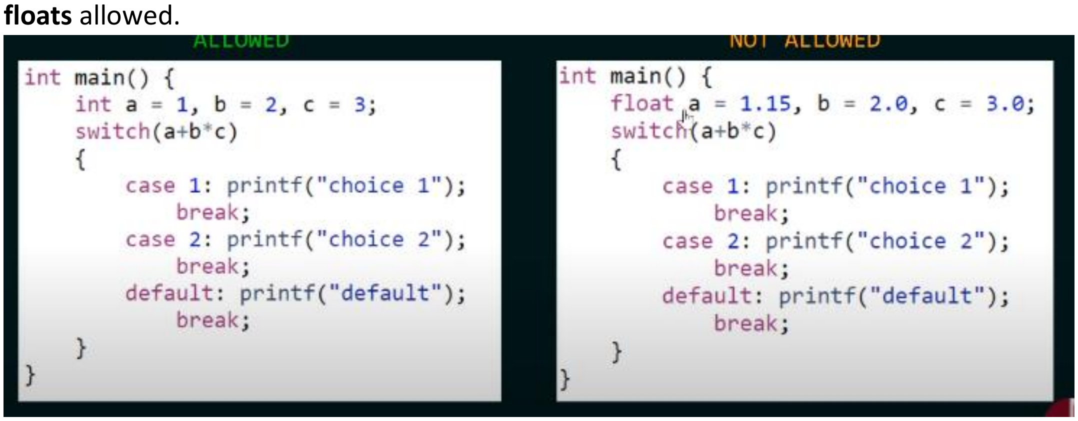
**Switch:** Good replacement to long if else constructs



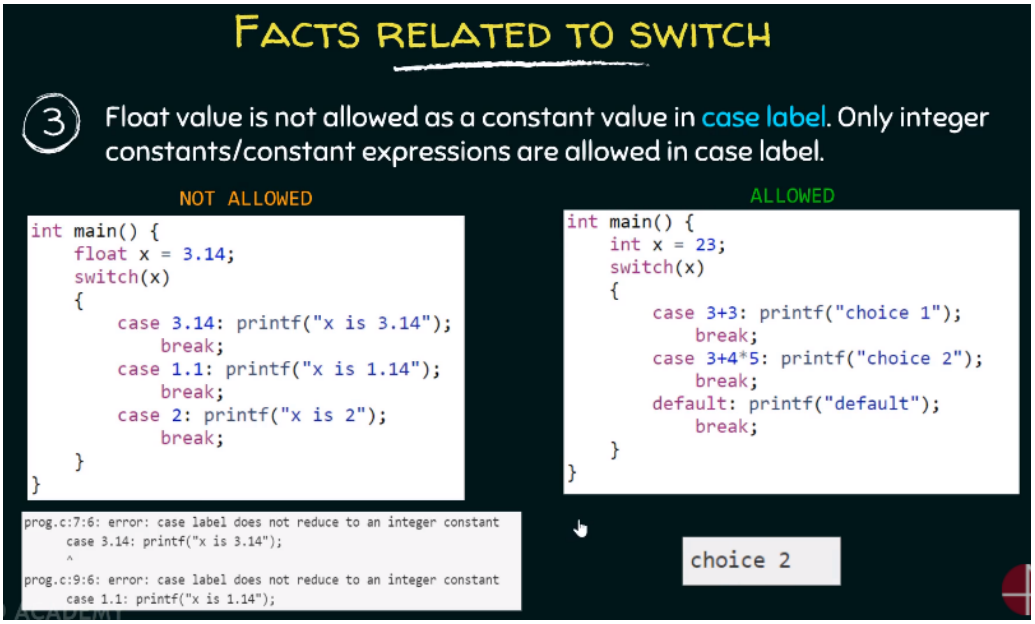

2. Why put break? If you don’t put break, then substituent functions will be evaluated **once the condition is satisfied** until we reach the **break**.
3. Conditions of switch: You aren’t allowed to duplicate cases.



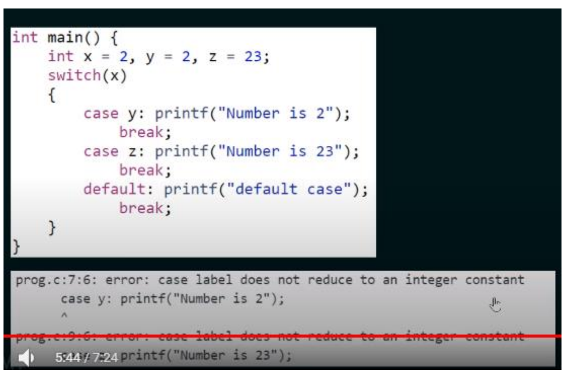

4. Only those expressions are allowed those are integral **constant values** in switch statement. **No floats** allowed.

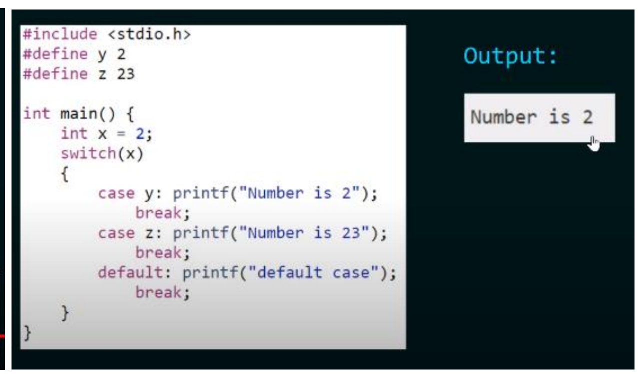


5. **Floats are not allowed in the CASE statement either**. However, **expressions** are allowed that will lead to integer value.




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

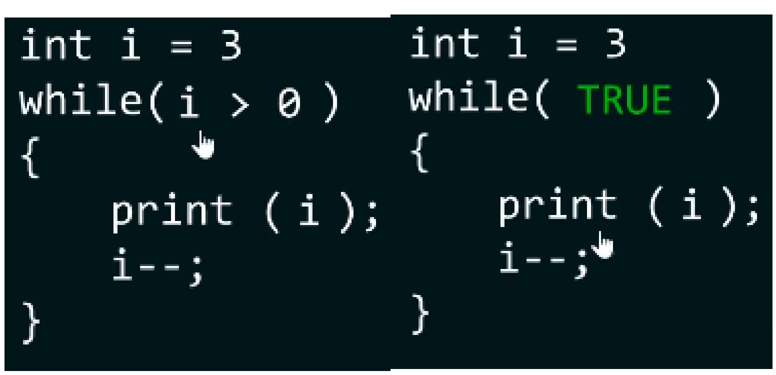



7. **Default** can be put anywhere. And will only be evaluated once all the cases are evaluated.

**For while loop:**

**While loop:**



**For loop:**

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

```
int i;
i = 3;
while(i > 0)
{
    print (i);
    i--;
}
```

≡

```
for(i = 3; i > 0; i--)
{
    print (i);
}
```

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

While

```
int i = 0;
while(i > 0)
{
    printf("%d", i);
    i--;
}
```

do-While

```
int i = 0;
do
{
    printf("%d", i);
    i--;
} while(i > 0);
```

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.

```c
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 21:27:50 on ttys
AntonieSmallMBP:~ Antonie$ /Users/Anton

Break Output:
0
1
2
3


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

**Questions:**

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

```c
#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);
    }
}
```



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!
```

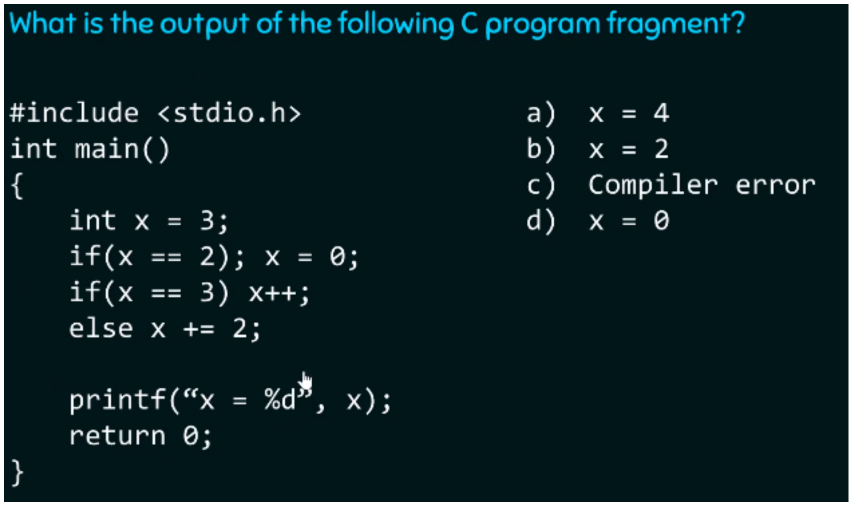
Also, look. ONE is only printed once.

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.




Answer: Do it again

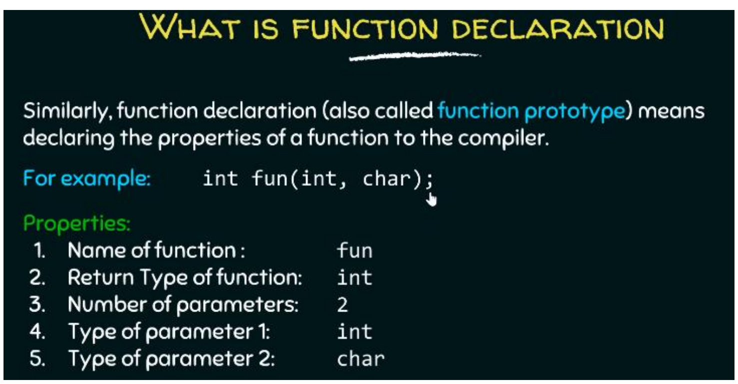
**Pyramid of Stars   Palindrome   Armstrong**

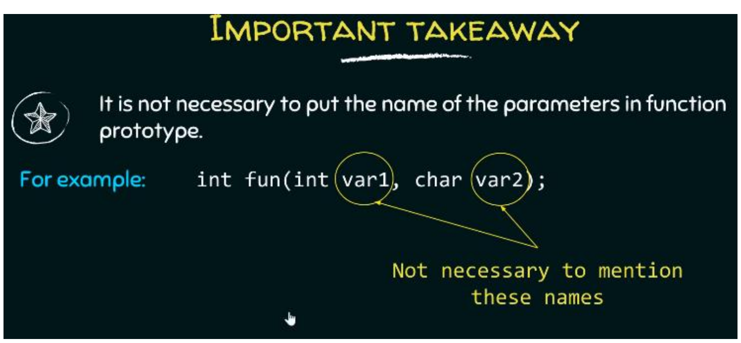
## Function:

Why functions?

- Reusability and abstraction

Function declaration as a prototype:




- 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

```c
#include <stdio.h>
int main()
{
    char c = fun();
    printf("character is: %c", c);
}

char fun()
{
    return 'a';
}
```

```
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, complier 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.

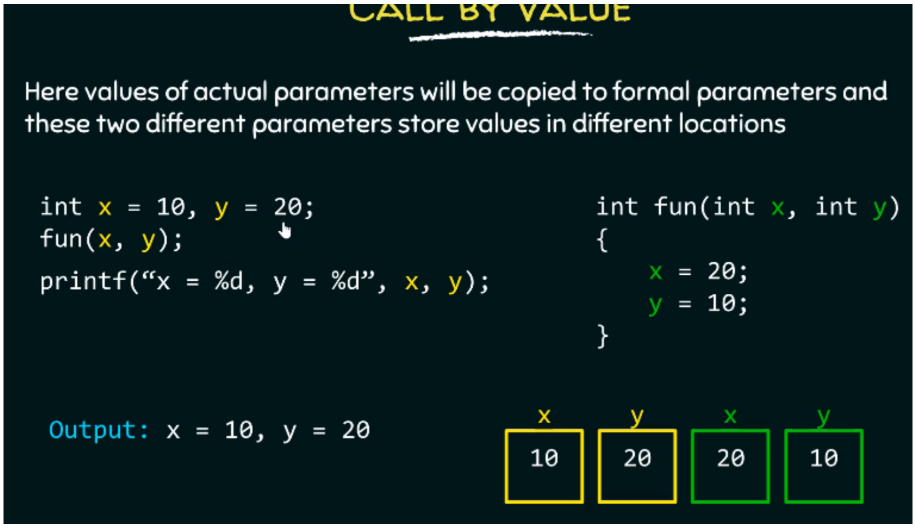
Argument: is the actual value of the parameter that gets passed to the function.

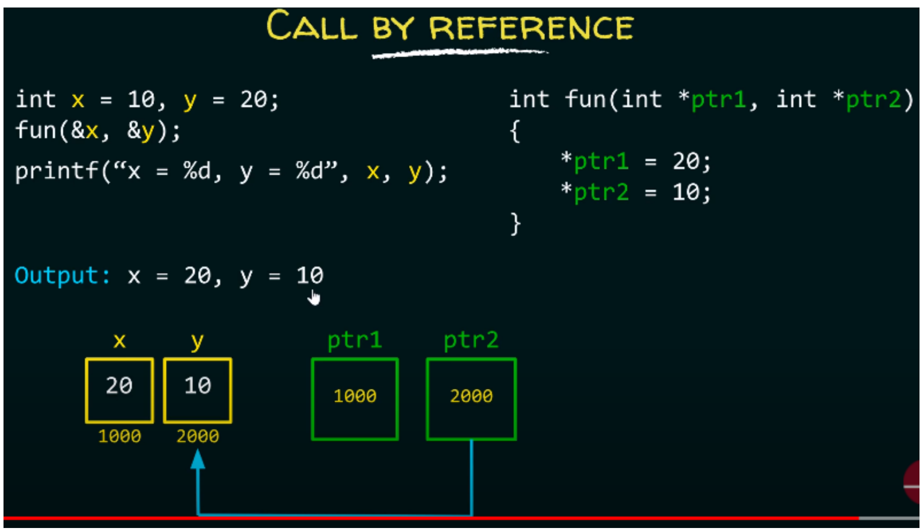
```c
int add(int, int);

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);
}
```

Arguments or Actual Parameters

Parameters or Formal Parameters

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



## Questions:

The output of the following C program is:

```
void f1(int a, int b)
{
    int c;
    c = a; a = b; b = c;
}
void f2(int *a, int *b)
{
    int c;
    c = *a; *a = *b; *b = c;
}
```

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

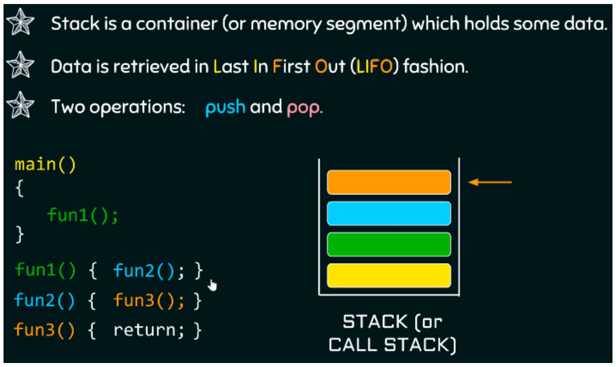
**Answer:** Very important. Do it again.

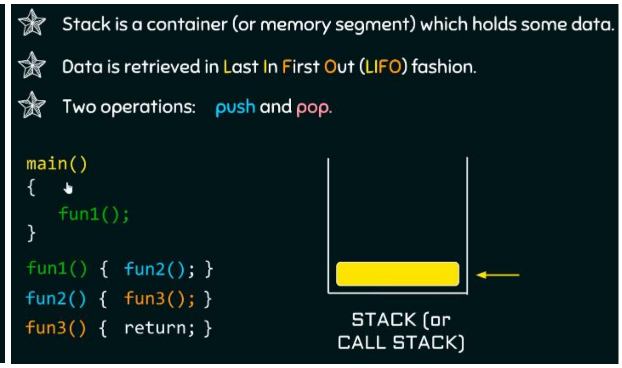
**Static Function:**

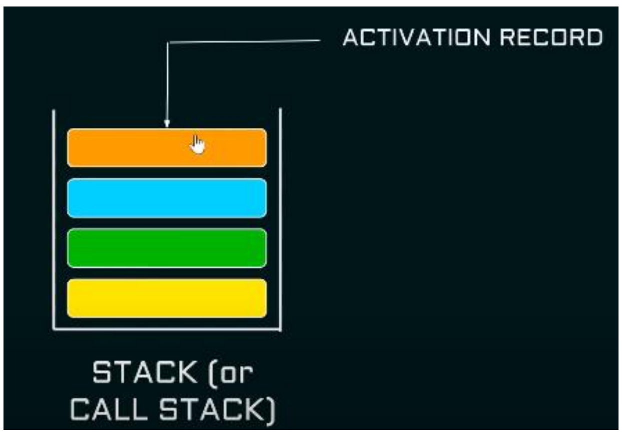
BASICS

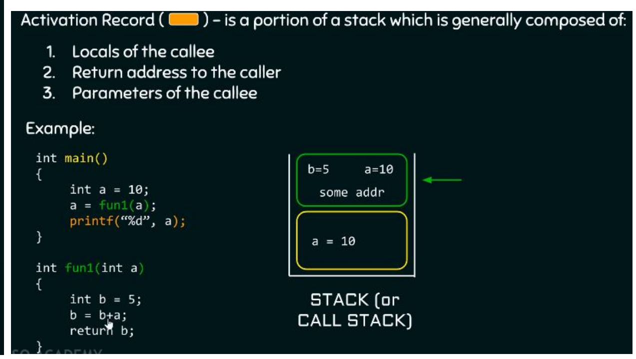
- In C, functions are global by default.
- This means if we want to access the function outside from the file where it is declared, we can access it easily.
- Now if we want to restrict this access, then we make our function static by simply putting a keyword static in front of the function.

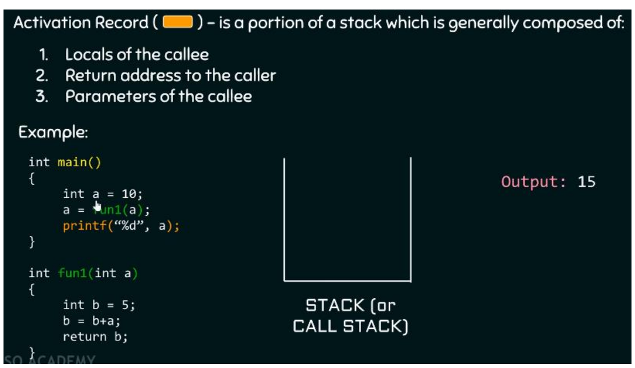
**Static scoping and dynamic scoping in C:**



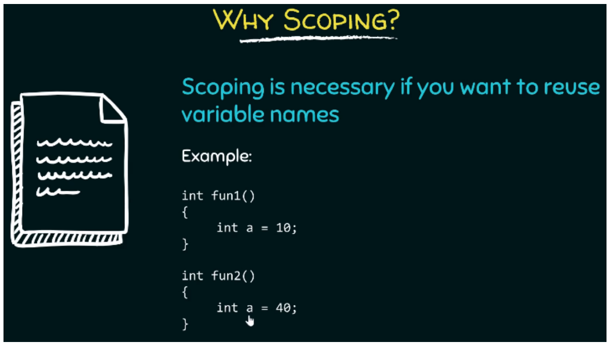



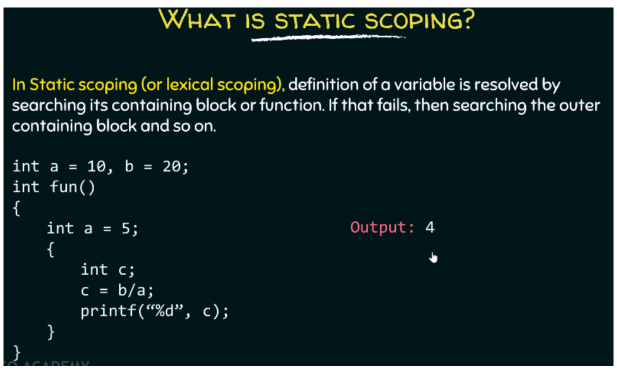



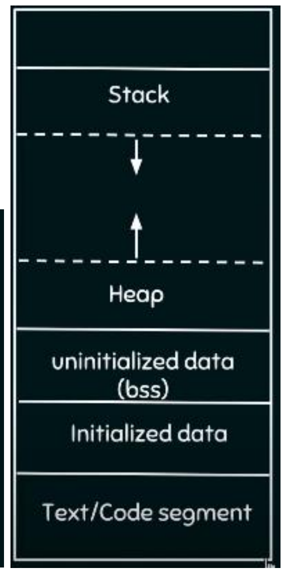
**Scoping:**



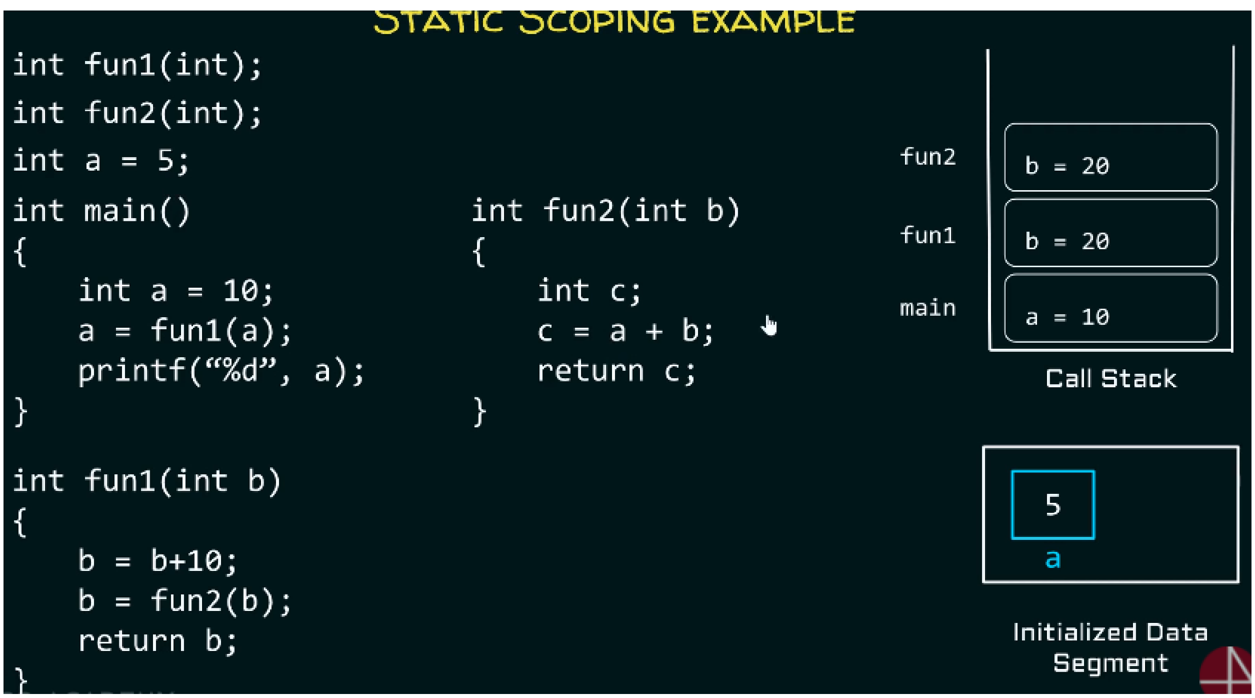
WHY SCOPING?
Scoping is necessary if you want to reuse variable names
Example:
int fun1()
{
int a = 10;
}
int fun2()
{
int a = 40;
}

WHAT IS STATIC SCOPING?
In Static scoping (or lexical scoping), definition of a variable is resolved by searching its containing block or function. If that fails, then searching the outer containing block and so on.
int a = 10, b = 20;
int fun()
{
int a = 5;
{
int c;
c = b/a;
printf("%d", c);
}
}
Output: 4

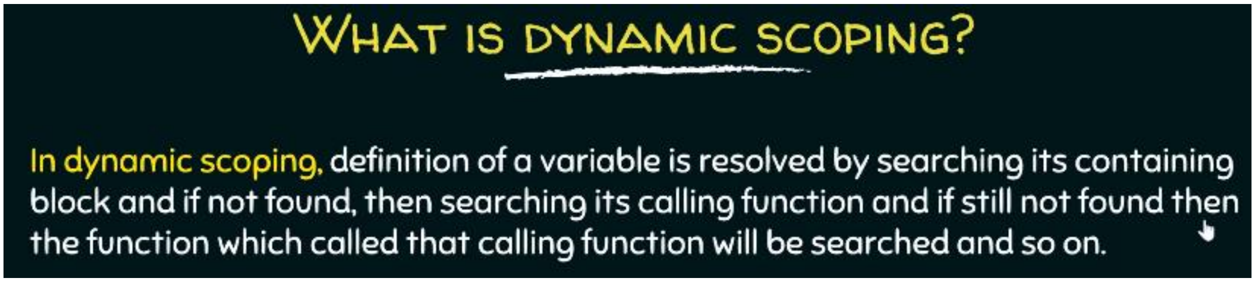



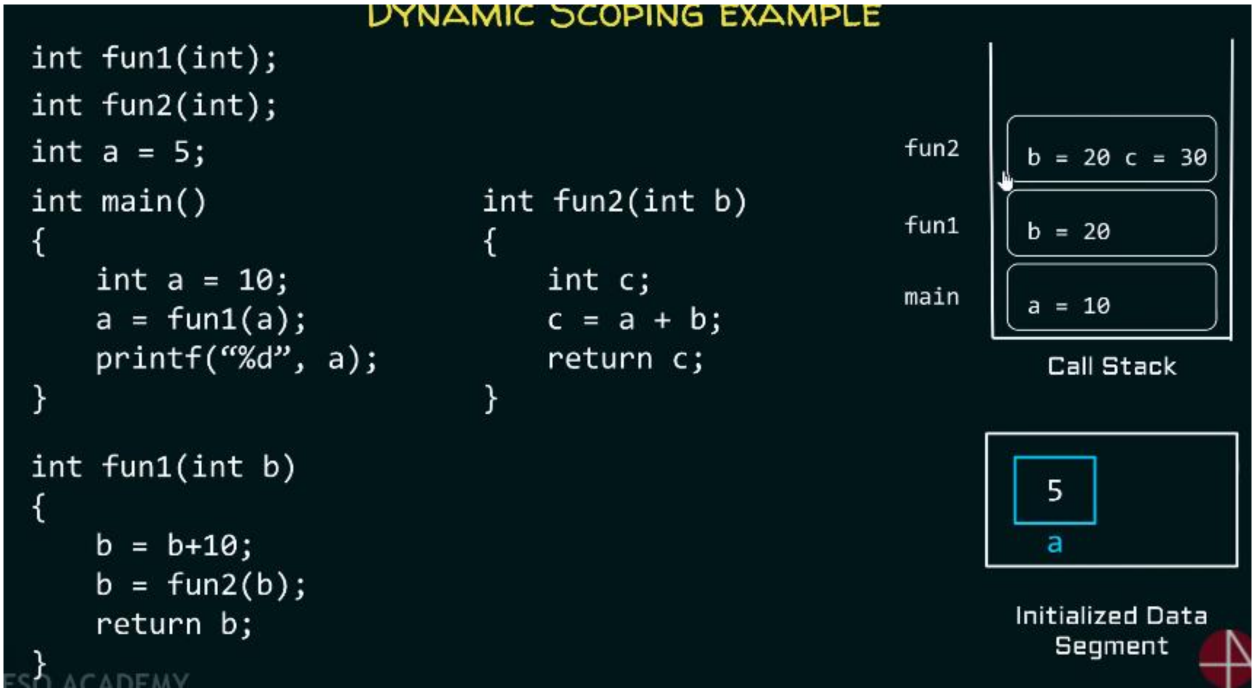
Stack
Heap
uninitialized data (bss)
Initialized data
Text/Code segment

STATIC 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;
}
int fun2(int b)
{
int c;
c = a + b;
return c;
}
fun2
fun1
main
b = 20
b = 20
a = 10
Call Stack
5
a
Initialized Data Segment

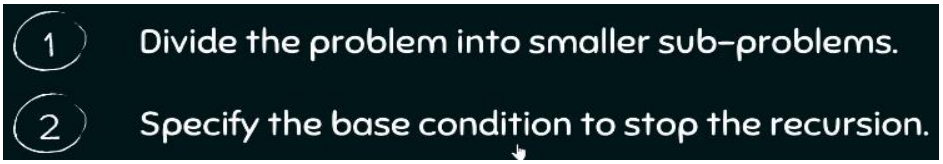
**Dynamic scoping:**

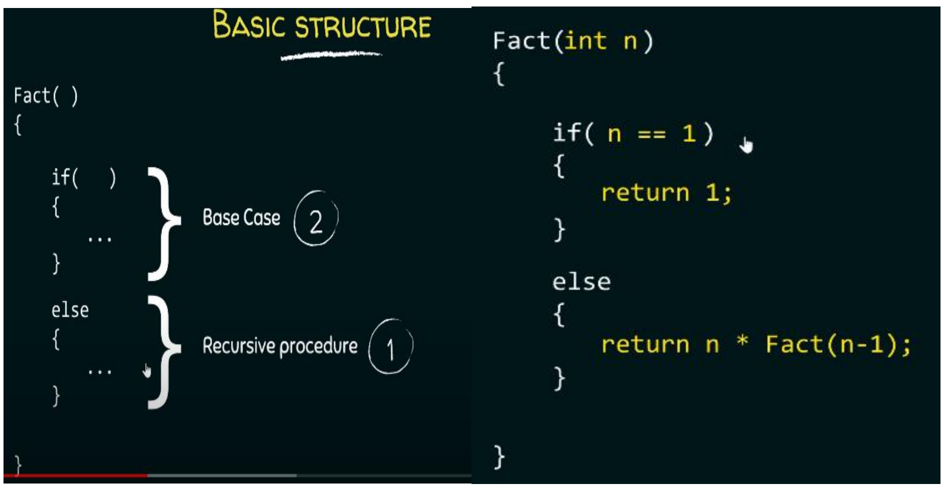


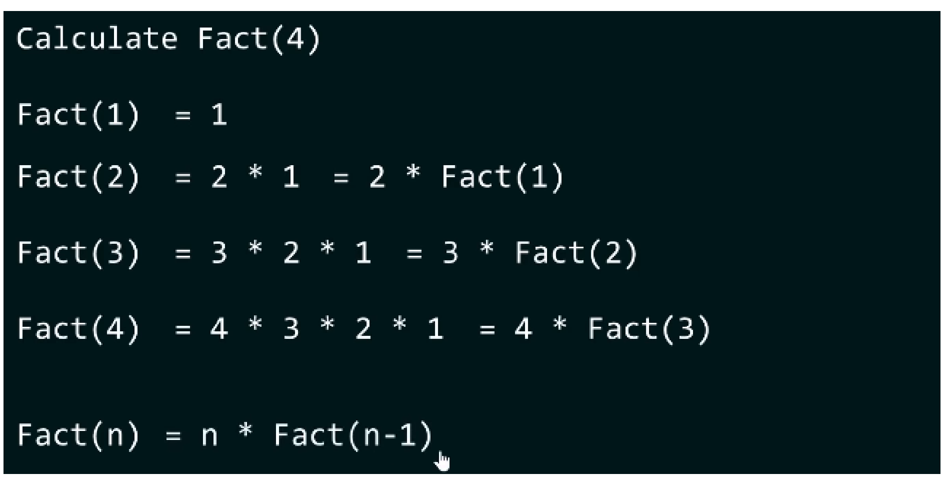
# Recursion:

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.

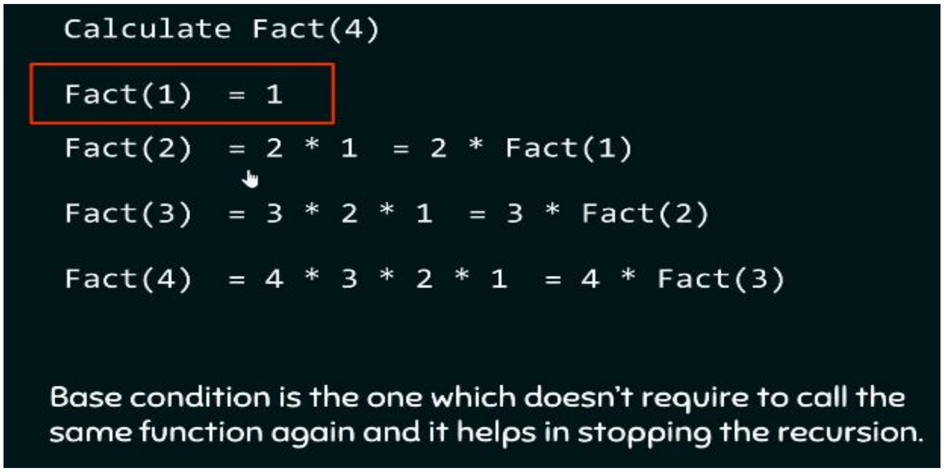


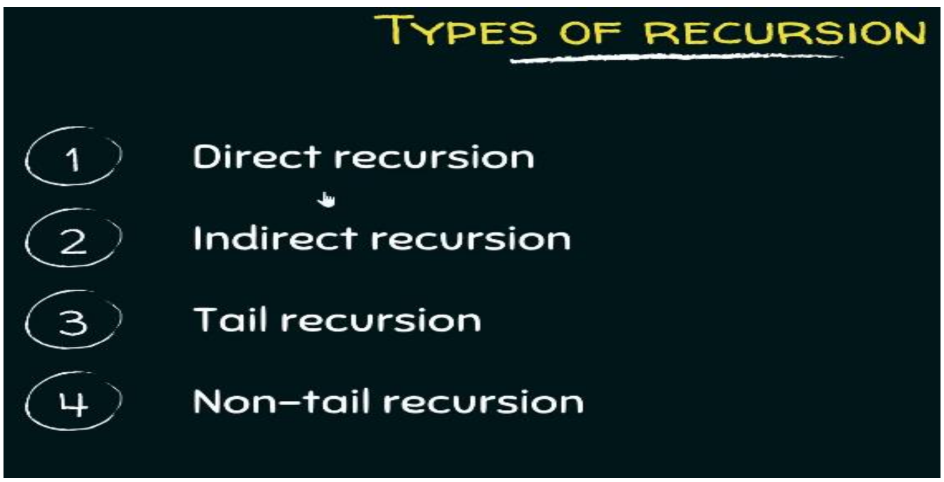

3. How to find **recursive procedure**: Look at the picture below.



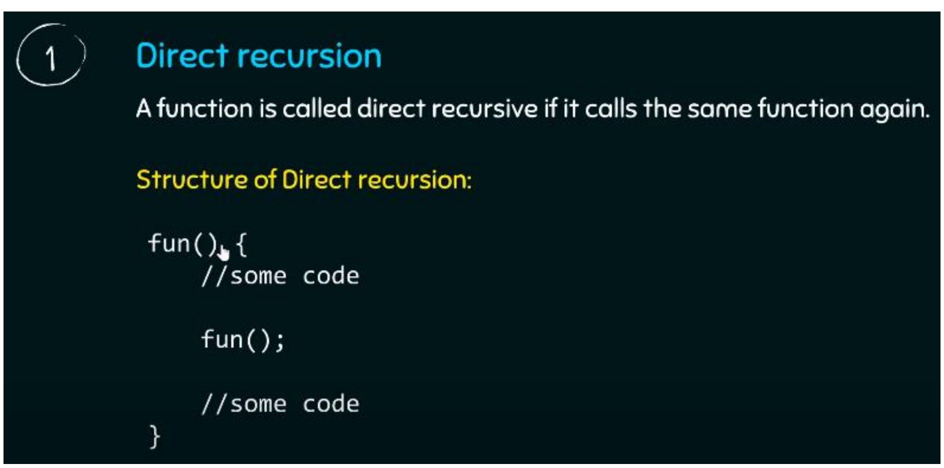

4. How to find the **base condition:** Condition which doesn't require to call the same **function again.**



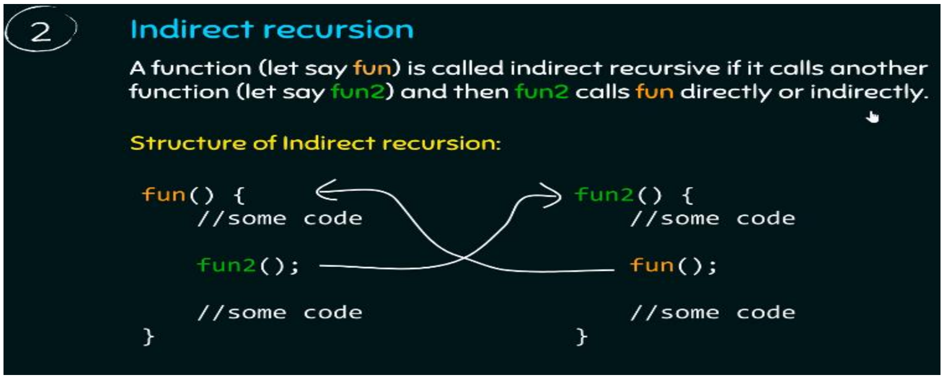

5. **Types of recursion:**

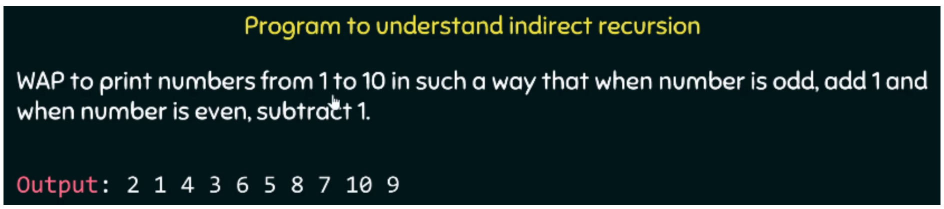



6. **Direct recursion:**

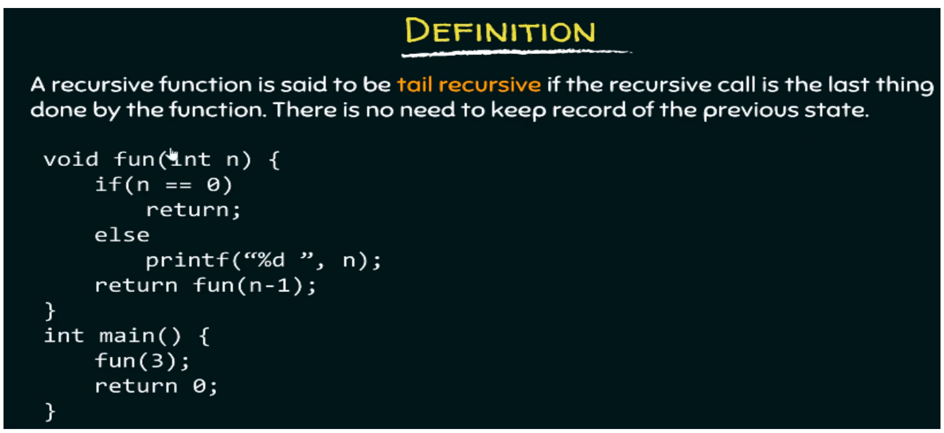


7. **Indirect recursion:**

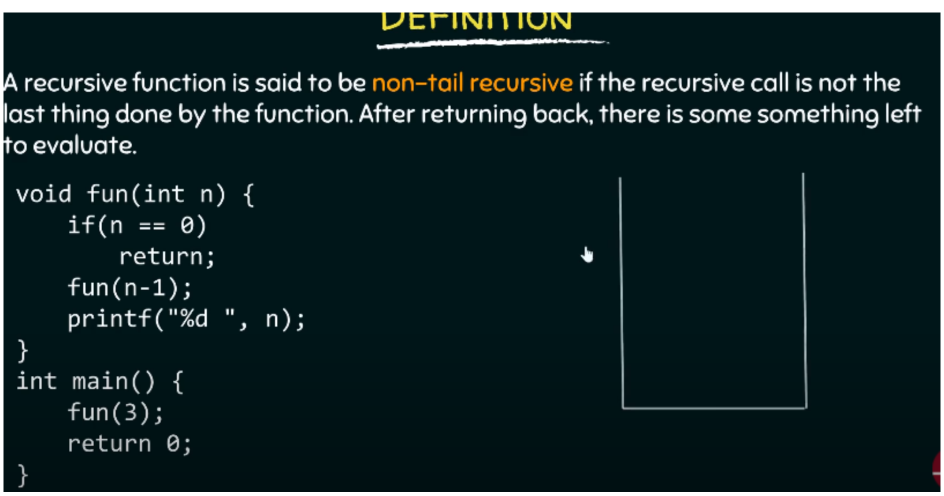



8. **Tail recursion:**




9. **Non tail recursive:**



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**:

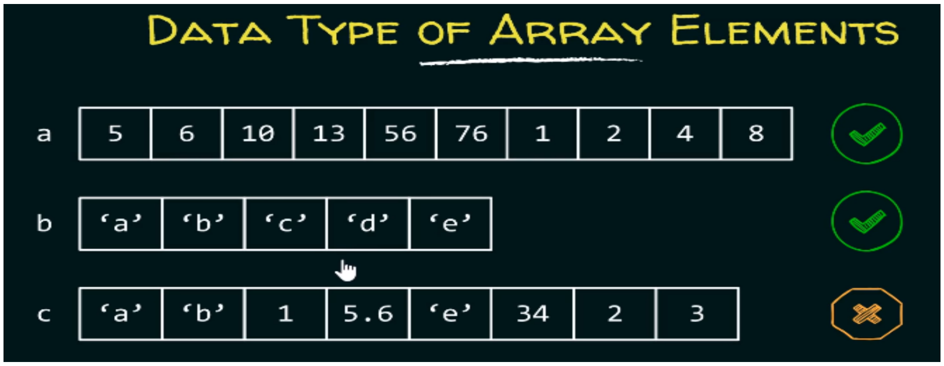
```
//
int sum(int n)
{
  if(n == 1)
  {return 1;}
  else
  {
    return n+sum(n-1);
  }
}

int main()
{
  int n,result;
  printf("Enter n : ");
  scanf("%d",&n);
  result = sum(n);
  printf("\nResult : %d",result);

}
```

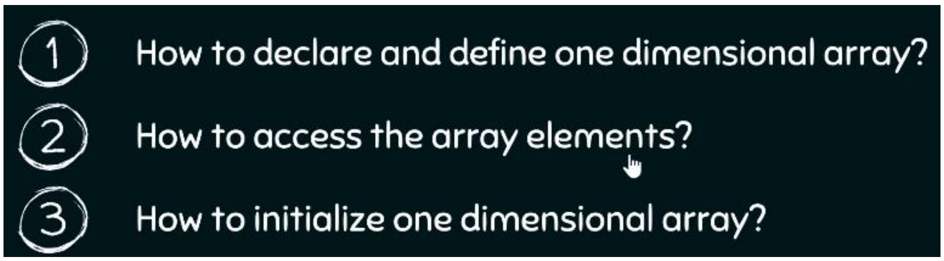
    ii. **Print** the **array** using **recursion:**

```
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);

}
```

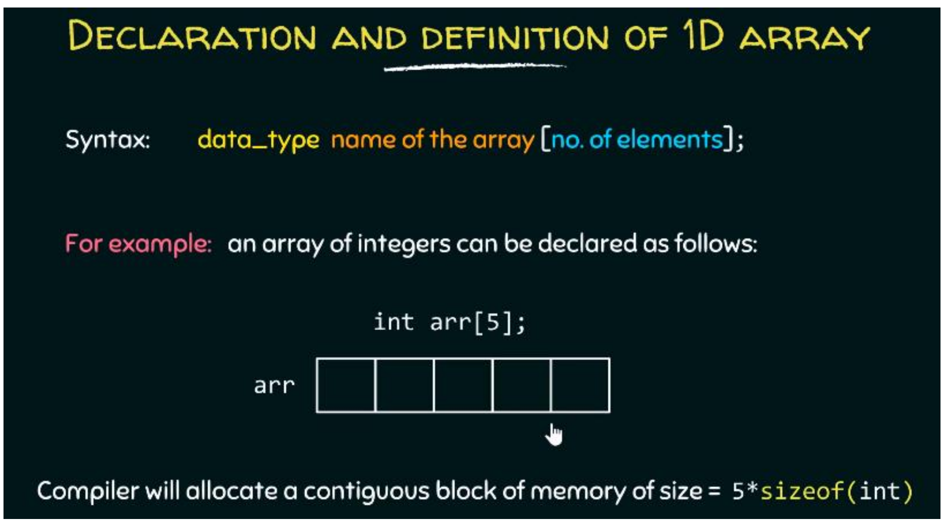
# Array 1-D:

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

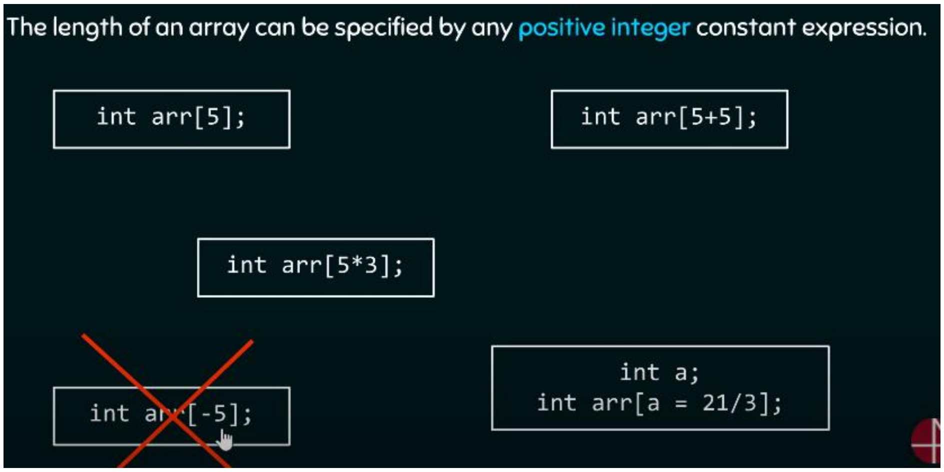


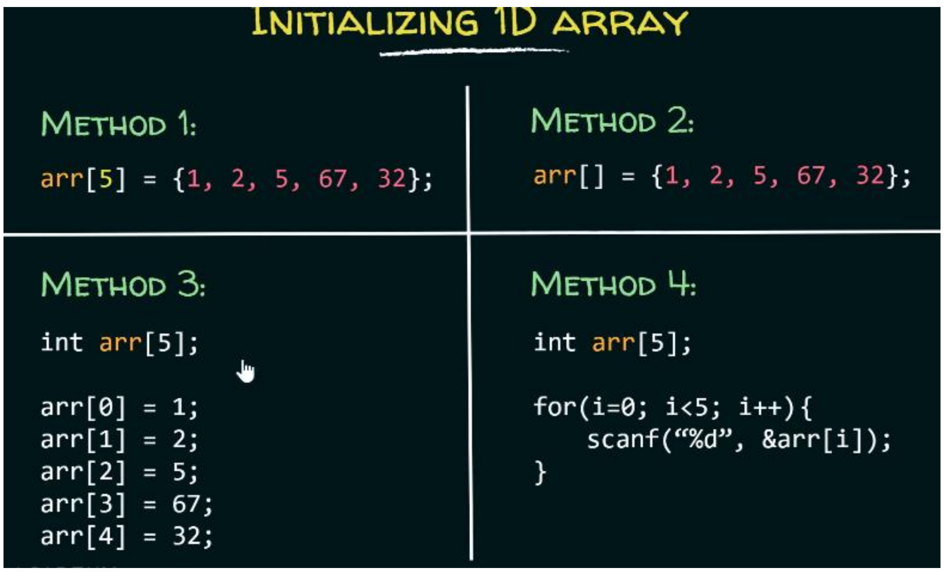

1. **Declaring an array:**



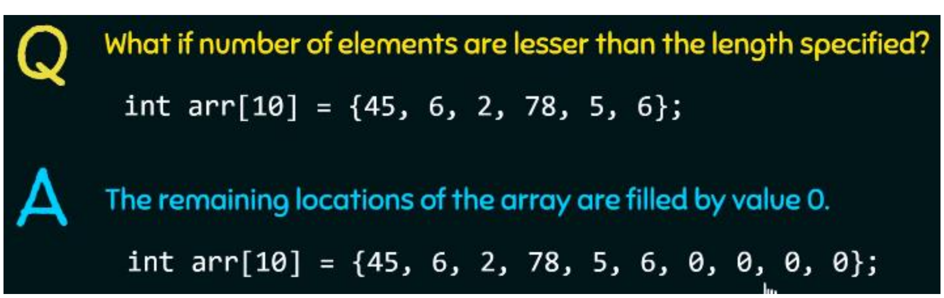

2. **Declaring size of an array:** Can be declared using a **positive constant value** or a **constant expression**.



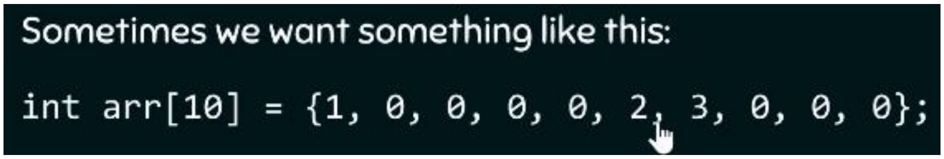

3. **How to define 1-D array:** In method 2, size is defined by the compiler since you've initialized the array already. You can't create an array without specifying the size of it.

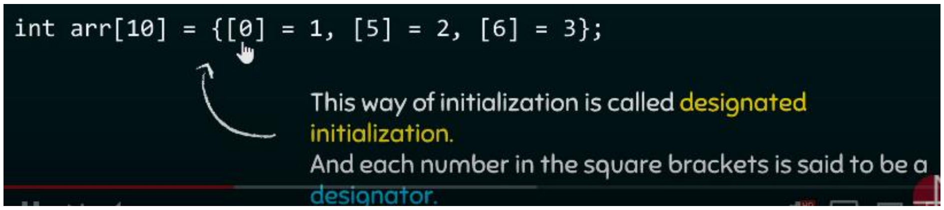


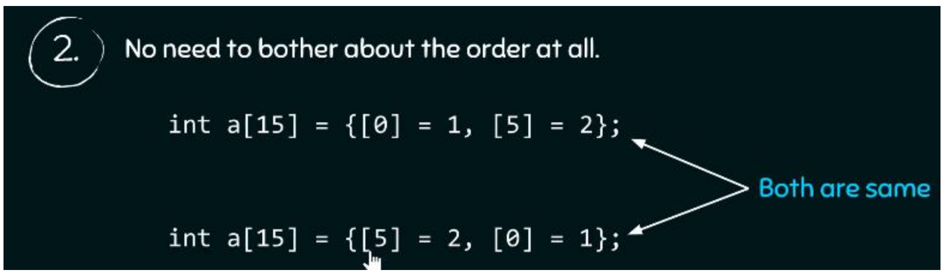

4. **Initializing the values in array:**
   i. **By default**, the **value** of an **array** is initialized to **garbage**.
   ii. **Partial initializing** leads to reset of the elements being filled as **0s**.
   iii. **To initialize to all zeros,** leave the curly bracket empty.

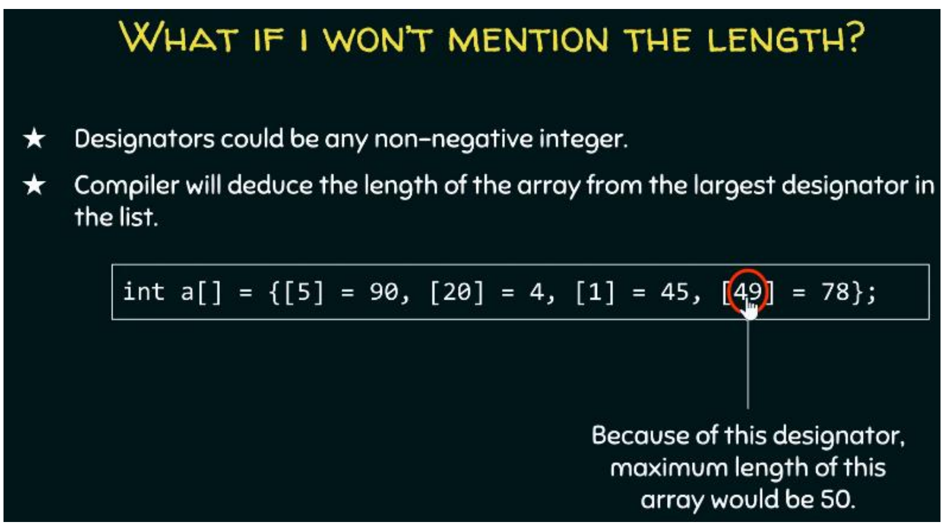


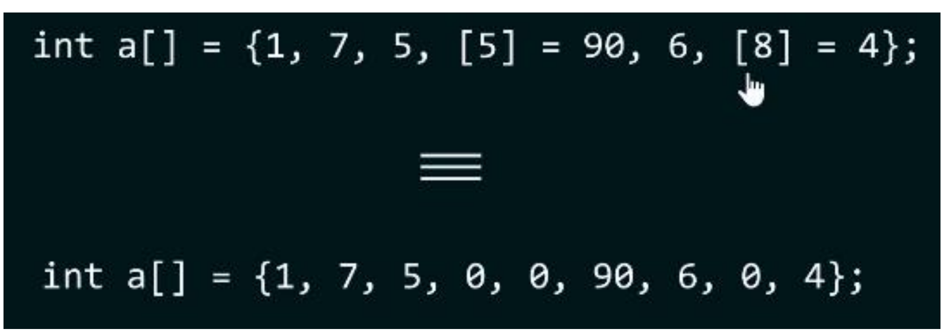
5. **What** to if we want something **like below?**



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

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

## Questions on Array 1-D:

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

```c
// 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.

```c
#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:**
   - Array[limit-1] = array[limit-2] ; limit—
   - Till **position +1;**
   - Insert new value at **position-1**

```c
#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 positon 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:**
   - Array[position-1] = array[position]; position++
   - Till **limit;**

```c
#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 positon 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.

```c
#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: Find Max Odd, if no odd, return max even. Take care of even numbers.**

```
int arr[] = {-3,-5,-4,-9};

//return me highest odd number : if no odd number is
ound return me even number
unsigned size = sizeof(arr)/sizeof(arr[0]);
int bufo = arr[0], bufe = arr[0];


for(int i = 1; i<size; i++)
{
  if(arr[i] %2 ==0)
  {
    if(arr[i]>bufe)
    bufe = arr[i];
  }
  if((arr[i]%2 == 1) || (arr[i]%2 == -1))
  {
    if(arr[i]>bufo)
    bufo = arr[i];
  }
}

//[1,2,4,3]
// bufe = 1; bufo = 1;
//if: even = bufe = 2; bufo = 1; i++
//if: even = bufe = 4; bufo = 1; i++
//if: odd =  bufo = 3; bufe = 4; i++

if((bufo%2 == 1) || (bufo%2 == -1))
  printf("Max Odd: %d", bufo);
else
  printf("Max Even: %d", bufe);
```

**Question 5: 2nd largest element in array.**

```c
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**.

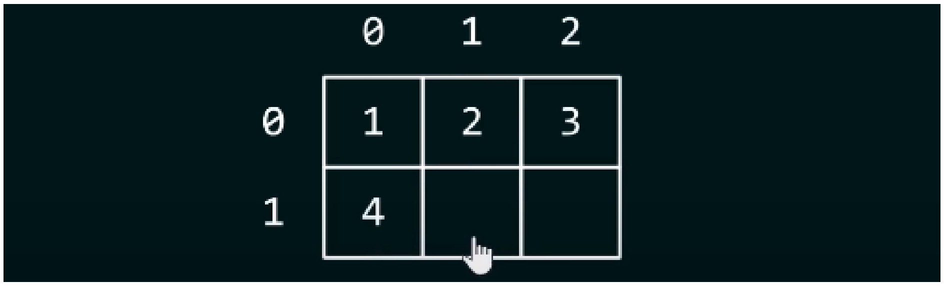
## Multi-dimensional Arrays: 2-D and 3-D:

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.

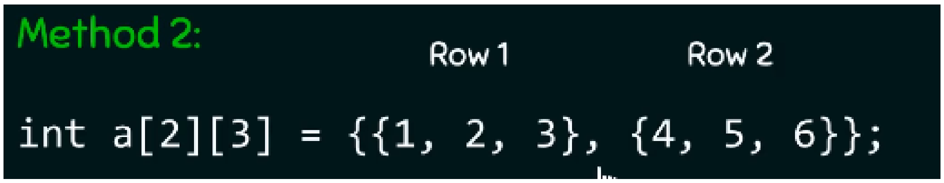

   - iii. How to initialize a 2D array? Initialize them like you will **initialize a normal 1-D array.**

```c
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.




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; <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:

```c
#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<rows; i++)
  {
    for(int j = 0; j<cols; j++)
    {
      arr3[i][j] = arr1[i][j]*arr2[i][j];
      printf("arr3[%d][%d]: %d\n",i,j, arr3[i][j]);
    }
  }
```

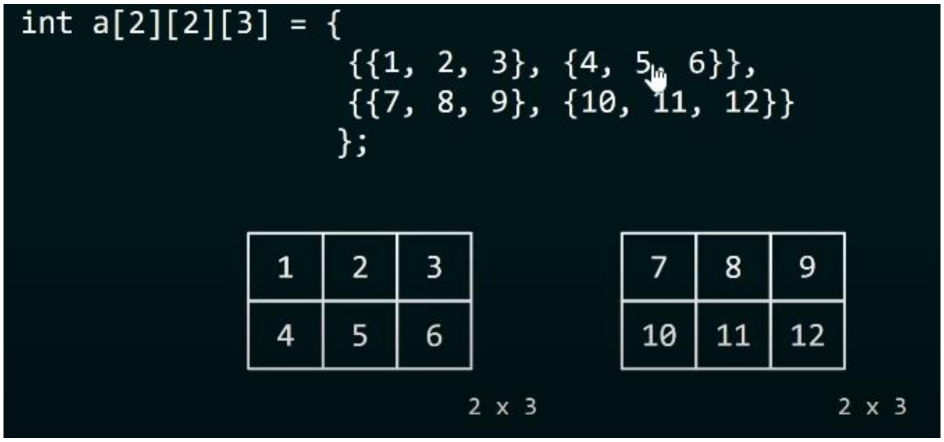
```
Harsh Dubey ran

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.

```c
#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]);
    }
  }
}

}
```

```
arr[0][0][0]: 0
arr[0][0][1]: 1
arr[0][0][2]: 2
arr[0][1][0]: 0
arr[0][1][1]: 1
arr[0][1][2]: 2
arr[1][0][0]: 0
arr[1][0][1]: 1
arr[1][0][2]: 2
arr[1][1][0]: 0
arr[1][1][1]: 1
arr[1][1][2]: 2
```

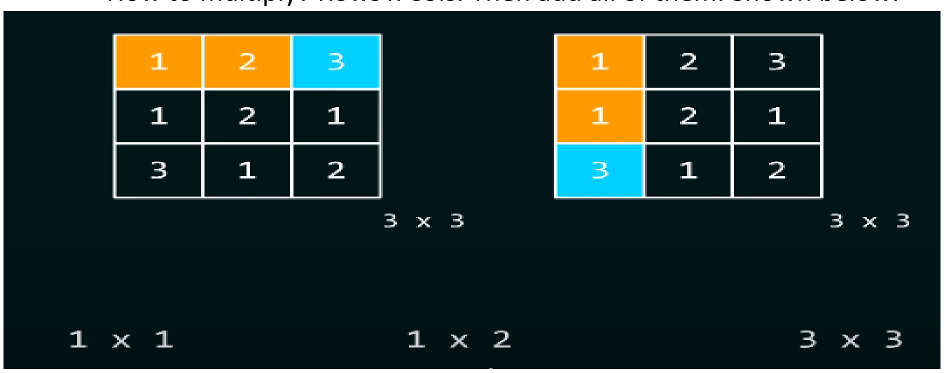



5. **Questions** on multi-dimensional arrays: **REMEMBER CRRC**

**i.** Write a program that will print sum of rows and cols in a 5x5 array.

**ii.** 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.




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

- Cols of 1st = rows of 2nd must be equal.
- **m1[rows][cols] x m2[rows][cols] =** Matrix [**rows of m1**] [**cols of m2**].
- **Matrix [0][0] =** Sum (m1 row0 x m2 col0)

**iv.** Write a program that multiplies two matrices:

```
//logic for matrix mutiplication
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.

```c
#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.**

```c
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.
   - **i.** **Variable length** arrays **can't be initialized** and **declared** at the **same time.**
   - **ii.** 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$^{st}$ 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 derefrencing : %d\n", *ptr);
```

```
Address of num using '&' : 0x10cd81e0
Address of num using pointer : 0x10cd81e0
Value of num by derefrencing : 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 derefrencing ptr2 : %d\n",
*ptr2);
```

```
solution.c:22:46: warning: format '%x' expec
 int', but argument 2 has type 'int *' [-Wfo
   printf("Address of num using pointer 2 :

Address of num using '&' : 0xce762340
Address of num using pointer 1 : 0xce762340
Address of num using pointer 2 : 0xce762340
Value of num by derefrencing 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.

```c
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
```

ii. **Another example of changing values by passing them by reference:**

```c
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;
}
```

```
My min: 1
My max: 9

Harsh Dubey

My min: 0
My max: 100
```

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

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

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

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

  summation(ptr);

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

}
```

```
a : 6
```

ii. **Receiving a pointer:**

```c
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;


}
```

```
 int', but argument 2 has type 'i
   printf("Mid value address is:


solution.c:20:32: warning: format
 int', but argument 2 has type 'i
   printf("Address of arr is: %#x
                              ~~^

Mid value is: 300
Mid value address is: 0x1350d7ec
Address of arr is: 0x1350d7e0
Value of byte is: 0X2C
Value of byte is: 0XFFFFFFD3
New Mid value is: 467
```

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.**

```c
#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 precesion of -4.25 BYTE[%d] is
%#x\n",i, *(pc+i));
  }
}
```

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

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


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

Double precesion of -4.25 BYTE[0] is 0
Double precesion of -4.25 BYTE[1] is 0
Double precesion of -4.25 BYTE[2] is 0
Double precesion of -4.25 BYTE[3] is 0
Double precesion of -4.25 BYTE[4] is 0
Double precesion of -4.25 BYTE[5] is 0
Double precesion of -4.25 BYTE[6] is 0x11
Double precesion of -4.25 BYTE[7] is 0xffffffc0
```

9. **Convert floating point into binary:**

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

  //coverting 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;
  }

  //coverting 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:**

```c
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:**

```c
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:**

```c
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.

```c
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.**

```c
void change(int arr[], unsigned size)
{
  for(int i = 0; i<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<size; i++)
  {
    printf("%d\n",arr[i]);
  }
    printf("Size: %d", size);

}
```

```
Size: 6
2
4
6
8
10
12
Size: 6
```

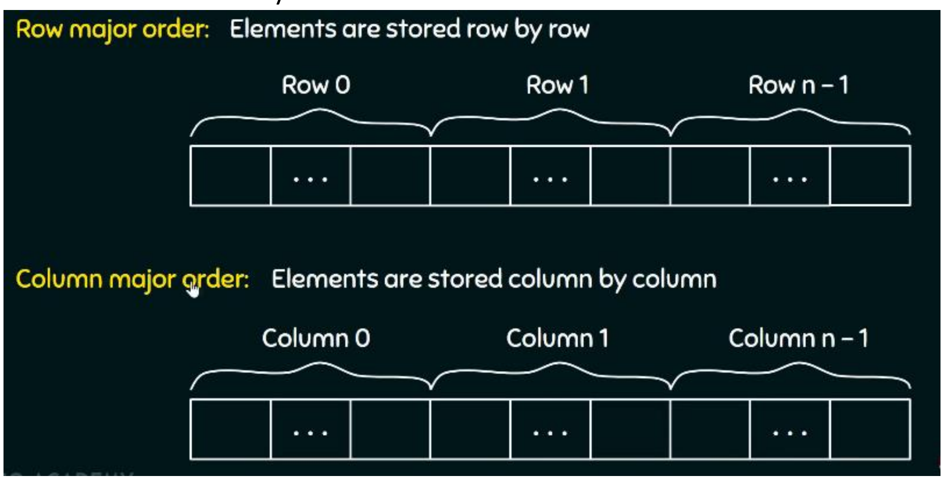
- **Can we pass array elements as values to the function: Yes,** it will behave exactly as pass by value? Look at the example below.

```
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.




- **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 1

Harsh Dubey ran 32

1 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!!**

```
 int arr[2][rows][cols] = {{{1,2},{3,4}},
                           {{5,6},{7,8}}

                           };

int*ptr = **arr;
 printf("%d \n",*(ptr));
```

```
Harsh Dubey

1
```

**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.

```c
#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 **1st element**. Example with memory layout. However, **it's a self-pointer.** The if we create a **pointer to the array,** and it's not a self-pointer, as shown below.

```c
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 anme
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 | **a** |
| 51 | 101 | *p |
| | | |

**ii.** **Let's analyze the 8 results:**

```c
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",&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 1st 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 1st 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**.

```c
// 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.

```c
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:**

```c
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)**

```c
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.

```c
int a = 300;

void*ptr = &a;

printf("%p\n", ptr);
printf("%p", ptr+1);
```

```
0x7ffd8f20d540
0x7ffd8f20d541
```

   **However,** you can't **de-refer** it.

```c
int a = 300;

void*ptr = &a;

printf("%p\n", *ptr);
printf("%p", *(ptr+1));
```

```
solution.c:11:16: error: invalid use of void express
solution.c:12:14: warning: dereferencing 'void *' po
 printf("%p", *(ptr+1));
              ^~~~~~~~
solution.c:12:14: error: invalid use of void express
```

   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;
printf("%p\n", ptr);
```

```
(nil)
```

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 1st 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 is 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));**

a. **If (x == 0):** Arrays are same.
b. **If(x > 0): a[] > b[]**
c. **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 1st.

```
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 l

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 memcopy function.**

```c
void memcopy(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};

 memcopy(&a,&b, sizeof(char));

for(int i = 0; i< sizeof(a)/sizeof(a[0]); i++)
  {
    printf("%d ", a[i]);
  }

}
```

```
Harsh Dubey ran

1 5 8 4 5

Harsh Dubey ran

300 302 303 4 5

Harsh Dubey ran

44 2 3 4 5
```

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.**

```c
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] );

}
```

```
Harsh Dubey r

Successful!!
A: 247
```

**In the code above,** look at the **answer carefully.** We changed the byte one of $2^{nd}$ element of an array to -9. Why is the answer 247 then? Because we only changed the $1^{st}$ 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.

```c
#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
Stack empty !
Stack: -1
Stack empty !
Stack: -1
Stack empty !
Stack: -1
Stack empty !
Stack: -1
Stack empty !
Stack: -1
```

**20. Queue using pointers:** pushptr and popptr

i. Pushprt > 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 avalibale 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;
}
```

**21. Function pointer:**

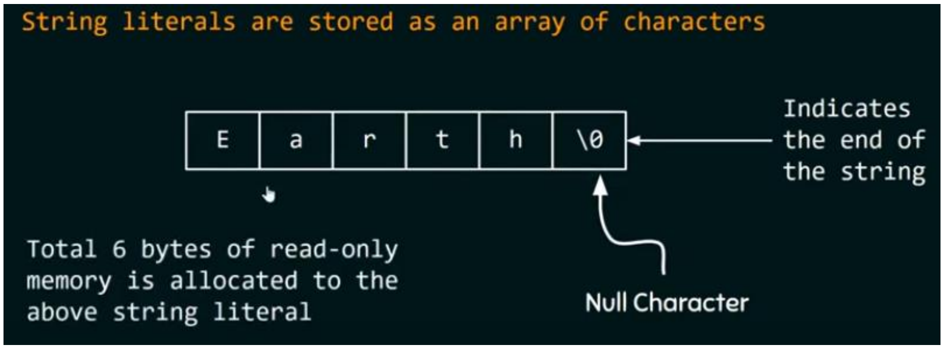
## Strings: Constant vs Variable

1. **String constant and string variable** are **both C style string, that is both end with a "\n".** The only difference is **string constant is read only and string variable is both read and write.**
2. **String Constant: Sequence of characters** enclosed in **double quotes. "%s"** is used to print the **string literal.**

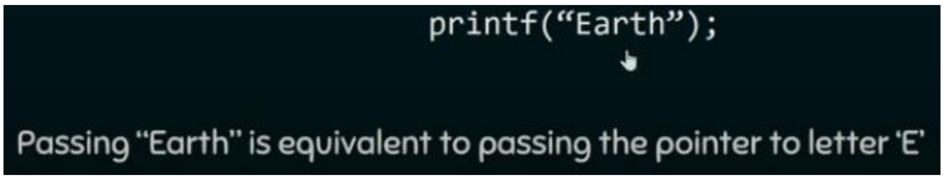
```
int main()
{
  printf("%s", "No matter what you chose, ""\nyou'll
regret everything");
}
```

```
No matter what you chose,
you'll regret everything
```

3. **Printf(): int printf( const char*ptr):** it takes in the **const char pointer** and returns number of character printed in the screen. Hence, anytime we **print a string literal** using **printf()** we take in **const char*ptr** to the first character of the string constant and not the whole string constant. A **string constant** or a **string literal** in C is stored as a character array, shown below.




**Every string literal** ends with a “\0” character and is read only.




When we are passing in the pointer to the string literal, and we will traverse the string until we reach the “\n”.

4. **Pointers to string literal:** We know that in C, we store string literals as **read only memory and** then we create a **pointer to the 1st character** of the **string literal.**

```
int main()
{
  char*str = "Harsh Dubey";

  printf("%s\n",str);
  printf("%c\n",str[0]);
  printf("%c\n",str[1]);
}
```

```
Harsh Dubey
H
a
```

**We can’t change** the string literal**, it’s read-only.**

```
int main()
{
  char*str = "Harsh Dubey";

  printf("%s\n",str);
  printf("%c\n",str[0]);
  printf("%c\n",str[1]);

  str[0] = 'k';
}
```

```
Harsh Dubey
H
a
AddressSanitizer:DEADLYSIGNAL
=================================================
==14==ERROR: AddressSanitizer: SEGV on unknown addre
 0x55d0cc3de514 bp 0x7ffe811e4760 sp 0x7ffe811e4750
==14==The signal is caused by a WRITE memory access.
    #0 0x55d0cc3de513 in main /home/coderpad/solutic
    #1 0x7fc1ecb9fb96 in __libc_start_main (/lib/x86
    6+0x21b96)
```

5. **String literal and Character constant:** String literal and character constants are not same. String literal is a pointer to the first character and character is a number (ASCII). Also, string literal comes with a “\0”.

```
#include <stdlib.h>

int main()
{
  char*str = "H"; //This comes with a "\0"
  char c = 'H';  //This is just ASCII code

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

}
```

```
H
H
```

If we try to **print the character** using **“%s”,** then we’ll get a **segmentation fault**. Since, printf() will look for a “\0” and won’t find it and this is also illegal memory access.

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

int main()
{
  char*str = "H"; //This comes with a "\0"
  char c = 'H';  //This is just ASCII code

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

}
```

```
harsh dubey ran 15 lines of C:(
solution.c: In function 'main':
solution.c:10:12: warning: form
 but argument 2 has type 'int'
   printf("%s\n",c);
           ~^    ~
           %d
H
AddressSanitizer:DEADLYSIGNAL
================================
==14==ERROR: AddressSanitizer:
 0x559b5e4f7272 bp 0x7ffe207c92
```

6. **Variable strings:** Create a character array but always remember to create the size 1 byte longer than the string length. To make room for NULL terminator. If you don't do it then there will be a segmentation fault bc printf() will look for that NULL terminator.

```c
int main() {

  char c[6] = "Harsh";
  printf("%s",c);
```

```
Harsh Dubey

Harsh
```

7. **Segmentation fault when byte for "\0" is not set aside:**

```c
int main() {

  char c[5] = "Harsh";
  printf("%s",c);
```

```
=====================================================
==15==ERROR: AddressSanitizer: stack-buffer-overflow
b8c55 at pc 0x5615b7bf3476 bp 0x7ffcd72b8ae0 sp 0x7f
READ of size 7 at 0x7ffcd72b8c55 thread T0
    #0 0x5615b7bf3475 in printf_common(void*, char c
```

8. **We can also assign strings like this:**

```c
int main() {

  char c[5] = {'H','a','r','s','\0'};
  printf("%s",c);
```

```
Hars
```

Again, remember to manually add "\0" in string. When declaring like figure 6, the compiler takes care of the null terminator. So, you don't have to worry about it. But when you are making a character array then you have to manually add the null terminator.

9. **Printing strings:** Printf and puts

```c
int main() {

  char*str = "Harsh";//read only
  char c[] = "Dubey";//r and w

  //basic printing
  printf("%s\n", str );
  printf("%s\n", c );

  //more coded printing
  printf("%10s\n", str );
  printf("%15s\n", c );
  printf("%15.2s\n", c );
```

```
Harsh
Dubey
     Harsh
          Dubey
             Du
```

**As, we can see,** we can use printf to print the number of characters on the screen, we can also add abbreviation to the printf function and control the indentation of the printed string.

```c
char*str = "Harsh";//read only
char c[] = "Dubey";//r and w

puts(str);
puts(str);
```

```
Harsh
Dubey
     Harsh
```

**Int Puts(const*char ptr)** function does the same, it takes in the const ptr and returns the number of integer printed on the screen. However, it automatically prints a new line character, and you can't really program any indentation.

10. **Reading strings: Using Scanf and gets function.**
    Biggest issue is scanf won't scan beyond "\n", we can fix it by adding "[^\n]" and we can fix the boundaries of scanf by specifying the numbers of characters to be accepted. However, still the issue persists of taking multiple inputs of strings from the user. That has to be fixed by flushing the buffer.

```
int main() {

  char c1[10],c2[10];

  //ask for string 1
  printf("Enter string: ");
  scanf("%9[^\n]s", c1);
  printf("\n%s", c1);

  //flush the buffer: wait till the enter key is taken
  while(getchar() != 10);

  //ask for another string
  printf("\n\nEnter string 2 : ");
  scanf("%9[^\n]s", c2);
  printf("\n%s", c2);
```

```
Enter string: Harsh Dubey
Harsh Dub
Enter string 2 : Harsh Dubey
Harsh Dub
```

11. **If you don't want to flush the buffer again and again the design your own scanning function:**

```
int scanstr(char arr[], int size )
{

  int ch, i = 0;

  while((ch = getchar()) != '\n')
  {
    if(i<size)
    {
      arr[i] = ch;
      i++;
    }arr[size] = '\0';
  }

  return i;
}
```

**Question: Print all the alphabets on the screen:**

```
for(int i = 65; i<92; i++)
   printf("%c ",i);
```

12. **String functions: #include <string.h>**

i. **Strcpy(): char*strcpy(char*destination, const char*source)**

```
char str[10] = "Harsh";
char*str2 = "Harsh Dubey";

//strcpy(str, str2);
strncpy(str, str2, 9);
printf("%s",str);
```

```
Harsh Dub
```

ii. **Strlen(): size_t strlen(const char*str):**

Takes the pointer to a string and returns the size of it without counting the NULL terminator.

```
int main() {
    char str[100] = "Hello World";
    printf("%d", strlen(str));
    return 0;
}
```

OUTPUT: 11

It calculates the length of the string and not the length of the array.

As, we can see, it returns 11, counting the space and ignoring the \0. Also, array size is 100 but string length returned is 11. Since, it only counts the string length.

iii. **Strcat() & strncat():**

Prototype: `char* strcat(char* str1, const char* str2);`

Strcat function appends the content of string str2 at the end of the string str1

```
char str[12] = "Harsh";
char*str2 = " Dubey";

strcat(str, str2);
printf("%s",str);
```

```
Harsh Dubey
```

**Now,** if you look above, we concatenated the two strings, look at the size of str[12], we have to have enough size to merge the string. If we don't the segmentation fault will happen.

```
char str[10] = "Harsh";
char*str2 = " Dubey";

strcat(str, str2);
printf("%s",str);
```

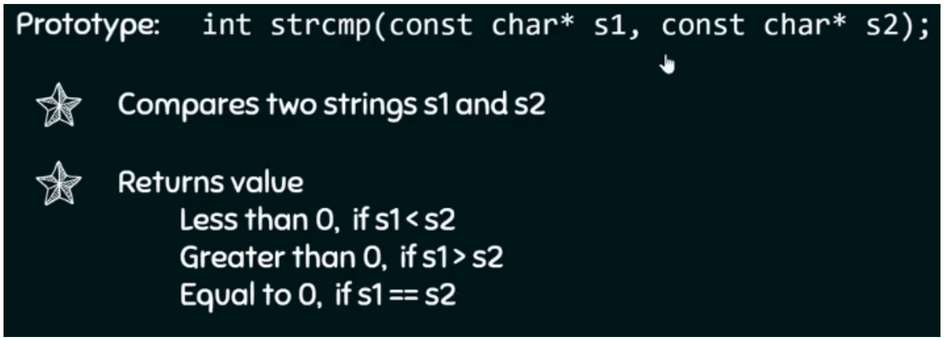
```
==17==ERROR: AddressSanitize
ae40a at pc 0x5631c1ffe3ed b
WRITE of size 7 at 0x7fff90a
    #0 0x5631c1ffe3ec in str
    #1 0x5631c20f14e8 in mai
```

What if we don't have any control over the second string? Then use strncat():

```
char str[10] = "Harsh";
char*str2 = " Dubey";

//strcpy(str, str2);
strncat(str, str2, sizeof(str)-strlen(str)-1);

printf("%s",str);
```

```
Harsh Dubey

Harsh Dubey

Harsh Dub
```

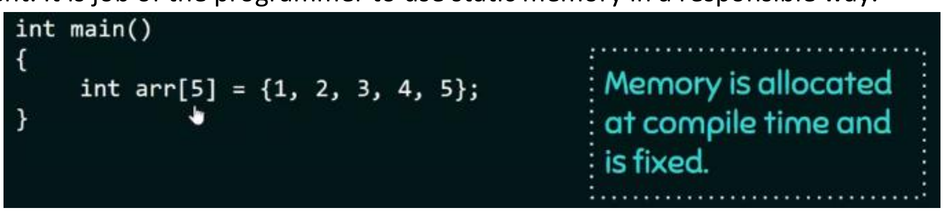
iv. **Strcmp():**




## Memory Allocation:

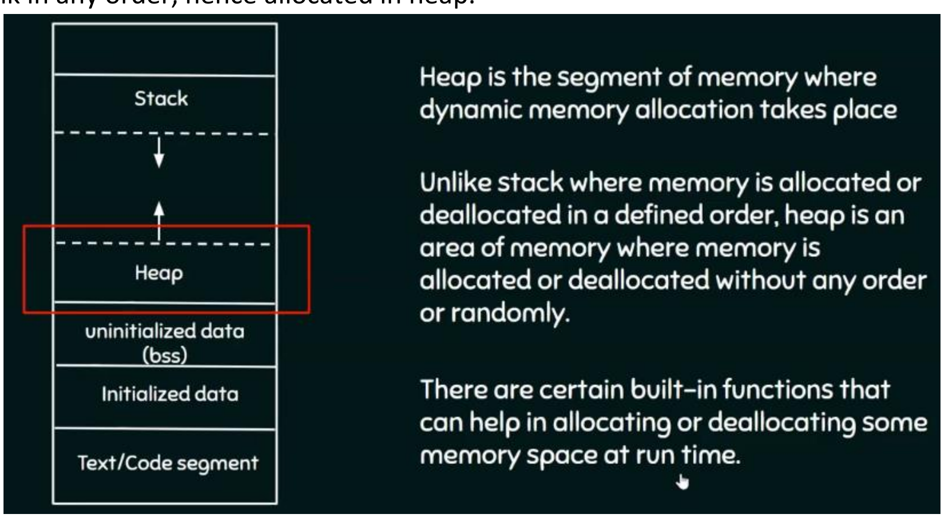
Memory allocation are two kinds: One – Static and Dynamic.

1. **Static memory allocation:** Memory allocated during compile time is basically allocated statically in **stack**. The static memory allocation is defined and can't be changed during run-time. It is permanent. It is job of the programmer to use static memory in a responsible way.




**Basically,** what you are doing is, you are allocating memory in a particular order in **stack**.

2. **Dynamic memory allocation:** We are dynamically allocating memory, which means it can grow and shrink in any order, hence allocated in heap.




3. **Functions used in Dynamic memory allocation: LAF:** #include<stdlib.h>
   i. **Malloc(): void* malloc(size_t): Memory allocation:** Allocates number of **size_t bytes** of **contiguous memory** and if it **successfully allocates the memory** then returns a **void pointer** to the **first byte** of it otherwise it **returns NULL. Also, size_t =** unsigned integer.

**Example:** Prompt user to enter number of **shorts** to store and then dynamically allocate the memory to the **shorts.**

```
int main() {
  //dynamically allocatign memory
  unsigned short n;
  printf("Enter number of shorts:\n");
  scanf("%hu", &n);

  unsigned short*ptr = (unsigned short*)malloc(sizeof(unsigned short)*n);

  //check if you were able to allocate memory or not
  if(ptr ==NULL)
  {
    printf("Memory not available!");
    exit(1);
  }

  //where is the &
  printf("\nEnter the values:\n");
  for(int i = 0; i< n; i++)
    scanf("%hu", ptr+i);//scanning it but still no "&" bc it's a ptr

  for(int i = 0; i< n; i++)
    printf("%d\n", i[ptr]);

  //freeing the memory
  free(ptr);

}
```

**Now what if we,** allocate 3 shorts and scan only 2 form the user and then print all 3 shorts. **We'll print** garbage **value.**

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

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

int main() {
  //dynamically allocatign memory
  unsigned short n;
  printf("Enter number of shorts:\n");
  scanf("%hu", &n);

  unsigned short*ptr = (unsigned
short*)malloc(sizeof(unsigned short)*n);

  //check if you were able to allocate memory or not
  if(ptr ==NULL)
  {
    printf("Memory not available!");
    exit(1);
  }

  //where is the &
  printf("\nEnter the values:\n");
  for(int i = 0; i< n-1; i++)
    scanf("%hu", ptr+i);//scanning it but still no "&"
bc it's a ptr

  for(int i = 0; i< n; i++)
    printf("%d\n", i[ptr]);

  //freeing the memory
  free(ptr);

}
```

```
harsh dubey ran 33 lines of C

Enter number of shorts:
3

Enter the values:
1
2
1
2
48830
```

ii. **Calloc(): void* calloc(size_t number, size_t bytes): Clear allocation:** Allocates **"n" number of bytes** and returns a void ptr to it , if not successful then returns **NULL. The difference is** in calloc() we get allocated memory **assigned to default "0" unlike malloc().**

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

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

int main() {
  //dynamically allocatign memory
  unsigned n;
  printf("Enter number of unisgned int:\n");
  scanf("%u", &n);

  unsigned short*ptr = (unsigned short*)calloc(n,
sizeof(unsigned short));

  //check if you were able to allocate memory or not
  if(ptr ==NULL)
  {
    printf("Memory not available!");
    exit(1);
  }

  //where is the &
  printf("\nEnter the values:\n");
  for(int i = 0; i< n-1; i++)
    scanf("%hu", ptr+i);//scanning it but still no "&"
bc it's a ptr

  printf("\nThe new values:\n");
  for(int i = 0; i< n; i++)
    printf("%d\n", i[ptr]);

  //freeing the memory
  free(ptr);

}
```

```
harsh dubey ran 34 lines of C

Enter number of unisgned int:
4

Enter the values:
1
2
3

The new values:
1
2
3
0
```

iii. **Realloc(): void*realloc(void*ptr, size_t size): re- allocate memory:** without losing the old data. **First argument** takes in the pointer to the old memory and second arguments takes in the **number of bytes to reallocate. Remember** it will “re” allocate the memory, it’s not adding memory to the old structure.

```c
int *ptr = (int *)malloc(sizeof(int));

ptr = (int *)realloc(ptr, 2*sizeof(int));
```

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

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

int main() {
  //dynamically allocatign memory
  unsigned n;
  printf("Enter number of unisgned int:\n");
  scanf("%u", &n);

  unsigned short*ptr = (unsigned short*)calloc(n, sizeof(unsigned short));// 2 * 16 = 4 bytes

  //check if you were able to allocate memory or not
  if(ptr ==NULL)
  {
    printf("Memory not available!");
    exit(1);
  }

  //where is the &
  printf("\nEnter the values:\n");
  for(int i = 0; i< n; i++)
    scanf("%hu", ptr+i);//scanning it but still no "&" bc it's a ptr  // 2  bytes

  printf("\nThe new values:\n");
  for(int i = 0; i< n; i++)
    printf("%d\n", i[ptr]);

  int*p = (int*)realloc(ptr, 4);// 8 bytes
  printf("\nThe new values:\n");
  for(int i = 0; i< 1; i++)// 4
    printf("%d\n", i[p]);

  //can't free ptr: it was freed by realloc
  free(p);
  p = NULL; //to avoid dangling pointer

}
```

```
harsh dubey ran 42 lines of C

Enter number of unisgned int:
3

Enter the values:
1
0
0

The new values:
1
0
0

The new values:
1
```

**Example:** Dynamically allocated 2D array and Dynamically allocated Stack.

i. **Dynamically allocated 2D array:**

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

#define rows 3
#define cols 2

int main() {
  //dynamically allocating 2D array


   char(*arr)[cols] = ( char(*)
[cols])malloc(sizeof(char)*(rows*cols));

  printf("Enter values: \n");

  for(int i = 0; i < rows; i++)
  {
    for(int j = 0; j < cols; j++)
    {
      scanf("%hhd", &arr[i][j]);
    }
  }

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

```
harsh dubey ran 40 lines of C

Enter values:
256
255
254
253
252
251
Dynamically allocated:
0 -1 -2 -3 -4 -5
```

ii. **Dynamically allocated Stack:**

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

int*stack = NULL;
int count = 0;

int pop()
{

  if(count == 0 )
  {
    printf("Stack is empty\n");
    return -1;
  }

  int pop_val;
  pop_val = stack[--count];

  stack = realloc(stack, sizeof(int)*(count));

  return pop_val;
}


void push(int value)
{
  //if this is the first push
  if(stack == NULL)
  {
    stack = (int*)malloc(sizeof(int));
    stack[count++] = value;
    return;
  }

  stack = realloc(stack, sizeof(int)*(count+1));
  stack[count++] = value;

  return;
}
```

```
harsh dubey ran 66

2
1
Stack is empty
-1
20
10
Stack is empty
-1
```

## 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.**

```c
//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.**

```c
//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:**

```c
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.**

```c
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.

```c
typedef struct point
{
  int x;
  double y;
  char c;
} point;
```

ii. **Basically, we can rename any other datatype.**

```c
typedef  int  harsh;

int main()
{
  harsh x = 10;

printf("int: %d",x);

}
```

```
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:

**Singly Linked List:**

i. **Step 1: Create:**

```
typedef struct node{
  int x;
  struct node*next;
}node;

void insert_end(node**root, int value)
{
  node*new = malloc(sizeof(node));
  new->x = value;
  new->next = NULL;

  if(*root == NULL)
  {
    *root = new;
    return;
  }

}

int main()
{
  node*root = NULL;

  create(&root, 6);

  printf("%d", root->x);

}
```

ii. **Step 2: Add element to the end of Link List:**

```
void insert_end(node**root, int value)
{
  node*new = malloc(sizeof(node));
  new->x = value;
  new->next = NULL;

  if(*root == NULL)
  {
    *root = new;
    return;
  }

  node*curr = *root;

  while(curr->next !=  NULL)
  {
    curr = curr->next;
  }
  curr->next = new;

}
```

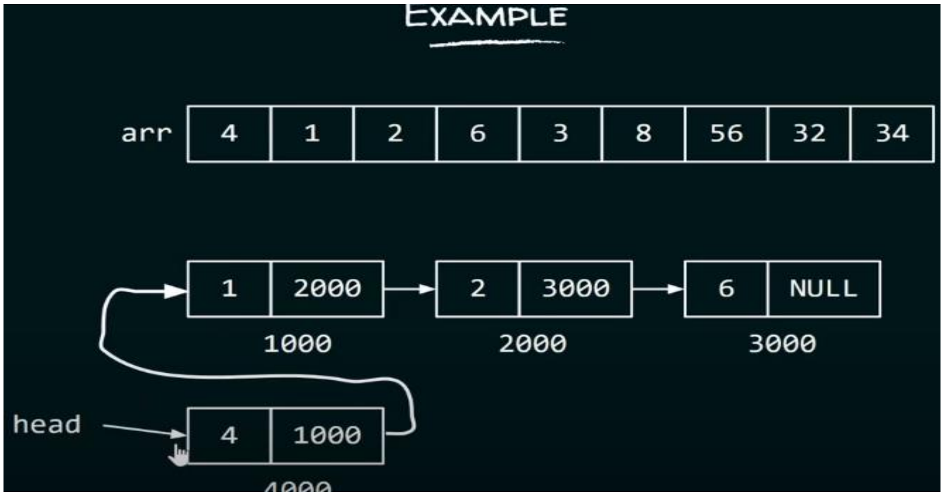
iii. **Step 3: Traverse the list:**

```
int main()
{
  node*root = NULL;

  for(int i = 0; i< 3; i++)
  insert_end(&root,i );

  node*curr = root;

  while(curr != NULL)
  {
  printf("%d\n", curr->x);
  curr = curr->next;
  }


}
```

iv. **Step 4: Add element in the beginning of the list:**

# Asymptotic Analysis:

**Efficiency:** How **efficient** is our **data structure** and how efficient is our **algorithm**. Efficiency is measured in two ways, **number one in time** and **number two in space. An ideal data structure** should take **least amount** of **space** and an **ideal algorithm** should take **least amount** of **time**.




For example, **inserting an element** in **the beginning of a linked list is way faster** as compared to inserting an element in the **beginning of an array.**

**Basically, what we did in the above example, is** compared the run-time for both the data structures.

**Question:** What does the run-time depends on? Size of the **input.** Since, imagine inserting an element in the **beginning of an array of size 100**. That be 100 units of time, for 10,000 size of an array that be 10,000 units of time.

**Time complexity function f(n):** we know that the run-time is dependent on the size of the input. Hence **f(n) is a function of time complexity** of a given data structure or algorithm and is dependent on **n** i.e. number of inputs.

Therefore, if the size of the input is n, then f(n) is a function of n denotes the time complexity.

In other words, f(n) represents the number of instructions executed for the input value n.

For example:

```
for(i=0; i<n; i++) {
    printf("Hello World!");    <---- Assuming, this instruction will take 1 unit of time.
}
```

printf instruction is executed n number of times.

Therefore, f(n) = n

**Asymptotic complexity f(n):** Complexity is input dependent; some data structures perform better with less input, and some perform better with high inputs. **Asymptotic complexity is** approximate time complexity of any algorithm or data structure.

$$f(n) = 5n^2 + 6n + 12$$

| n | $5n^2$ | 6n | 12 |
|---|---|---|---|
| 1 | 21.74% | 26.09% | 52.17% |
| 10 | 87.41% | 10.49% | 2.09% |
| 100 | 98.79% | 1.19% | 0.02% |
| 1000 | 99.88% | 0.12% | 0.0002% |

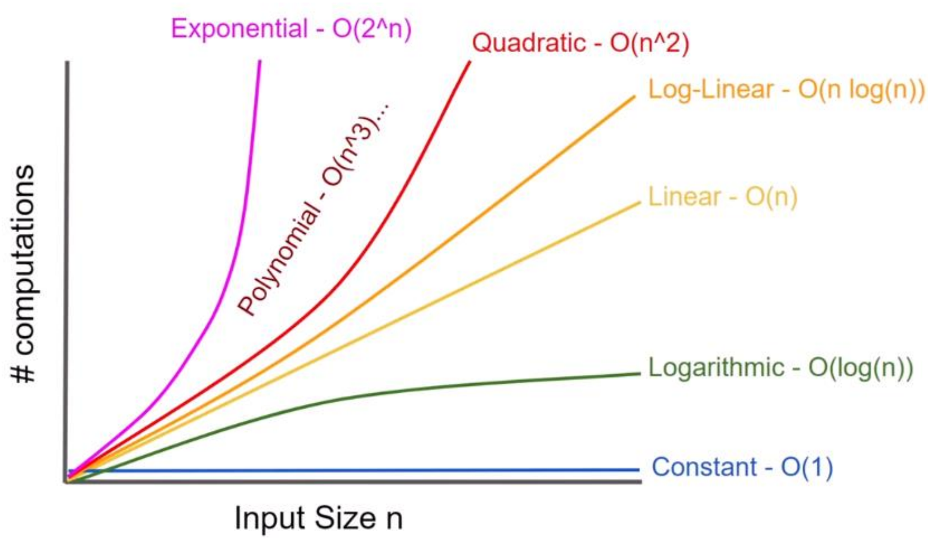
As we can see, 5n^2 is less dominant for less input but for lot of inputs it takes a lot of time. Hence, we can ignore other components and say the approximate time complexity of this algorithm is 5n^2. And this is called asymptotic complexity.

**Time complexity:** Number of operations performed on a particular data structure. Example, adding an element in the beginning of the list.

Now we know that we can compare data structures by looking at the time complexity of the data structure but how do we find the time complexity?

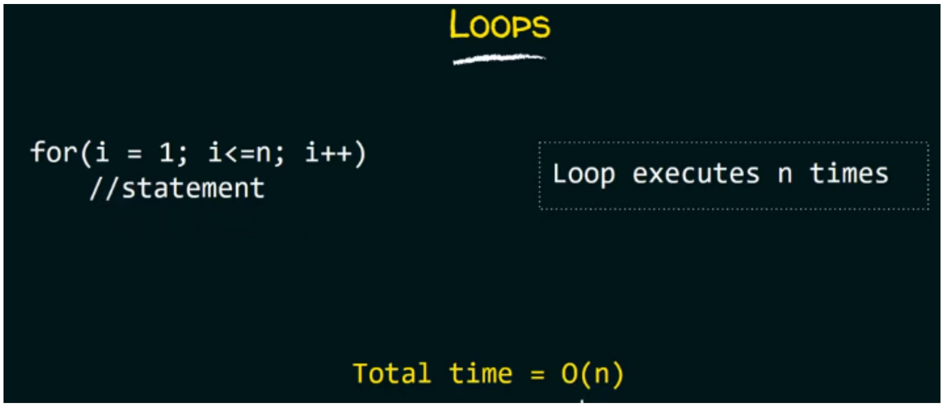
**Big O notation:** Gives you the upper bound of the growth rate of the function. That is, we know time grows with growth in number of inputs.

Different standard functions:

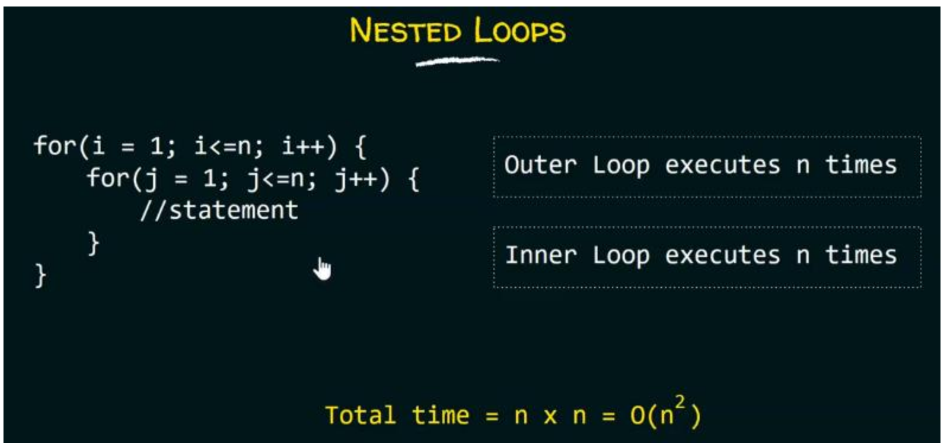


**How to calculate the time complexity of an algorithm:**

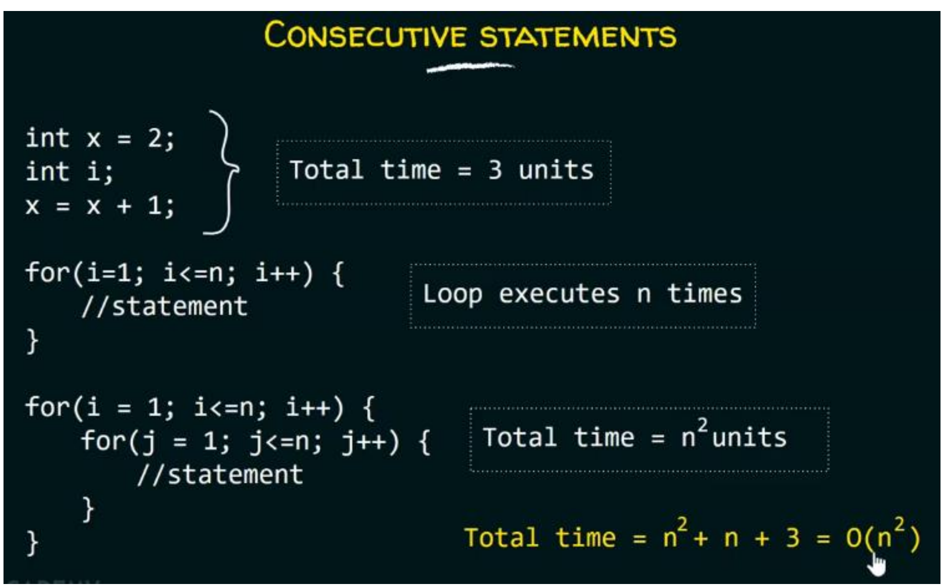
i. **Loops:**



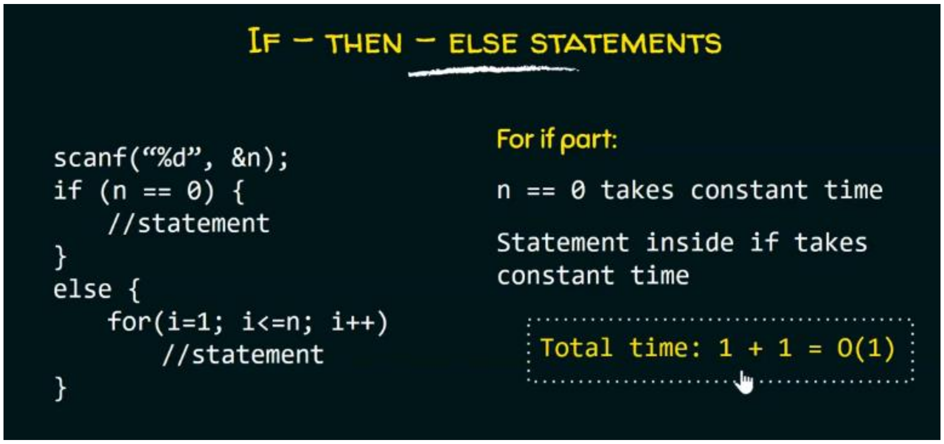

ii. **Nested loops:**

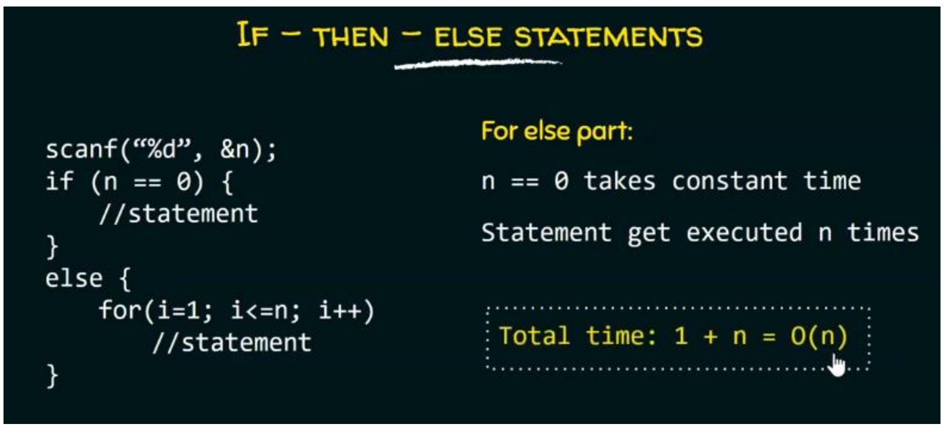



iii. **Consecutive statements:**



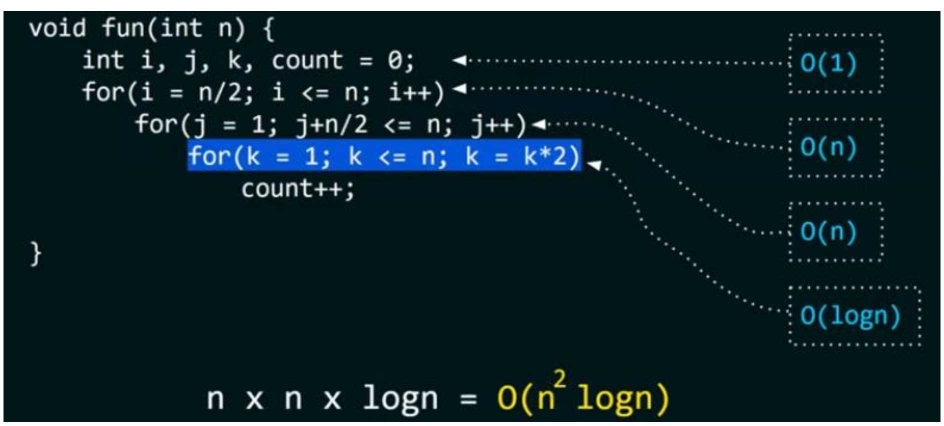
iv. **If and else:**




**Questions:**

**1.**




**2.**

```c
void fun(int n) {
    if(n <= 1) return;
    int i, j;
    for(i=1; i<=n; i++)
        for(j=1; j<=n; j++) {
            printf("Hello\n");
            break;
        }
}
```

**Answer: O(n), due to break statement**

```c
void fun(int n) {
    int i = 1;
    while(i<n) {
        int j = n;
        while(j>0)
            j = j/2;
        i = 2*i;
    }
}
```

**Answer: log^2x**

```c
void fun(int n) {
    int i, j;
    for(i=1; i<=n/3; i++)
        for(j=1; j<=n; j += 4)
            printf("Hello\n");
}
```

**Answer: O(n^2)**

| Data Structures | Algorithms | Concepts |
|---|---|---|
| Linked Lists | Breadth First Search | Bit Manipulation |
| Binary Trees | Depth First Search | Singleton Design Pattern |
| Tries | Binary Search | Factory Design Pattern |
| Stacks | Merge Sort | Memory (Stack vs Heap) |
| Queues | Quick Sort | Recursion |
| Vectors / ArrayLists | Tree Insert / Find / etc | Big-O Time |
| Hash Tables | | |

**How to answer a technical question:** QTTEC

**Step 1:** Ask questions to remove ambiguity. (Q)

**Step 2:** Think out loud while designing an algorithm. (T)

**2.1** Example: Calculate the angle between an hour and min. given time.
**2.2** Pattern matching: If this problem can be solved by any other algorithm.
**2.3** Simplify and generalize: A ransom note from a magazine.
**2.4** Base case and build: Design an algorithm to print all permutation of a string.
**2.5** Data structure brainstorm:

**Step 3:** Test your code. (T)

**Step 4:** Think about edge cases and install protection. (E)

**Step 5:** Think about space and time complexity. And how could you potentially improve the algorithm. (C)

# 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.

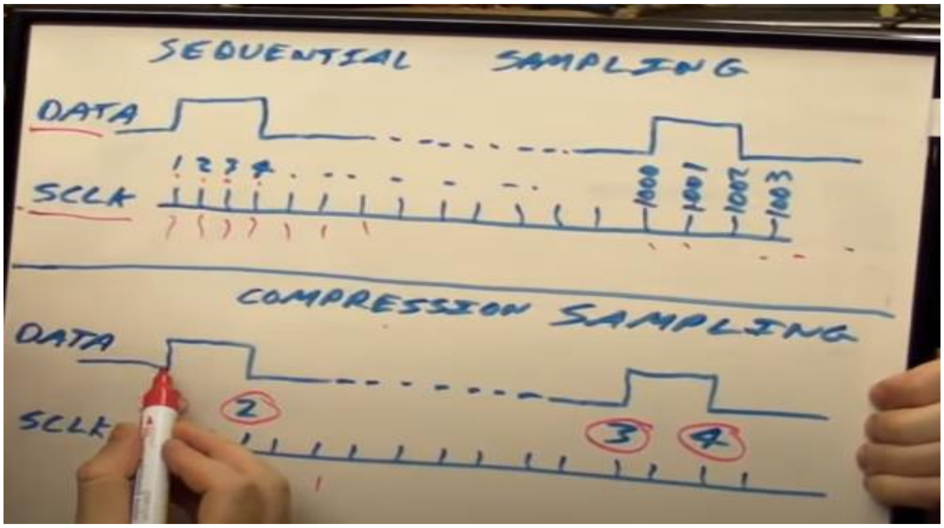
25. **What is Dynamic linking:** You just have the name of the libraries and they are linked when the execution starts.
26. **Wild pointer:** Pointer which is not initialized.
27. **Dandling pointer:** Pointing to object that doesn't exists.
28. **Near pointer:** Bit address up to 16 bits only.
29. **Far 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.

## 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 oscope 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:

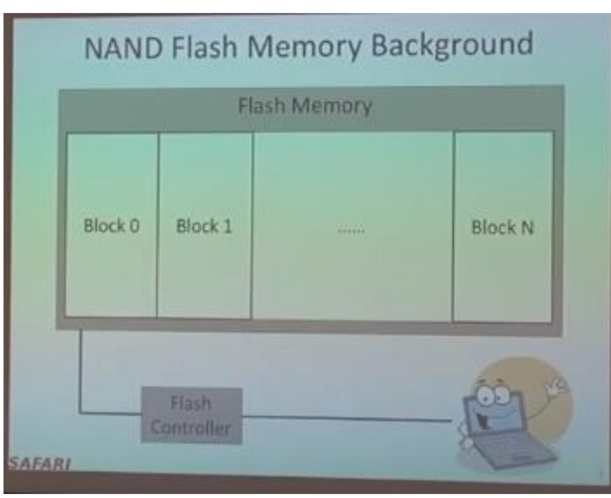
12. Hate logic analyzer: Doesn't show what's actually happening.
13. Timing analysis vs State analysis.
14. Timing analysis: Have to provide internal clock.  Asynchronous mode.
15. State analysis: Have to provide external clock. Synchronous mode.
16. Sampling memory: sequential sampling vs compression sampling.

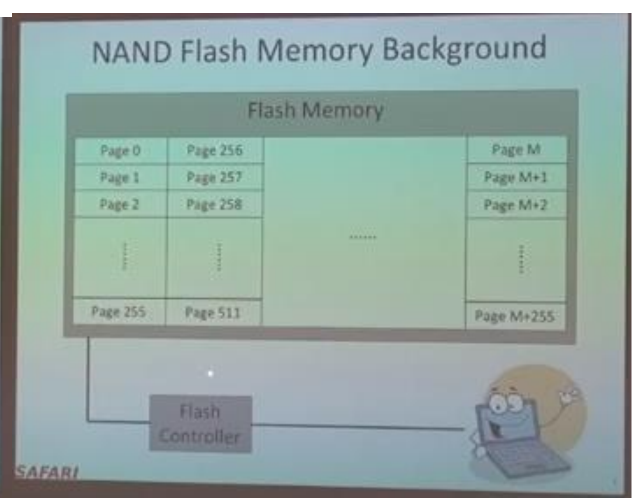
17. 

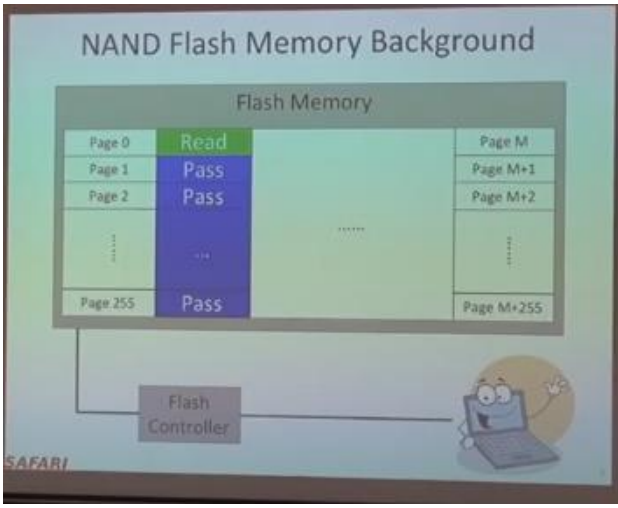

## Take home challenge:

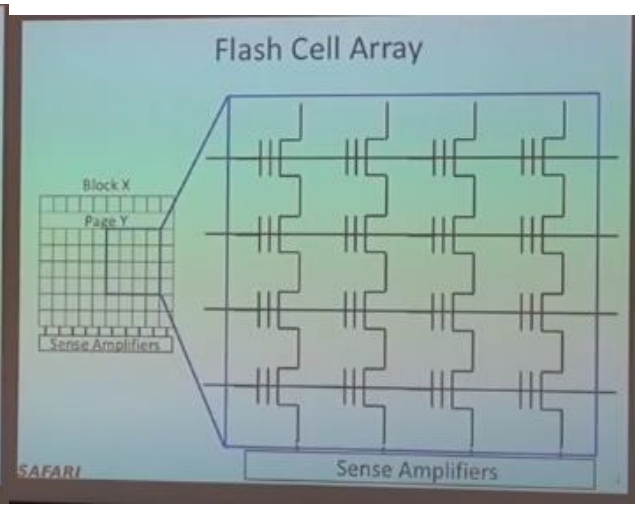
1. **Readme file:**
    i. **Contributors:**
    ii. **Brief description and Introduction:**
    iii. **Planning:** Whatever planning went before the actual code.
    iv. **Technologies used for the code:**
    v. How to do running and debugging.
    vi. Future improvement. What if there was more time?
2. **Unit test and integration test:**

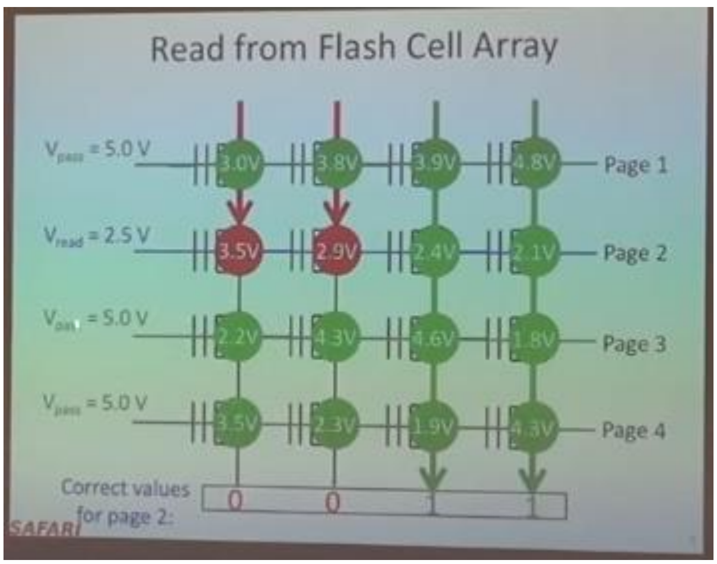
**NAND FLASH Memory:**




NAND Flash Memory Background
Flash Memory
Page 0
Read
Page M
Page 1
Pass
Page M+1
Page 2
Pass
Page M+2
......
Page 255
Pass
Page M+255
Flash Controller
SAFARI


Flash Cell Array
Block X
Page Y
Sense Amplifiers
Sense Amplifiers
SAFARI


Read from Flash Cell Array
$V_{pass}$ = 5.0 V
3.0V
3.8V
3.9V
4.8V
Page 1
$V_{read}$ = 2.5 V
3.5V
2.9V
2.4V
2.1V
Page 2
$V_{pass}$ = 5.0 V
2.2V
4.3V
4.6V
1.8V
Page 3
$V_{pass}$ = 5.0 V
3.5V
2.3V
1.9V
4.3V
Page 4
Correct values for page 2:
0
0
1
1
SAFARI