* for int main () for presence of any int there has to be Return 0; at the end.
* empty parentheses actually makes us understand that there is no value that is returned and it can be explicitly mentioned by main (void).
    **before main the #define directive is announced

***any mentions with #** in the beginning is a **PREPROCESSOR COMPILER DIRECTIVE**

***stdio.h** is actually standard input output, (they are header files containing library functions)
    *they are called on with #include preprocessor directive
*the ending brace of the main function is the Logical end of a function.
**No semicolon after a statement outside braces.

BASIC STRUCTURE

Documentation section (set of **comments** giving name, author, details)
Link section((instructions for the compiler to add link functions from the library)
Definition section (**defines** all symbolic constants)
Global declaration (variables that need to used in multiple funtions are mentioned here)

main () **function section(**contains two parts declaration and executable part

{
Declaration part
Executable part
}

Tokens

A token is the smallest **individual unit** of coding. In the case of the C language, these are called C tokens. (Ref: Page 25 – ANSCII Book)

C Tokens:

1. Keywords – words that are fixed for C and cannot be used by the user (e.g. float, int, while etc.)

2. Identifiers – any words that are used in the program (except keywords)

3. Constants – numbers used in the program (can be decimals)

4. Strings – characters used in the statement (e.g. “cat”, “dog”)

5. Special Symbols – symbols used for specific purposes in coding (e.g. (), {}, ; etc.)

Operators – symbols of addition, subtraction etc.

Data Types

int	– integers	– 16 bit/32 bit size	(4 bytes)
char	– characters	– 8 bit	(1 byte)
long int	– long integers	– 32 bit size always	(4 bytes)
long long int		– 64 bit size always	(8 bytes)
float	– decimal numbers	– 32 bit size	(4 bytes)
double		– 64 bit	(8 bytes)
long double		– 80 bit	(10 bytes)

Computers store the values in binary 0s and 1s. For 9 bits, the first bit is used for the sign and the other 7 bits are used to represent binary numbers from 0 to $2^7 - 1$.

Rules of Identifiers

1) It can only contain alphabets, digits and underscores

2) It cannot **start with a digit**

3) It cannot have **spaces**

4) Cannot use keywords

For constants,

Integer constants: the largest integer saved by a *16 bit machine is 32767

*32 bit machine is 2147483647

* float numbers are also called real constants.(2.165e^2 means 216.5) exponents cant be decimal.

Formatted input

%c----------charactertype (special char)Backslash characters(control character)

%d----------decimal integer type \a---------audible alert

%f----------float form \\----------

display one backslash

%s ----------string type \'----------display one

quotation

%e----------exponential form \"---------- for double

quotation

%lf----------Long float type \O----------null

%ld----------Long double or integer type \n----------new line

%i----------intger type \?----------

display ? mark

READING OF A CHARACTER:

char name;

putchar(var); [simply prints a character]

name=getchar(); [takes input]

* Here, a character is read, **only a single character** is taken as input.

* It is used on the right side to the variable name.

Identifier of Character & digit: [Comparision of character included]

isalpha()

isdigit()

*these give a Non zero value only if the **character input** is **given for alpha**, a character and **for digit**, **a number**.

*Again these are **Zero** only if **character input** is not **a digit** or **number** respectively.

islower() Lower case letter identifier .

isupper() Upper or Not.

FORMAT USAGE

%-x.y [here **- sign** indicates that it will be in **left alignment** and space will stay on right]

%+x.y[here **+sign** indicates that it will have **right alignment** and space on left]

For %d

("%5de",3) ----------> 3 e [Here **5 acts as the maximum space** that can be taken by the digit. **If more is tried to be printed only the first 5 digits will be taken**.]

("%**4**.**3**d",56)----------> 056 [here **4** acts as a space specifier which gives us the idea of how much space there should be infront of the number (extra space). **3** specifies the maximum number of digits that should be printed. [4 spaces are here where 3 are occupied by the number 2 for 56 and 1 for the 0 and the last space is just a space]

It is a field width specifier

For %f

("%4.3",5.6767677676)----------> 5.676 [The number after decimal actually restricts the total number of decimal place there should be in a number]

*here decimal point (.) also is a space taker.

*IF WE WRITE 08 it will be counted as invalid as the number is acutally taken as an octal value here. and 8 and 9 are invalid octals.

For %*s

printf("%*s\n", i, "ARKAM");----------> This will have **"i" space** here before ARKAM string. **"*"** is the unknown value the placeholder of which is i. In this way variable space can be anounced for looping purpose and more.

For %*d

The number will be skipped but a input will be taken which will have no use whatsover.

int random = rand() % 100;

For printing of random number whose last 2 digits are taken by modulus of the number.

Operators

[1] Relational

== ----------> will check if operands are equal [result=true if equal}

!= ----------> will check if operand are not equal [result= true]

> ----------> will check if left hand side is bigger than left hand side

<----------> will check if rhs bigger than lhs

>= ----------> will check if result bigger or equal

[2]Logical

&&----------> AND

||----------> OR

!----------->NOT

[3]Arithmatic operator

There are two types of arithmatic operators. They are

i)Unary operators: Used before the a variable.

UPLUS (+) and UMINUS (-) (both indicates that the numbers are positive or negative.)

ii)Binary operators: (*, +, -) *% gives us the reminder value after division of a number by another number.

USAGE:

- The mixing of variable types during operation is also allowed.

  int op int → int (5 + 10 → 15)

  int op float → float (5 + 10.0 → 15.0)

  float op float → float (5.0 + 10.0 → 15.0)

- Remainder operator takes the symbol of left operand.

  $-9\%7 = -2$

  $9\% - 7 = 2$

  $-9\% - 7 = -2$

[4]Assignment operators:

*= stores the value in an overall expression.
*all assignment operators are actually **Right associative**.
*They have the **least precedence**.
*Most operators do not modify the value of its operand(s) and produces a new result. However, some operators including simple assignment (=) do modify the value of left operand. Such behavior of operators is called Side Effect.

usage:

1)
```
int i;
float j;
i=72.99 //i=72 as its a variable
j=5          //j=5.00 as its a float value
```

2) The following example shows how the float value is first converted to int to assign to an int and then that int value is converted back to float value, losing the fractional part of the value in process.

```
        int i;
        float f;
        f = i = 72.99;
 /* i = 72, f = 72.0 not 72.99*/
```

3) int i=1;
i=i+2; [This can be simplified to i += 2; // i = 3]

This is known as **compound assignment**.

List of common arithmetic compounnd assignment operators:

```
v += e → v = v + e
v -= e → v = v - e
v *= e → v = v * e
v /= e → v = v / e
v %= e → v = v % e
```

[5]INCREMENT AND DECREMENT OPERATORS

*There are two versions which are Postfix and Prefix.

*i++; → i = i + 1; // (i += 1)

i--; → i = i - 1; // (i -= 1)

*As it is left associative, the Operand is sequenced from the left side.

*Postfix: Here, at first the variable is seen for an opearation and later after that the value is added.

```
int i=4;
printf("%d", i++);
\*we will get printed value 4.*\
```

*Prefix: Here, at first the value of variable is added to 1 and then the variable is printed.

```
int i=4
        printf("%d",++i);
        \*we will get printed value 5*\
```

*int a = 0, b = 1, c = 1;

if(a++ && b--)

in this case the value of a is first taken to be 0 and then added later. So the final value of this

process inside if will be 0 as a is evaluated to be 0 at first.**FALSE**

int a = 0, b = 1, c = 1;

if(++a && b--)

In the case of this the ++ part will be pre evaluated as it has been used as prefix and so the final answer will be **TRUE**

[5]Logical operators:

* These types of operators can combine conditions in a condition statement.
* && is AND operator (this has higher precedence) [BINARY}
* || is OR operator (this has lower precedence) [BINARY]
* ! is NOT or logical negation. [UNARY}
* Assignment operation can be done in a conditional statement.
* Precedence is **lower** than Arithmatic operators and relattional operators.
* **left** associative.
* **Here any non zero operators are considered as TRUE**

USAGE:

- TRUE=1 and FALSE=0
- NOT(!) !A-> true if A is false (0)
  !A-->false if A is true (1)
- Always work will be done following precedence so, if a conditional statement is written
  int a=5 , b=10;
  if (1<a<10)
  {printf("hello");
  at first as the associativity is from the left, the red part will be executed first and if

1<a is true 1 will be gotten which will then be compared to 10 i.e 1<10. Hello will be printed so brackets should be used and conditions are to be seperated using relational operators.

Instead we can write if(1<a && a<10)

- In case if a integer or float is dealt with && or ||,
  *For any integer value the value will be taken TRUE (ex: 50&&true(1)==1)

- SHORT CIRCUIT: In some actions, sometimes some **cases are absolute** for which **only one part of the code is executed** and the **rest is not taken into consideration**. Activities are started usually from the left side and so the right is ignored.
  * TRUE(1) is ABSOLUTE in an OR operation.
  * FALSE(0) is ABSOLUTE in an AND operation.

- Relational operators can compare mixed type variables too. (between both float and integers and other sorts)

- Following is a "Correct Statement" in C.

$$i < j < k$$

However, it does not check j to be in between i and k. Rather, First checks whether i < j which returns 1(true) or 0(false) .Then it checks whether the previously returned 1 or 0 is less than k, which again returns 1(true) or 0(false) .

Example:

```
int a=50, b=10;
if(a==50 || a>10 && a>b)
```

Only a==50 will be executed and as found TRUE (1) when OR operation occurs anything added to 1 will be 1. So TRUE IS ABSOLUTE.

int a=50, b=10;

if(**a==40** && a>10 && a>b)

only a==40 will executed first and we will get a false and anything Multiplied to 0 is 0 and so the answer will be 0 in the end. So FALSE IS ABSOLUTE.

int a=50, b=10;

if(**a==50** && a>10 && a>b)

in this case as 1 is not absolute..... eventhough its true on the left side, the operation wil be carried out in the right side too.

int a=50, b=10;

if(**a==40** || a>10 && a>b)

in this case too 0 is not absolute and so the right side will be aken into consideration.

6EQUALITY OPERATORS:

*used to check the equality between two operands.

*Lower precedence than Relational operators.

* "!=" and "==" are respectively not equal and equal.

USAGE:

- Mixed types can be identified at the same time expressions can also be used

  5=6---> false (0)

  2=2---->true (0)

  5!=6---->false (0)

  5!=2=3-->false(0)

- Dont mix up = and == signs because they will shit on you later.

7Comma Operator:

*has the **LOWEST PRECEDENCE** of all the operators.

*It is left associative.

USAGE:

It is evaluated in two steps.

Evaluate expr1 and discard the value

Evaluate expr2 and it becomes the value of the entire equation

This is why expr1 must have side effect, otherwise it is useless.

Example:

$i = 1, j = 2, k = i + j \rightarrow ((i = 1), (j = 2)), (k = (i + j))$

So, the key point is that the values of the first two sub-expressions ($i = 1$ and $j = 2$) are computed but discarded, and only the value of the last expression ($k = i + j$) is kept and returned as the result.

Precedence [ORDER OF EVALUATION OF OPERATORS]

Associativity [OPERAND GROUPING SEQUENCE]

***Operations inside the Brackets are always done first**

*using plenty of brackets (parentheses) maybe be best if you are in doubt about Precedence and associativity

- *Left Associative:When operations Grouped from the Left to right

  +, -, *, / are Left Associative.

- Right Associative: When done from the Right.

  UPLUS and UMINUS are right Asscociative.

  -+t --> -t (actual sense)

Precedence & associativity table

Precedence	Name	Symbol(s)	Associativity
1	increment (postfix)	++	left
	decrement (postfix)	--	
2	increment (prefix)	++	right
	decrement (prefix)	--	
	UPLUS	+	
	UMINUS	-	
3	multiplicative	* / %	left
4	additive	+ -	left
5	assignent	= *= /= %= += -=	right

*There might be problematic situations where the computer might not be able to understand which is to be implemented first. Here the computer will just take the route which suits it more in terms of memory reachability.

```
int i=10,
i=(i*2)+(i/2);
```

Here, it is impopssible to tell which one will be implemented by the computer.

*Sometimes the result might also be **changed**.

In most cases, this evaluation order does not matter, but it might change results in case of operators with side effects.

```
int i = 2;
int j;
```

j = i * i++; → j = (i) * (i++);

In the code above, value of j can either be 4 or 6 depending on the implementation of the compiler, so it is an **undefined behavior.**

SELECTION STATEMENT

There are mainly 2 types of selection statements:

i) "if" statement

ii)"switch" statement

iii)temary/conditional operator(?:)

[1]IF STATEMENT:

*Can execute more than one statement which should end with ";"

*Compound statements can be used in loops and othe places too.

USAGE:

- if (expression) statement {Here if the EXPRESSION in the statement not 0 then the next statement will be executed.

- NESTED IF statements are If statements which can be used

```
if (i > j){
        {if (i > k)
        max = i;
        else
        max = k;}
else if (j > k)
max = j;
else
max = k;}
```

- **Cascaded "if" statement:**

Best way to test a series of condition is to use cascaded if statements, meaning another if statement will be nested inside first if statement's else clause and so on.

```
if (n < 0)
printf("n is negative\n"); else
if (n == 0)
printf("n is zero\n"); else
printf("n is positive\n");
```

*Again, it can be nested like if and then else if and the else and so on

[2]SWITCH STATEMENT:

*it can **only check EQUALITY.**

*not more than **one condition** can be added to this statement.

*around the blocks of action no braces have to be used.

*the name of the case itself is known as **case label** usually used as integers.

*CASE LABELS END WITH A **COLON(:)**

*Break statement has to be added as required after every cases if multiple cases are not to be runned simultaneously as all of it will run if "break;" statement is not used.

*DEFAULT case has to be added if the None of the case values actually match.

- **Controlling Expression "**switch(controlling statement)**"**

  *Must be an integer/enumerable expression

  *Characters are treated as integers in C (Can be used)

  *Floating point numbers and strings are not allowed

- **Constant Expression** "case controlling statement: statements"

*Can be evaluated at compile time.

*Similar to ordinary expression but cannot contain variables or function calls

Duplicate case labels are not allowed

*The order of cases does not matter

*Use of constants and proprocessor constants are fine.

--->5 or 5 + 10 is allowed

----->n + 10 is not allowed (given n is a variable)

- **Statement**

  *Any number of statements are allowed

  *No braces required around statement

  *The last statement is usually a break.

USAGE:

- Several case labels may precede same group of statements.

```
switch (grade) {
case 4:
case 3:
case 2:
case 1: printf("Passing");
break;
case 0: printf("Failing");
break;
default: printf("Illegal grade");
break;
```

- Several cases labels can be put on the same lines to save space

```
switch (grade) {
case 4: case 3: case 2: case 1:
printf("Passing"); break;
```

```
case 0: printf("Failing");
break;
default: printf("Illegal grade");
break;
```

- If default case is missing and controlling expression is 0, control passes to the next function after switch statement.

[3]Conditional statement:(?:)

* An operation that allows an expression to produce one of the two values depending on the condition.

*Operands can be of any type and it requires 3 operands, hence it is a ternary operator.

USAGE:

i)expr1 ? expr2 : expr3

If expr1 is 1(true) , then expr2 is evaluated, otherwise expr3 is evaluated and the evaluated value becomes the value of the entire expression.

ii) It can exchange if and else statement in some cases. For example:

```
if(x>0){
        flag=0;
}
```

can be exchanged by

```
flag=(x<0) ? 0 : 1
else{
flag=1;
}
```

Iteration statements

An iteration statement also known as a "Loop" is a statement whose job is to execute some other statement (loop body) based on some "controlling expressions".

*As long as controlling expression is true (1) it runs.

*It breaks when controlling exp. is 0.

TERMINATION OF LOOPS:

**The loop can be ended from anywhere between if we use the break statement.

** An if statement can judge is the loop was ended prematurely.

*break can escape only one level of nesting in nested statements.

*Usage of **continue**

```
int n = 0, sum = 0, i;
while (n < 10) {
scanf("%d", &i);
if (i <= 0)
continue;
sum += i;
n++;
/* continue jumps to here */
}
```

3 types of loops:

1)while: Controlling exp. is tested BEFORE execution of loop.

2)do : Controlling exp. is tested AFTER execution of loop.

3)for : *It has multi purpose. Assignment(1); Condition(2) ;Increment/Decrement(3)

*when loop is deterministic.

1while:

*form: while (expression) statement

*loop continues until cont. exp. is not 0. (non-zero--->execution--->check cont exp---> non zero (redo)/ zero (dead))

USAGE:

- int i=1;
  while(i<n){ //controlling expression
  i=i*2; // loop body

*Can use compound statement.
*Always using braces is not required if there is only one statement.
*Body of while loop will not be chekced if the condition is false to begin with.
*Infinity loop & its solution: Loop should be terminated so that it does not turn into an infinity loop and to terminate the loop inside the loop itself we have to use (break, goto, return) or call a function to terminate the program.

[2]do statement:

*This statement executes the loop body first, then evaluates controlling expression.
*The loop body is executed at least once.
*work sequence:
i)EXECUTE FIRST BITCHHHH----->eh condition not match--->ig we can end here then lol.
ii)EXECUUTEEEEE------->ok condtion working----->LEZGOOO--->until it doesnt give false.

USAGE:
i)Usage format: **do statement while** (expression);
ii)A simple code using do statement to find the number of digits in a number.

```
#include <stdio.h>
main (void)
{
```

```
int n,a,div_count ;
printf("Enter a non negative number:");
scanf("%d",&n);
  div_count=0;
  do{
              n/=10;
              div_count++;
              a=n;
        }while (a>0);
printf("%d", div_count);
}
```

[3]for statement:

*Ideal for looping where there are counting variables (deterministic loop count)

USAGE:

- for(**exp 1**; **exp 2;** **exp3**)
  initialization **Cont. expn** **operation**

- For some rare cases "while" can replace the for statement
  initialization;
  while(**controlling expression**){
  statement;
  operation;}

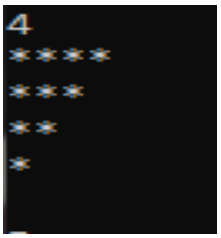
- C99 allows declaration of variables inside the braces where the value is only usable inside the loop. Here more than one variable can be announced with the use of coma operator.

- At the same time more than one initialization or several increments can be made.

Multiples in only **FIRST** and **THIRD** expression.

Nested Looping:

- for obtaining the solution of the star problems:




```
#include <stdio.h>
int main(){
int i, n, j;
scanf("%d",&n);
        for(i=0;i<n;i++){
                for(j=i;j<n;j++)
                        printf("*");
        printf("\n");
        }
}
```

- 


```
#include <stdio.h>
int main(){
int i, n, j;
scanf("%d",&n);
        for(i=n ;i>0;i++){
                for(j=n-i ;j<=n; j++)
```

```
                printf("*");
        printf("\n");
        }
}
```

STRING

*A string is a sequence of characters treated as a single data item.
*end with special symbol null : '\0'
*array size should always be (whatever we need + 1) [+1 for a null character].
**Any group of character defined between double quotation is called string.
---------->for example: "what do you mean"
*if a **double quote** needs to be printed then it needs to
    **be started with \"
    **again in the end \"
printf("\"my nigga\"")-----------------> "my nigga"

*In c string is not acepted as a data type but **allows the string to be a character array**.
---------->for example: char city [9] = "New York"
    **automatically a null terminator is initialized at the end "\0"
    Here the initialization **can not be seperated from the declaration meaning both needs to be done together.
    Also they cannot be used as the left operand of an **"assignment operator".
    ****\0 signifies how far of a string or character array is filled with data. In fact, if a character array has \0, it becomes a string. .

[READING A STRING]

USING SCANF

*The problem with scanf is that it takes the **first** blank space given as the **terminator of the array**.
    For example: The letters typed in the array were "New York". It will **only take New** and then stop afterwards.

Solution:

if this whole line needs to be read there needs to be **two** declaration of arrays and **two** inputs to **two arrays** inform of string. For example: scanf("%s %s", &ar[3], &ar[5]);

*Unlike the other scanf calls, in case of character arrays, the **ampersand is not required**.

for example:
char address[10];
scanf("%s", address);

***READING A LINE OF TEXT:

**********white spaces can also be printed**********

USING SCANF:
*a very rarely used method

```
char line[80];
scanf("%[^\n]", line);
printf("%s", line);
```

USING **GETS**: [RECOMENDED]

```
char line[80];
gets(line);
puts(line);
```

<string.h> functions:

- *strlen(ar);*---------> How many character in a string array.
  *it does not take the null character as a total_space.....
  total_space=[what we take - 1]
- *strcpy(where, what)*--> Overwrites 'what' with 'where' in a string array.
  *overwrites the space of 'where' in 'what' part with a null at end.
  *if ar[2]="si" copied to ar[6]="arki" we will get---> ar[6]='s' 'i' '\0' 'i'
- *strcat(where, what)*---> Adds what after where.
  *adds 'what' after 'where'.
  *only one '\0' character at the end after addition.
- ***strcmp**(with, what)*---> Compares two strings identified by the arguments.
  *when equal (**returns** 0) [The character are same].

  *when different (the difference is returned **according to lexicography**).

for strcmp(str1, str2)

* if returned value **smaller than 0** **str1 comes before str2**
* if returned value **bigger than 0** **str2 comes before str1**

- *strcmpi(ar1, ar2);*--> string compares all and ignores the upper case lower case variation.
- *stricmp(ar1, ar2,1);*------> same as strncmp but with ignorance towards case sensititvity
- *strlwr(ar);*---------> The string of an array will be converted directly to lower case.
- *strupr(ar);*---------->The string of an array will be converted to upper case.
- *strset(ar, 'x');*---------> All the letters of the array will be converted to 'x'.
- *strnset(ar, 'x', 5);*--------> First 5 number of array characters will be converted to 5.
- *Strncpy(with, what, how_much)*--> Copies one strings leftmost num of chars.
  *it is a 3 parameter function invoked as follows: strncpy(s1, s2, 5);
  *It copies one string to the other as much needed from the leftmost side.
- *strncmp(with, what, how_much)*---> A variation of function strcmp.
  *compares a certain number of elements from one place to the other.
- *strncat(with, what, how_much)*---> variation of strcat.
  *adds a certain part of an array to the assigned target as much wanted.
- *strstr(with, what)*---> searches the string to the if the target string is there.
  *The position of the occurance is returned by strstr.
  ***if successful, returns a pointer to the location of the match**
  *If **unsuccessful, a match then it returns NULL pointer.**
- *strchr(inside_array, char)*--> searches if it can find the target character in array.
  *The existence of a character can be identified within a character array.
  *It is invoked as follows. strstr(s1, 'm'); to search m in the array.
- *strrchr(inside_array, char)*--> identifies the first time it sees a character inside

Using width specifier:

*scanf("%**w**s", name); **[hardcoded approach] [it does not take variables]**
here w is the width of the string we want to take input for\

Here two cases might occur:

1. W is equal or greater than the string length--------> we get the **whole** string in string variable.
2. W is smaller than string length----------> the excess letters are **truncated** and **unread**.

itoa()

&

atoi()

* We can convert a **string** to **number & vice versa** with the help of the function.

itoa() converts an **integer to string**.

Purpose: The `itoa()` function converts an integer value to **a null-terminated string representation**

Syntax**: char* itoa(int value, char* str, int base);**

Value: The **integer value** to be **converted to a string**.

str: A **character array** (buffer) where the **resulting string will be stored.**

base : The **numerical base for the conversion** (e.g., **10 for decimal, 16 for hexadecimal, 2 for binary**).

atoi() converts a **string to integer.**

Purpose: The `itoa()` function converts a string to **a integer**

Syntax: **atoi(char *str);**

str: The string to be converted to **an integer**.

return value:

On-success: It returns an **integer value** which needs to be saved in a var.

On-failure: If the string **cannot be converted to an integer, it returns 0.**

atoi() **stops reading the string** at the **first non-numeric character** (e.g., spaces, letters, or symbols).

Example:

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

int main() {

    char str1[] = "12345";
    char str2[] = "1001";
    char str3[] = "Hello";
    char str[] = "123abc";
    // Convert string to integer
    int num1 = atoi(str1);
    int num2 = atoi(str2);
    int num3 = atoi(str3);
  int num4=atoi(str4);
    printf("String to Integer 1: %d\n", num1);
    printf("String to Integer 2: %d\n", num2);
    printf("String to Integer 3: %d\n", num3); // Invalid conversion
    printf("String to integer 4: %d\n", num4); //incomplete conversion
    return 0;
}
```

Output:

String to Integer 1: 12345

String to Integer 2: 1001

String to Integer 3: 0

String to integer 4: 123

ARRAY

*Either character or integer collections can be made with arrays.

*No mixture of different types of data can be done.

*b[i] inside i indicates the array size.

PRE-INITIALIZATION

```
int ar[5];                          /*initialization of memory spaces */
int ar[5]={1, 2, 3, 4, 5}; /* initialization of memory spaces with values*/
int ar[]={1, 3, 4, 5, 2 }; /* fixed 5 memory spaces*/
int ar[5]={0};                      /*all will be initialized to 0*/
int ar[5]={2,3}                     /*all other memory spaces will be initialized to 0*/
int ar[30]={44};                    /*if one array value is initialized then the rest will be zero*/
```

!! int ar[i] cannnot be done as the size cannot be dynamic

2D ARRAY

*Two dimensional arrays can be mentioned like this:

ar[row][column]

*two parentheses are used to define row and column number

two dimensional arrays are sorted in the memory in this way:

row\coloum	[0][0]	[0][1]	[0][2]
	[1][0]	[1][1]	[1][2]

INITIALIZATION:

- int number [] []={(1,2,3), //for rows and coloumns individual assignation required.
  (2,3,4)
  }

- int number[10] [10]; //one on one initialization can also be done.
  number [1] [1]=4;
  number [2] [1]=5;

USAGE:

- to find the row numbers and column numbers of an array. A generalized solution.

  ```
  int rows=sizeof(numbers)/sizeof(numbers[0]);
    //total size divided by size of row
  int column=sizeof(numbers[0])/sizeof(numbers[0][0]);
  //total row size divided by element
  ```

- This code can be used to print the 2D array altogether.

  ```
  for(int i=0; i<rows; i++){
      for(int j=0; j<columns ; j++){
      printf("%d ", j);
  ```

```
}
printf("\n");
}
```

- Whole row can be printed without mentioning of the columns.... in this code we can find the whole array just by printing the row.

```
char cars [][10]={
                    "Mustang";
                    "corvette";
                    "porsche"
}
for(i=0; i<rows ;i++){
        printf("%s\n", ar[i];
}
```

Use of srand() and rand()

*for srand and rand we will need <stdlib.h>

*for using TIME(NULL) we will need <time.h>

*if same seed value used we will get same output.

*if TIME(NULL) used as a seed number it will give us total random numbers.

- An array to print random numbers

```
int main(){
srand(time(NULL));//seed number
for(i=0;i<8;i++){
        ar[i]=rand(); //random value assignation
}
```

- An array to print random number between **n1 [minimum] -> n [maximum]**

```
int main(){
srand(time(NULL));                    //seed number
for(i=0;i<8;i++){
        ar[i]=rand(); //random value assignation
        ar[i]=ar[i]%n2 + n1;
}
```

- character array for random **Alphabet** from A-->Z

```
int main(){
srand(time(NULL));//seed number
for(i=0;i<8;i++){
        num=rand(); //random value assignation
        num%=26;
        ar[i]='A' + num;
}
```

*sequential string readin is only available for character arrray

*if string of size 'n' needs to be saved we will need to assign the string length to (n+1)

*

```
int main(){
```

```
	int i,num;

	char ar[8000];

	scanf("%s", &ar[0]);

	printf("%s", ar);

	return 0;

}
```

*here if we input anything we will get each alphabets or character to be assignemd to the place sequentially.

*when null signal found the printing of an array will stp

*if after initialization new string is inputted then the first values will be replaced by that part again the null part will take the place of the place after the string input

	but the rest of the string will remain.

	and only the inputted string will be printed

	if we try printing values after the terminating loop part then they will be printed

	conclusion: no problem will be found withg printing as long as we dont find terminating character.

	if null not stored at the end it will print until the end with all garbage values

	but for string operation already a null character is already set there.

Swapping of values of a variable

MAIN STRATEGY: To swap values of variables; we will introduce a 3rd variable. To store the value of the previous variable in a different variable so that, that variable can be changed in ease without being lost or overwritten.

- [character transfer]

```
int main(){
        char x= 'X';
        char y='Y';
        char temp;
temp = x;
x=y;
y=temp;
return 0;
}
```

- [string transfer]

```
#include <string,h>
int main(){
        char str1[]="bomboklaat";
        char str2[]="what the sigma";
        char temp[100];
        strcpy(temp, str1);
        strcpy(str1, str2 );
        strcpy(str2, temp);
}
```

Sorting

Sorting is moving the value within an array to actually make sense out of the sequencing. We can get ascending, descending etc. Below we have [might have some errors]

- [bubble sort]

```
void sort(int array[], int size){
        for(int i=0; i<size-1; i++){
        for(int j=0; j<size-i-1; j++){
        if(ar[j]>ar[j+1]){
                int temp= ar[i];
                ar[i]=ar[i+1];
                ar[i+1]=temp;
}
}
}
}
void print(int array[], int size){
        for(int i=0; i<size;i++){
                printf("%d ", array[i]);
}


}

int main(){
        int array[10]={5,6,3,4,9,1,10,2,8,7};
        int size = (sizeof(array)/sizeof(ar[0]));
        sort(array[10], size);
        printf(array[10] , size );

}
```

STRUCT

*__collection__ of __related members__ ("variables").

*they can be __different data types__.

*listed under __one name in a block of memory__.

*__called externally out of the main function__.

MEMORY ALLOCATION:

EX:

```
#include <stdio.h>

struct ExampleStruct {
    int a;      // 4 bytes
    char b;     // 1 byte
    float c;    // 4 bytes
};
int main() {
    struct ExampleStruct s;
    printf("Size of struct: %lu bytes\n", sizeof(s)); // Output: Likely 12 bytes (4 + 1 + 4 +padding);
    return 0;
}
```

// int a gets 4 bytes

// char b gets 1 bytes

// float c gets 4 bytes

//total = 9 bytes + padding (likely 12, padding **to align the structure**)

USAGE

- # Initialization

1)**Singular initialization**

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

struct Player{
        int score;
        char name[12];
};
main(){
        struct Player player1;
        struct Player player2;
        strcpy(player1.name ,    "bob");
          player1.score=5;
        strcpy(player2.name ,"marley" );
        player2.score=6;
//we can assign the value of the required part by mentioning it and inputting it in //format
        printf("%s ", player1.name);
//we access a struct value by name_of_it . score/ name
        printf("%d\n", player1.score);
        printf("%s ", player2.name);
        printf("%d\n", player2.score);
}
```

- ## Mutiple Initialization

This can be done directly after initialization not after calling but direct initialization immediately after calling it

```
struct User
```

```
{
        char name[12];
        char pass[25];
        int id;
}
main(){
        struct User user1= { "baker bhai", "bomboklaat", 230041157};
        struct User user2={ "string", "string", int    };
        struct User s1 = { .name = "Alice", .roll = 102, .marks = 91.0 };
}
```

- **ARRAY** with **STRUCT**

```
#include <stdio.h>
#include<string.h>
struct User{
        char name[25];
        int cg;
};
main(){
        struct User user1={"lulu", 4};
        struct User user2={"chomu", 3};
        struct User user3={"lullu", 2};
        struct User user4={"chullui", 5};
        struct User user5={"sallu", 6};
        struct User user[]={user1,user2,user3,user4,user5};
        for(int i=0; i< 6;i++){
                printf("%-12s\t", user[i].name);
                printf("%d\n",user[i].cg);
                }
```

- We can Initialize **Altogether without individual initialization**.

```
struct Student {
        char name[50];
        int roll;
        float marks;
};
struct Student students[3] = {
            {"Alice", 101, 95.5},
    {"Bob", 102, 89.0},
            {"Charlie", 103, 92.3}
        };
```

- **structs with pointers**

```
#include<stdio.h>

struct student{
            char name[50];
            int roll;
            float marks;
}
int main(){
            struct Student s1 = {"David" , 20, 50.57};
            struct Student *p = &s1;                    // pointer p points to the s1 struct

            printf("Name: %s", p->name);
            printf("Roll: %d", p->roll);
            printf("marks: %f", p->marks);
}
```

// p->name stands equivalent to (*p).name

- **nested Struct**

```
#include<stdio.h>
struct address{
        int house_no;
        int road_no;
        char elaka[20];
};
struct name{
        char name[20];
        struct address address;          // we named the address struct as address
};

int main(){
        struct name Lallu;
        Lallu = { "Lallu" , { 23 , 5 , "Nikunja-2"} };
        printf("Name: %s", Lallu.name);
        printf("House no.: %d", Lallu.address.house_no);
        printf("Road no: %d", Lallu.address.road_no);
        printf("Elaka name: %s", Lallu.addreess.elaka);
}
```

Typedef

*It is a reserved keyword which gives an existing data type a nickname

- ## Basic Typedef use

```
//if we use typedef to give this struct a name, then we wont need to call for struct
//again.



typedef struct
{
        char name[12];
        char pass[25];
        int id;
} User; // We give it a nickname user. "User" will be the new data type
main(){
}
typedef int mew;          //we wil name the int format as mew


mew main(){
        struct User user1= { "baker bhai", "bomboklaat", 230041157};
        struct User user2={ "string", "string", int    }; //we dont need to bring struct
}
```

- ## Typedef with pointer

```
typedef struct User* Studentptr //naming the student pointer as Studentptr
typedef struct User
```

```
{
        char name[12];
        char pass[25];
        int id;
} User;
int main(){
        User s1 = {"Milu", "siuuu", 230041232};
        Studentptr ptr = &s1;      //taking a student pointer named as ptr
        printf("%s", ptr->name);
}
```

Binary Number

* A binary number is a number expressed in the base-2 numeral system — a positional notation with a radix of 2.
* Only symbols are 0 & 1.

Unsigned integers

* All bits are considered **data bits**.
* 10101010--------> 2^7+2^5+2^3+2^1 = 170

Signed integer

* The MSB or the most significant bit is the **sign bit**.
* All rest are **data bits**.

EX: 10101010--------> -2^7 +2^5 +2^3 + 2^1 = -86 [signed bit means **negative**]

Bitwise operators

AND & **OR |** **^ XOR**
<< rightshift **>> left shift**

WORKING MECHANISM

interaction of numbers will be done from binary bit to binary bit by converting the decimal to binary.

Then afterwards the number will be bitwisely operated and then we will get the final value in binary.

Then it will be converted to **decimal** again as the result.

bitwise shift << / >>

actually shifts a number from the one side the other by the amount that is mentioned .

>> will move it to the left side by one bit.
<<will move it to the right side by one bit.

FOR EXAMPLE: z=x<<2;
the bits will be shifted to the left by 2.
00001100----------> 00110000

** extra 0s will be added front or back
** The Decimal value increases by the **power of shift with the base n.**
**** IF Left, it multiplies by 2^(1,2,3,4,5.....).**
**** IF Right, it divides by 2^(1,2,3,4,5.....).**

XOR ^

* **If one of the number is 1 and the other is 0 , only then we will get 1.**
* **If both of them are 1 or 0, the output will be 0.**

Memory

memory = an array of bytes within RAM (street).
memory block = a single unit (byte) within memory, used to hold some value.(person)
memory address= the address of where a memory block is located. (house address)

%p to find the **memory addresses** which is taken by a variable and also the & should be taken

before the variable we will need.

- the addresses are found in hexadecimal.
  printf("%p\n", &a);
- With this the size of the varibale can be found in bytes.
  printf("%d", sizeof(a));

Pointers

* a "variable-like" reference that holds a memory address to another variable.
* * = indirection operator (value at address)
* Data type of a pointer needs to be consistent with what we want to save there.

USAGE:

- **A variable can be used to store the address of a variable**

  int age=21;

  int ***pAge**=&age; //here * is the operator and the variable where we want to store
  // the value should be named with *p as prefix as convention
  //the first letter of the variable should be Capital like: Age.

  printf("Address of age= %p", &age);

  printf("value of pAge=%p", pAge);

  printf("value stored at address: %d", *pAge);

  //dereferencing in order to find the value at the address

- It is good practice to assign NULL if declaring a pointer:

  int age= 21;

  int *pAge=NULL;

  pAge=&age;

- INITIALIZATION & ACCESSING VALUE :

  int x, *p , y; **# right: int x, *p=&x , y;**

  x = 2; **# wrong: int *p=x , x , y;**

  p = &x;

  y= *p ; // value of y will be set to value of the location pointed that is x

  *p = 20; //address location value can be accessed and changed

  We can store the address of a value to a variable

  quantity = 20

  p=&quantity

  n=*p // here n=20

  All this is equivalent to:

  n = *&quantity [it points to the value of the address]

  n= quantity // same line as the last one

- We can store ONLY SAME format address to SAME format POINTER

  int x,*p;

  x=20;

  p= &x; **// ok**

  BUT,

  int *p;

  float x=20;

  p = & x; //wrong

- ## CHAIN OF POINTERS:

  Multiple indirections: One pointer contains the address of another pointer which points to a certain value.

  *p1 is just a pointer pointing to a certain value.

  **p2 is a pointer pointing to the address of another pointer. [if p2 points to an integer address holder then p2 is integer pointer.]

USAGE:

```
main(){
int x, *p1 , **p2;
p1 = &x;          /address of x stored in p1/
p2 = &p1      /address of p1 stored in p2/
}
```

****p2 == **(address p1)**

== *(&x)

== value of x

- ## Base Address of an Array

An array name actually **represents the starting address** of an array.

printf("%p %p", ar ,&ar[0]) // we get the same output here which is the first element

- ## Pointer Arithmatics with Arrays

In main function:

* **Incrementing Ar** moves the **pointer forward by** the **entire length of array.**
    * **if int moves by 4**
    * **if double moves by 8**
* **Incrementing Ar[0]** moves the pointer forward by **a single element.**

In self defined function:

* if an increment is performed inside a function where the **array is passed** as an **argument**, only the **pointer itself is incremented**, not the entire array.

- **POINTER EXPRESSIONS:**

I) We can do operations with pointer values. **Like:**

y = *p1 * *p2 // p1 multiplies to p2

sum = sum + *p1

*p2 + = 10 // here this is the increment of a pointer variable

II) If p1 and p2 are both pointers to an array, then p1-p2 returns the number of elements between them.

III) Comparision between pointers of the same data type should be made else error will be made.

IF p1 & p2 are two pointers, then when compared we compare the addressses.

p1 > p2 ----------> let p1(0x007) & p2(0x008) returns false (0)

p1++----------> The memory p1 is moved to the next memory cell. ar[0]--->ar[1]

IV) Arithmatic operations for pointer addresses can not be done like: multiplication, or additionn, substraction of two pointers etc.

- ## Pointers & array

Understanding 1D ARRAY:

We can store the array location to a variable.

for example:

int x[100];

printf("%p", x); // this gives us the location of the first array cell

which can be equivalent to:

p=x; ----------> p= &x[0];

whereas,

p+1= &x[1]

p+2 = &x[2] etc

Value can be obtained using : *(p+i) [i==index] // The i^{th} index value of the array

Understanding 2D ARRAY:

* IF singly used in the main function, both pointer to address and the 2D array work similarly

CODE [**with 2D array**]:

```
char ar[3][10] = {"Hello", "World", "C"};
for (int i = 0; i < 3; i++)
        printf("%s\n", ar[i]);
```

CODE [**with Pointer to Strings**]:

Hardcoded initialization:

```
char *p[] = {"Hello", "World", "C"};
for (int i = 0; i < 3; i++)
        printf("%s\n", p[i]);
```

Initialization with variables is also possible:

```
char str1= "hello";
char str2="mello";
char *p[]= {str1, str2};
```

- **pass through a function:**

```
void printString( char * p[] ,    int size ){
        for(int i=0; i<size; i++){
                printf("%s\n", p[i] );          // array holds base address of the
        }

}
```

- **Sorting is also done with it**

```
// Swap first and second strings

char *temp = p[0];          //holds the address of the first word
p[0] = p[1];
p[1] = temp;
```

```
printf("%s %s %s\n", p[0], p[1], p[2]);    // Output: World Hello C
```

- **Dynamic Allocation : [non-contiguous memory blocks]**

```
p[2] = malloc(10);                          //auto creation  of the pointer array.
strcpy(p[2], "Dynamic");
printf("%s\n", p[2]);    // Output: Dynamic
free(p[2]);    // Don't forget to free!
```

ADRESS PLAYOFF

* If we pass in an Array to a function, then we are actually passing in only **the first array** value [**ar[0]**] to the function only and then, the next is processed.

*If array itself is **Incremented:**

We actually move forward in the array with length of array .

int ar[12];

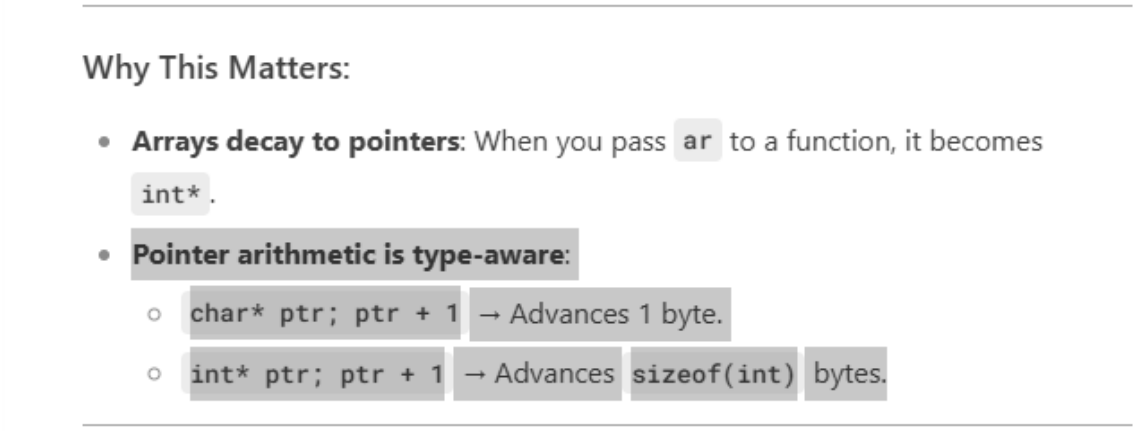
ar+1 = ar[13] // *(ar+ sizeof(int)) **moves onto the next integer**

* But if only ar[n] is **incremented:**

We only move the array by 1 element

* If incremented within a **self defined function:**

We move still only by 1.



* To find the **total size of array**:

We can use strlen for string array but for a string array. But if we wamt to find the total size of the array let us say an **int array:**

int ar[4];

sizeof(ar); // This returns **16** as **every integer will take 4 bytes.**

So to find the actual array size we will need to divide it by **size of int :**

size = sizeof(ar) / sizeof(int); // we get the size of array.

* Initialization skip off:

If we initialize sometimes, the dimension does not need to be mentioned. The dimension are set up automatically:

char ar[][] = { " Hello " , "enshhs" }

* Double Dimension array with one Dimensional array : **[1]**

void f1 (char **p){} // The address pointing to the value of p is pointed to which again shows the value pointing to the address of address within p.

char *p[3] = { "abc", "cuz", "done"};

printf("%s",p[0]); // we get abc

```
f2( &p[0] );                    //we pass in the first array address to functions.

// Here the strings are saved in different places of the memory. But this array holds
      the    FIRST ADDRESS of EACH string which are located scatteredly.
```

Things to notice:

i) p[0] holds an address (ad1) which points to those strings.[char *]
ii) &p[0] holds the address of ad1 address [char **]
iii) *&p[0] simplifies to p[0]

p[0] ------> first address of the string
&p[0]-----> holds the address where it saved the address as the pointer it self has addresses

* Double Dimension array with one Dimensional array : **[2]**

```
void f1 (char **p){
        int i;
        for(int i=0;i<3;i++){
                printf("%s stored at %p", p[i], p[i] );
        }
}
int main(){
        char *p[3]= {"abc", "cuz", "sdkl"};
        f1(&p[0]);
        return 0;
}
```

Parameters

&

Arguments

In programming, functions **can receive data** through **parameters**, which are referred to as **arguments**

when passed into the **function**.

There are two primary ways to pass arguments: **Call by Value** and **Call by Reference**.

Call by Value:

*** In call by value, only the value of a variable is passed to the function.**

*****The function operates on a **copy of the variable**.

* **changes made inside** the function **do not affect the original** variable in the main function.

For example:

```
void function(int x) {
    x = 10; // Changes only the local copy of 'x'
}
int main() {
    int num = 5;
    function(num);
    // 'num' remains 5 in the main function
}
```

Call by Reference:

* In call by reference, the function **receives the address of the variable** rather than a copy.

* **MODIFICATIONS CAN BE MADE** as **THE ACTUAL MEMORY LOCATION** is worked with.

For example:

```
void function(int *p) {
   *p = 10; // Modifies the original variable
}
int main() {
    int num = 5;
    function(&num);
    // 'num' is now 10 in the main function
}
```

SCOPE OF VARIABLES

Local Variables:

***A local variable is **declared inside a function** and **is only accessible within that function**.

*Local variables of the **same name as a global variable** override (ignores) the global variable within their scope.

*Local variables cannot directly modify other local variables outside their function.

Global Variables:

*A global variable is declared outside all functions and can be accessed anywhere in the program.

* If a global variable is declared but **not initialized**, it **defaults to 0**.

* If **no local variable** of the **same name** **exists**, the **global variable** is used by **default**.

*If **a function modifies a global variable**, the **change persists throughout the program**.

FOR EXAMPLE:

1----------

```
int x = 10; // Global variable

void function() {
    int x = 20; // Local variable (hides global x)
    x = 30;          // Changes only the local x, not the global one
}

int main() {
    function();
    // The global 'x' is still 10
}
```

2----------

Void function(*p)

```
int main(){
        function(&x);
}
```

// function has been called through the address and so when the variable is accessed it is accessed for real not a dummy value but the real value of the main function.

MALLOC

* Malloc is a built in function of **stlib.h**.

* which is a short form of **Memory Allocation**.

* It is used to dynamically allocate a **single large block of contiguous memory** according to the size specified.

* It **dynamically allocates** memory at **runtime** instead of **compile-time**.

* **REMEMBER THAT IT DOES NOT INITIALIZE MEMORY, STILL GARBAGE LIKE YOU**

* **Always multiply by `sizeof(datatype)` for portability.**

***Always check if `malloc()` returned `NULL`** to avoid **segmentation faults.**

***** Memory allocated with `malloc()` **must be released** with `**free()**` to **avoid memory leaks**.

SYNTAX: (void *) malloc (size_t size)

* it requires a size and returns a void pointer

* That pointer points to the **first byte** of the **allocated memory else returns NULL.**

* size_t is **defined in <stdlib.h>** as the **largest unsigned int**

* **UNSIGNED INT as** size can never be negative

/An unsigned int is a data type in programming that represents a non-negative integer. Unlike a standard (signed) integer, which can represent both positive and negative numbers, an unsigned int can only represent values that are zero or positive.**/**

* **Why a VOID POINTER?**

***Malloc simply allocates memory requested by the user

*it has **no idea of what type of data its pointing to**.

* As void pointer can point to **any kind of data,** it can be **typecasted to an appropiate type.**

For example:

int *ptr = (int *) malloc (4)

i) Malloc allocates 4 bytes of memory and then **returns a void pointer.**

ii) We can **typecast the returned pointer to any type we want.** Here, we typecasted it to a integer pointer.

iii) Then **it will be stored inside** our given pointer, and we can find the address where it was allocated to.

NOTE: The memory allocation might actually fail, so we should also take the possibilities of it failing.

```
if (ptr == NULL) {
```

```
        printf("Memory allocation failed\n");
        return 1;
}
```

CODE:

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

int main(){
    int n;
    scanf("%d", &n);
    int* ptr = (int *)malloc(n*(sizeof(int)));
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
}
    for(int i=0; i<n; i++){
        scanf("%d", ptr+i );      //directly passes in the addresses of contigous memory
    }
    for(int i=0; i<n; i++){
        printf("%d\n", *(ptr+i));    //passing the dereferencing operator to find value from address
    }
}
```

Bit Field

A bit field is **a way to define a structure** where **each member uses only a specified number of bits** instead of taking up an entire `int` or `char` or other data type size.

* In other words, it **allows you to pack multiple small integer values** into **a smaller amount of memory**.

Why use bit fields?

* To save memory when you only need a few bits to represent values.

Use cases:

i) Hardware programming (where you work with registers).

ii) Network protocols (bit-level flags and configurations).

iii) Compression, optimization, and memory-tight scenarios.

Syntax:

****unsigned integer**** **a type of integer that can only represent non-negative values (zero and positive numbers). It does not store negative values, which allows it to use all of its bits for representing larger positive numbers.**

in a 32 bit system, where we can express:

signed--------> -2,147,483,648 to 2,147,483,647

un**signed-----> 0 to 4,294,967,295**

```
struct example{
    unsigned int a : 3;          // a uses 3 bits
    unsigned int b : 5;          // b uses 5 bits
    unsigned int c : 1;          // c uses 1 bit
}
```

- ## How it works?

  * The compiler packs all the fields together into the **smallest chunk of memory possible**. All of them are **tightly packed bits**.

  * **a, b, c** are altogether taking **9 bits**.

  * We can **still access & assign values** normally.

- ## Size Constraint:

  ```
  struct example e;
  e.a = 5;          // okay, 5 fits in 3 bits (max value = 7)
  ```

```
e.b = 20;                    // okay, max value for 5 bits is 31
e.c = 1;
```

****If we assign a value larger than the field can hold,** It will **wrap around** or will **lose the extra bits** **

Example:

```
#include<stdio.h>

struct Flag{
        unsigned int isOn : 1 ;                  // 1 bit for on/off
        unsigned int mode : 2;                   // 2 bits for mode (0-3)
        unsigned int speed : 3;                  // 3 bits for speed (0-7)
};
int main(){
        struct Flags f = {1, 2, 5};
        printf("isOn = %u, mode = %u, speed = %u\n", f.isOn, f.mode, f.speed);
        // isOn = 1, mode = 2, speed = 5
          return 0;
}
```

- ## Memory Location

The bit fields are **usually packed in an **`int`** or **`unsigned int`** **type** **internally**.

*You **cannot take the address** **of a bit-field member** (like `&f.isOn` is not allowed), because it **might not be byte-aligned.**

*Bit-fields are not stored as individual memory locations but are **packed together within a single byte or word.**

*Their position depends on how the compiler arranges them, often to save space.

- ## LIMITATIONS

Compiler-dependent packing: Different compilers may align bit fields differently.

- You can't have arrays of bit fields.

- You can't use `sizeof()` on individual bit fields (it will give size of the entire struct).

- Can only use `int`, `unsigned int`, or `_Bool` as the base type (in most compilers).

What happens if you store a larger value?

If you try to store a value **larger than the maximum** that **fits into the given bits**:

* Only the **lower (least significant) bits are stored**, and the **higher bits are discarded**.

* This effectively results in wrap-around or modulo behavior.

EX:

```
#include <stdio.h>
struct Example {
    unsigned int a : 3;    // can hold values from 0 to 7
};
int main() {
    struct Example e;
    e.a = 13;              // binary: 1101 (4 bits), only last 3 bits will be stored (101 = 5)
    printf("%u\n", e.a);          // Output will be 5
    return 0;
}
```

UNION

- **Defintion:**

*A **special data type** that **allows storing** different **data types** in the **same memory location**.

* A union **shares memory among all its members**. This means only one member can store a value at any given time.

- **Why do we use it?**

  *When we **assign a value to a member**, **the other members** become **undefined,** when tried taken access of.

  * That is, we use the **same memory space for multiple activities** and the **same memory** is used multiple times **overwriting over it** multiple times with **different variables**.

- **Memory Allocation----------**

  * The **size of a union** is equal to the **size of its largest member.** For example:

  EXAMPLE:

```
union Data {
        int i;
        float f;
        char str[20];
};
```

  Here, int i might take **4 bytes.**
  float f might take **4 bytes.**
  char str will take 20 bytes.

  ***20 bytes of space is max union space**

  *Writing in one value will **erase the 2 other values.**

- **EXAMPLE [to store mutiple values]:**

```
#include <stdio.h>

union BitField {
    int combined;                 // 4 bytes (32 bits)
    struct {
        unsigned int first : 8;           // 8 bits
```

```
        unsigned int second : 8;        // 8 bits
        unsigned int third : 8;         // 8 bits
        unsigned int fourth : 8;        // 8 bits
    } parts;
};

int main() {
    union BitField u;
     u.parts.first = 1;
     u.parts.second = 2;
     u.parts.third = 3;
     u.parts.fourth = 4;
     printf("combined: %d\n", u.combined);
     printf("first: %d, second: %d, third: %d, fourth: %d\n",
              u.parts.first, u.parts.second, u.parts.third, u.parts.fourth);
     return 0;
}
```

so, the values of the integers wont be lost even if they are together as they together use only **8 bits only if they are mentioned in a struct** but **str takes 20 bytes altogether.**

MEMORY ALLIGNMENT----------

```
#include<stdio.h>

union bit{
int dec;      // 4*8 = 32
struct{
     unsigned int a:8;
     unsigned int b:8;
     unsigned int c:8;
     unsigned int d:8;
}miu;
```

```
};

int main(){
        union bit x;
        x.miu.a=4;
        x.miu.b=5;
        x.miu.c=2;
        x.miu.d=1;
        printf(" --------ASSIGNMENT OF FRAGMENTS---------\n");
        printf("%d %d %d %d\n",x.miu.a,x.miu.b,x.miu.c,x.miu.d);
        printf("combined=%d\n", x.dec);
        printf(" --------ASSIGNMENT OF COMBINED---------\n");
        x.dec=2024;
        printf("%d %d %d %d\n",x.miu.a,x.miu.b,x.miu.c,x.miu.d);
        printf("combined=%d\n", x.dec);
}
```

**When we assign the fragments we will get the fragments getting value together unscathed as they have been properly MEMORY ALLIGNED.

**But, assigning outsider jeoperdises all of it.

**Either struct or outsider gets to get assigned. One at a time.

- **Binary representation of a char or a number or a ASCII----------**

```
#include <stdio.h>

struct bits {
        unsigned int a0:1;              //LSB Least Significant
        unsigned int a1:1;
        unsigned int a2:1;
        unsigned int a3:1;
        unsigned int a4:1;
```

```
        unsigned int a5:1;
        unsigned int a6:1;
        unsigned int a7:1;        //MSB Most Significant
};
union test {
        char ch;
        struct bits b;
};

int main(){
        union test t;
        t.ch=13;    // 13 in binary is 00001101
                    printf("%d%d%d%d%d%d%d%d\n",
                    t.b.a7, t.b.a6, t.b.a5, t.b.a4,
                   t.b.a3, t.b.a2, t.b.a1, t.b.a0);
         return 0;
}
```

// here the a7 is the MSB and the a0 is the LSB. When char of 8 bits are given in struct along with struct of 8 bits , the memory space of struct is taken as they take on the same memory space.
// The `union` allows that byte to be broken down into `a0` to `a7`.

FILE

- To Write something:

fputc()

* Writes a single character.

* **Prototype:** fputc(**char** character, **FILE** *stream);

// first argument is a character

// second argument is the pointer to the file

***On success it returns the character written,**

***On failure it returns int EOF (end of file marker usually -1)**

fputw() [Not Standard in C]

* Writes a **integer** to a file.

* **Prototype:** fputw(int value, FILE *stream);

***On success it returns the value,**

***On failure it returns int EOF (end of file marker usually -1)**

fputs()

***** Writes a **string to a file** (**excluding the null terminator**).

* **Prototype:** fputs(const char *str, FILE *stream);

***On success it returns the a non-negative value,**

***On failure it returns int EOF (end of file marker usually -1)**

fprintf()

* **fprintf** is a **formatted output function** in C that **writes data to a file stream** (instead of the console like printf). It is part of the <stdio.h> library.

* **Prototype:** fprintf(FILE *stream, const char *format, ...); // ... are the additional args

EXAMPLE:

```
#include<stdio.h>
int main(){
    FILE *file = fopen("output.txt", "w");        //opening file in write mode
    int num = 42;
```

```
    float pi = 3.14159;
    char name[]="Alice";

    fprintf(file ,"Number: %d\n", num );
    fprintf(file ,"Pi: %.2f\n",pi );
    fprintf(file , "Name: %s", name);
    fclose(file);
    return 0;
}
```

-

fwrite()

* fwrite **writes a block of data** from **memory to a file** directly.

* **Prototype:** fwrite(*data, size_t size, size_t nmemb, *file)

PARAMETERS EXPLAINED

ptr: Pointer to the data to be written (e.g., an array or struct).

size: Size (in bytes) of each element (use sizeof).

nmemb: Number of elements to write

stream: File pointer should be opened in **binary mode ("wb")**

*** On success returns the number of elements written (equal to nmemb)**

*** On failure returns a number less than nmemb**

- Read something:

fgetc()

* Reads a **single character** from a file

***Prototype:** fgetc(FILE *stream);

// It takes only one argument which is the **File stream pointer**

RETURNING VALUE:

* **On success returns character read**
* **On failure returns a number EOF.**

EXAMPLE USE [to print the whole file]:

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("file.txt", "r");
    if (fp == NULL) {
        printf("Failed to open file.\n");
        return 1;
    }

    char ch=fgetc(fp)) ;
    while ((ch != EOF) {      //unless file pointer is returned as EOF
        putchar(ch);          //Print character
          ch=fgetc(fp) ;    //File pointer is automatically moved with each reading
                        //we move & initialize
    }
    fclose(fp);
    return 0;
}
```

fgetw() ⚠ Avoid using in modern C.

*Reads an **integer** from a file.

* fgetw(FILE *stream);

fgets()

*Purpose: Reads **a line from a file** until a **newline or EOF**.

*__Prototype:__ char *fgets(char *str, int n, FILE *stream);

// **char *str** is where **the char array read from a file** will be **store**

// int n is the **maximum number of chars** to read **including** **'\0'**

if line is **smaller than n-1** fgets() stops at **newline \ EOF**

if line is **bigger than n-1** fgets() makes sure to stop with **'\0'**

// FILE *stream

RETURNING VALUE:

* **On success returns** **same buffer pointer**
* **On failure returns a number NULL**

fscanf()

*Reads **formatted data** from a file (like: scanf)

*__Protoype:__ fscanf(FILE *stream, char *ptr.....)

RETURNING VALUE:

* **On success returns** **number of successful assignments**
* **On failure returns a number EOF**

fscanf(fp, "Name: %s Age:%d Salary:%f",name,&age,&salary);

From the **file,** we will read the name, the age & the salary and then we can copy it to the **char variable** in the beginning of the function which we have mentioned.

fread()

- *Purpose: Reads a block of data from a file.

  * Prototype: size_t fread(void *ptr, size_t size, size_t count, FILE *stream);

  Returns: Number of items read successfully.

-

WRITING TO A FILE

*To write to a file, Data type should be FILE

* A pointer should be announced as it is a pointer which is pointing to a file.

*if path needs to be mentioned then

*	FILE	*pF= fopen("C:\\Users\\Knight\\OneDrive\\text.txt", "w");

[Here every slash should be converted to double slashes **reversed**]

*	FILE fopen *pF ("text.txt", "w");

// Here a file is opened and then a pointer pointing to that file is called.

//Then the file name is given text.txt

// many arguments might be there for the next part "w" used for writing right now.

	// w---->write	r---->read	a--->append

fprintf

*fprintf(pointer_name , "bomboklaat");

//we will write the given string to the file if we use "w".

//we will add / append some file if we use "a". [adds up more strings]

*fclose(pointer_name); // to close any file that are open

REMOVE

remove("text.txt") will help to know if a file was removed.

remove("text.txt")==0; when we can remove a file successfully.

READING A FILE

If a file is unreadable\unlocateable, then the given pointer will be found NULL i.e pF==null

- **Code to read a line:**

```
#include <stdio.h>
int main()
{
FILE  *pF= fopen("C:\\Users\\Knight\\OneDrive\\text.txt", "r");
char buffer[255];  //buffer acts as a container to hold texts of our file one line at a time
fgets(buffer, 255 , pF);      //  (array_where_we_input, maximum_size, pointer_to_the_file);
printf("%s", buffer);
fclose(pF);
return 0;
}
```

-
```
#include <stdio.h>
int main()
}
FILE  *pF= fopen("C:\\Users\\Knight\\OneDrive\\text.txt", "r");
```

```
char buffer[255];    //buffer acts as a container to hold texts of our file one line at once
if(pF==NULL){
        printf("unable to open file");
}
else{
        while(fgets(buffer, 255 , pF) != NULL){      // file will be read until fgets gets        null(end)
        printf("%s", buffer);
        }
}
fclose(pF);
return 0;
}
```

Link List

I) Making node:

A node contains data & link part. It is nothing but a combination of different types of data. Now, to make combination of different types of data in the same group which can only be done by **struct**.

Self Referential structure: A structure which contains a pointer to a sturcture of the same type. Inside there can be any type of data char, int, float etc. We will use this to create a **referential structure for creating a node of the single linked list**. Ex:

```
struct abc{
```

```
        int data;
        data_type member2;
        data_type member3;
        data_type member4;
        struct node    *link;              // contains a struct pointer pointing to itself
}
```

* There might be **multiple types of data**

*** BUT THERE MUST BE ONLY A SINGLE LINK**

MAKING OF THE NODE:

```
#include<stdio.h>
#include<stdlib.h>       // These 2 libraries are necessary

struct node{
        int data ;
        struct node *link;
}:

int main(){
        struct node *head = NULL;                  // initialization of struct pointer to NULL
        head = malloc(sizeof(struct node));
                  //allocation of memory for both link & information
                // use of pointer here is done so that it can be linked to other nodes too
                // because the node will have an address of itself

        head->data = 45;      // Putting data in for the information part
        head->link = NULL // next address is yet not given
```

// A NODE HAS BEEN MADE!!

```
}
```

ii) Making a SINGLE LINKED LIST:

* We wil create a node having the address of another node having the address of another node and so on to create a single linked list.
* The ending of the list should have **NULL** to show the ending of the list.

CODE:

```
#include<stdio.h>
#include<stdlib.h>        // These 2 libraries are necessary

struct node{
        int data ;
        struct node *link;
}:

int main(){
        struct node * head = malloc(sizeof(struct node));
        head->data = 45;
        head->link = NULL;        //replacing garbage (you)

// THE HEADING LINK IS MADE
        struct node *current = malloc(sizeof(struct node));
        current->data = 50;
        current->link = NULL;     //replacing garbage (you)

// LINKING PART

        head->link = current;
```

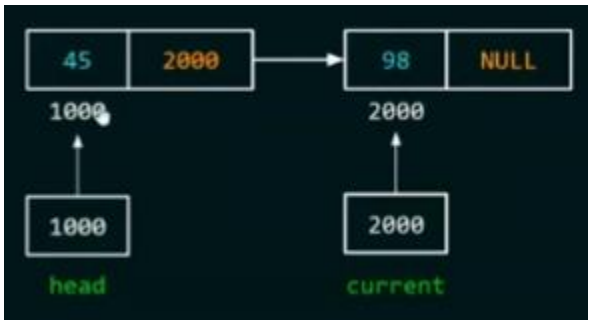
// The link part of the first struct now contains the **address of the next struct**

// **Linking is complete and this proccess can be repeated multiple time to create a list**

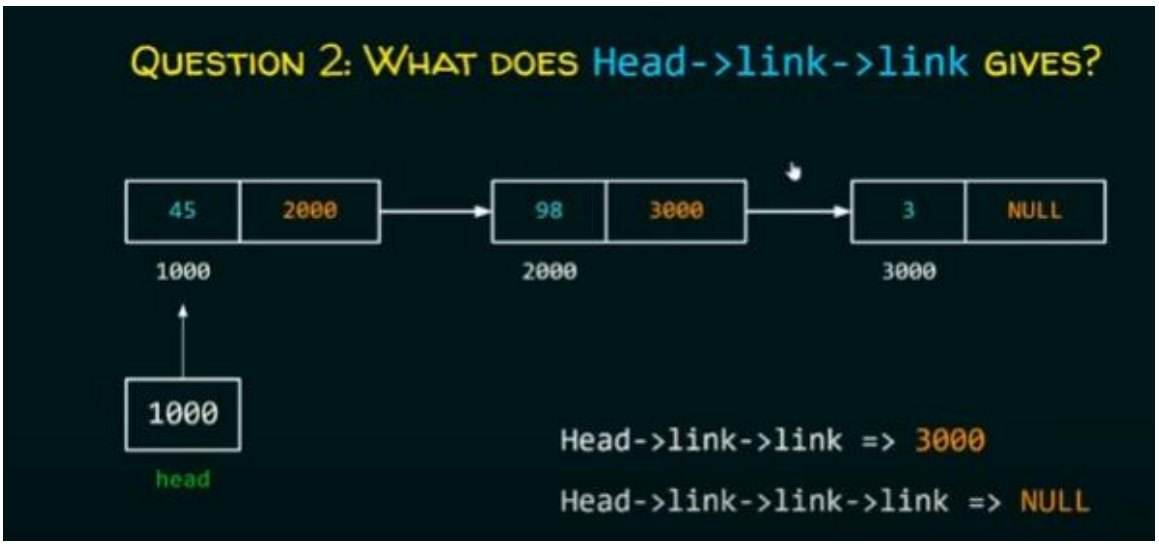
// we should have **NULL in the end or else it would point to garbage**

}

Pictorial representation:



Important:



* adress points to address points to address
* we can access the link of one node and then again access that link to get to the address of the second node and then again for the third as LINK actually gives us a struct address.

Optimised Code:

```
#include<stdio.h>
#include<stdlib.h>        // These 2 libraries are necessary

struct node{
        int data ;
        struct node *link;
}:

int main(){
        struct node *head = NULL;                    // initialization of struct pointer to NULL
        head = malloc(sizeof(struct node));
                  //allocation of memory for both link & information
                // use of pointer here is done so that it can be linked to other nodes too
                // because the node will have an address of itself

        head->data = 45;      // Putting data in for the information part
        head->link = NULL // next address is yet not given

// A NODE HAS BEEN MADE!!
}
```

ii) Making a SINGLE LINKED LIST:

* We wil create a node having the address of another node having the address of another node and so on to create a single linked list.
* The ending of the list should have **NULL** to show the ending of the list.

CODE:

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

```
#include<stdlib.h>        // These 2 libraries are necessary

struct node{
        int data ;
        struct node *link;
}:

int main(){
        struct node * head = malloc(sizeof(struct node));
        head->data = 45;


        struct node *second = malloc(sizeof(struct node));
        current->data = 50;

        head->link = current;

        struct node *third = malloc(sizeof(struct node));
        third->data = 65;
        head->link->link = third;

        // by head->link we access the address of the second part
        // then again by head->link->link we access the link part of the second address
        // then put in third for the address of the next struct we made


}
```

TRAVERSING A SINGLE LINKED LIST

* we can visit each node of a single linked list until the end node is reached.
* for that we have to calculate the **total number of nodes** by **traversing the linked list**

*** For this we first have to build the nodes connecting each other one by and only then can we implement this function by making it.**

CODE: [***O(N) is the time complexity***]

```
void count_of_nodes( struct node *head ){
```

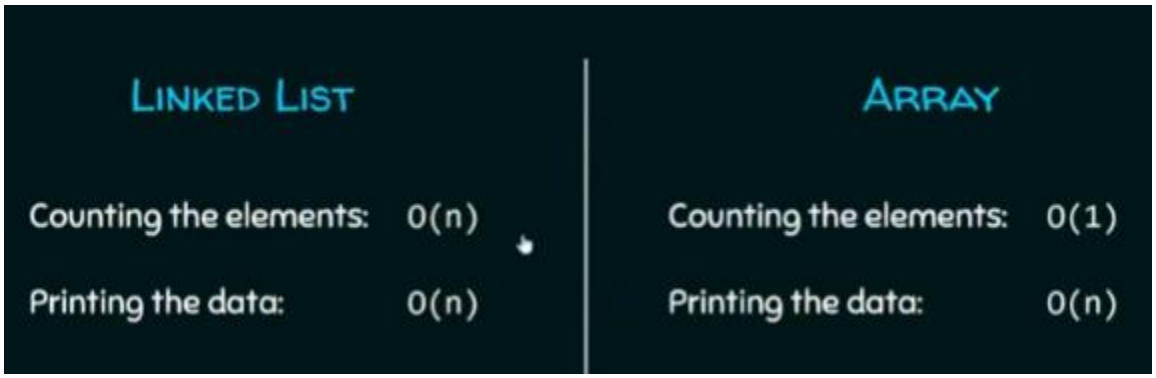
```
        int count =0;

        if(head == NULL){
                        printf("Linked List is empty");          // if NULL detected in first node
                }

        struct node *ptr = NULL    // address tracking pointer

        while(ptr != NULL){
                count++;                    // count incremented with each iteration
                ptr = *ptr->link;           //until null is detected we keep moving up the nodes
        }

        printf("%d",count);        //count of nodes is printed
}
```

PRINTING DATA OF SINGLE LINKED LIST

CODE: [*O(N) is the time complexity*]

```
void print_data(struct node *head){

        if(head == NULL){
                printf("LINKED LIST IS EMPTY");
        }

        struct node* ptr=NULL;          // removing the garbage
        ptr = head;                     // address containing the first pointer

        while(ptr!=NULL){
                printf("%d", ptr->data);          // printing data for each case
                ptr = ptr->link;                  //MOVING UP NODE
        }

}
```

ARRAY VS LINKED LIST



INSERTION

Appending a Node

void append(struct node *head, int data**){** [***O(N) is the time complexity***]

```
stuct node *temp;
temp->data = data;          // creating a temporary node with desired data
temp->link = NULL;          // Adding a cap at the ending node

struct node* ptr=NULL;      // tracker node initialization
ptr = head;                 // address containing the first pointer

while(ptr!=NULL){
        printf("%d", ptr->data);        // printing data for each case
```

```
            ptr = ptr->link;                    //MOVING UP NODE
        }

        //Now as the end is reached we will link the temp node with our node

        ptr->link = temp;

        //Linking to our required temp with the required data is done

}
```

*** We call the function in the main program and then put in the arguments as required then AUTOMATICALLY a node will be appointed at the end of the link.**

Main body:

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

struct node(){
        int data;
        struct node *link;
};

int main(){

        append(head,67);

        // we appended a node with 67 to the list

}
```

OPTIMIZATION: **[*O(n) is the time complexity*]**

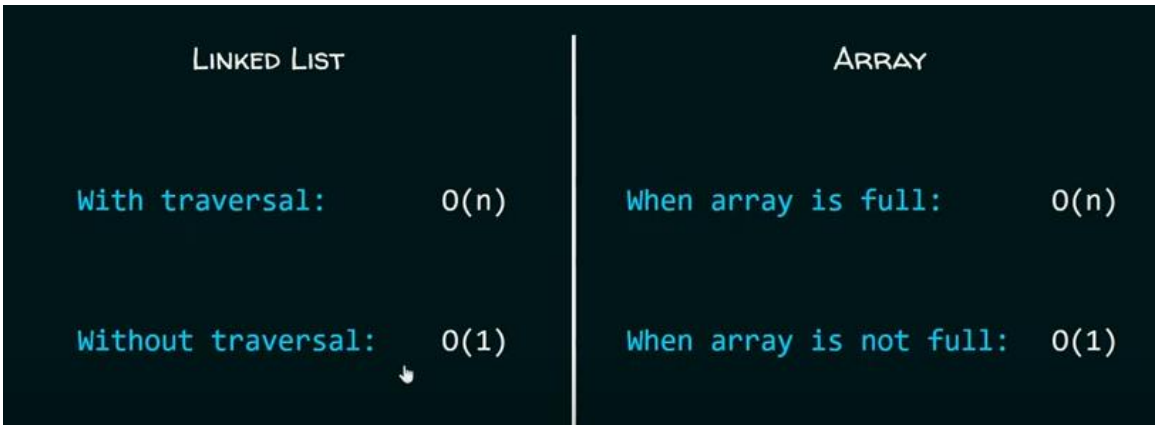
We can append immediate as we make the node with the help of the function with the help of the main function. This will save time for us by **leaving out the traversing part**.

CODE:

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

```
struct node{
        int data;
        struct node* link;
};



struct node * append(struct node *ptr, int data){

        struct node * temp = malloc(sizeof(struct node));    //initialization ALWAYS NEEDED
        temp->data = data;
        temp->link = NULL;

        ptr->link = temp;
        return temp;
}

int main(){

        struct node *ptr = malloc(sizeof(struct node));
        struct node*head = malloc(sizeof(struct node));
        head = ptr;
        head->data = 45;          // If initialization of head is not done then we will get
        head->link = NULL;        // garbage in the heading pointer as function only appends

        ptr = append(ptr,65);    // why equal to ptr?
        ptr = append(ptr,75);    // if we equalise them then ptr starts moving to the next point
        ptr = append(ptr,85);    // ptr then acts as Moving Node & function appends & links


}
```

IF we use array:



Prepending a Node

* Before we added another node in the end of the list and we used the moving pointer to move to next one.

BUT, now we use the **head pointer** to **move the pointer to the back** as we need

* Make a new temp with required data
* **add the address of the old head pointer to the prepended one**.
* We will **return the new temp to our header variable that we had.**

// ALWAYS REMEMBER TO INITIALISE THE HEAD POINTER BEFORE ANY PREPENDING OR APPENDING //

CODE:

```
struct node* prepend(struct node *head, int data){
```

```
    struct node * temp = malloc(sizeof(struct node));


    temp->data = data;
    temp->link = head;
    return temp;
};




int main(){

        struct node *ptr = malloc(sizeof(struct node));
        struct node*head = malloc(sizeof(struct node));
        head = ptr;
        head->data = 45;          // If initialization of head is not done then we will get
        head->link = NULL;        // garbage in the heading pointer as function only appends

        ptr = prepend(ptr,65);    // why equal to ptr?
        ptr = prepend(ptr,75);    // if we equalise them then ptr starts moving to the next point
        ptr = prepend(ptr,85);    // ptr then acts as Moving Node & function appends & links


        // 45 -> 65 -> 75 -> 85
}
```

Inserting at certain position

CODE:

```
void insert(struct node * head, int data, int pos){



            struct node *ptr = head;    //moving pointer from head
            struct node *temp ;         // taking desired node
            temp->data = data;        //INSERTION OF DATA
```

```
        temp->link = NULL;

        pos--;                  // position tracker

        while(pos!=1){          // postion if 1 location found
            ptr = ptr->link;    // moving up pointer if not found
            pos--;              // moving up
        }

        temp->link = ptr->link; // new node has next link
        ptr->link = temp;       // old node has added node link

}
```

DELETING A NODE

* Here we free the memory at the end / beginning of the array and then we clear up the memory of the part we do not want.

DELETING LAST NODE

CODE:

```
struct node* del_last(struct node *head){
        if(head==NULL){             //IF HEADER IS NULL ALREADY NOTHING TO DO
            printf("List is already empty!");
    }
```

```
        else if(head->link == NULL){            // IF FIRST NODE HAS DATA & NULL
                free(head);
                head = NULL;            // destroy if we see NULL as the last has NULL
    }
        else{                   //MOVING FORTH IF EMPTY NULL ONES NOT FOUND
                struct node *temp = head;
                struct node *temp2 = head;

                while(temp->link!=NULL){
                        temp2 = temp;           // one holds the pointer from before
                        temp= temp->link;       // another holds the pointer to destroy

                }

                temp2->link = NULL;             //    Linked to NULL as it ends here
                free(temp);                     // freeing the last memory to kill off
                temp = NULL;                    // removing garbage memory
    }

}
```

DELETING THE FIRST NODE

CODE:

```
struct node* del_first(struct node* head){
        if(head==NULL){
                printf("LIST IS ALREADY EMPTY      :(      ");
```

```
        }
        else{
                struct node*temp = head;    // we take head in temp
                temp = head->link;              // we take the head's link in
                head->link = NULL;
                free(head);                     // free it to destroy it
        }
                return temp;            // return link address of next one as new head
}
```

DELETING A CERTAIN NODE

CODE:

```
void del_pos(struct node **head, int position ){// head to be passed in a reference
                                                // so that we can delete immediately

        struct node *current = *head;           // previous current both needed to
        struct node *previous = *head;          // keep old link & destroy target one

        if(*head == NULL){
                printf("List is already empty :( ");
        }
        else if(position == 1){
                *head = current->link;  //link of the deleted array to be kept
                free(current)           // chosen one to be deleted
                current = NULL;         // clearing out garbage
        }
```

```
        else{
            while(position!=1){
                previous = current;        //moving up prev one
                current = current->link;//moving up new one
                position--;                //keeping track of position
            }
                previous->link = current->link; //to keep new link in old node
                free(current);//freeing up after linking remnant nodes
                current = NULL;
        }
}
```

DELETING ENTIRE NODE

CODE:

```
struct node* del_list(struct node *head){
        struct node * temp = head;

        while(temp!=NULL){
            temp = temp->link;          // move temp forward
            free(head);                 // old saved temp to be freed
            head = temp;                // new head to be updated to move forth
        }

        return head;                // at some point head will have nothing but NULL

}
```