C Programming

1) Number in C
   - Signed → 2's comp
     - +ve no. → Normal binary
     - -ve no → in 2's comp.
   - unsigned → Normal binary

2) 0101 <-- 2's comp --> 1011

3) 2's comp to decimal : do as binary to decimal & multiply MSB by (-1).

4) copying no. to higher bit representation
   - Signed no. : Sign extension
   - unsigned no. : zero extension

5) Default size:
   int (2B/4B) char (1) short (2B) float (4B) long (4B/8B) Double (8B) long double (10)

6) Format specifier:
   char (%c) signed int (%d) unsigned int (%u) Hexadecimal format (%x) float (%f) address (%p) → given in hex string (%s)

7) By default int, short, char are signed.

8) In signed represen. remove extra 0's and 1's
   00000011 = 011 11111101 = 101

9) printf does not care of data type, it only uses binary as per format specifier.

10) short x ≡ short int x

11) Extension: only look RHS, if
   signed → sign extension
   unsigned → zero extension

12) Truncation: Take required no. of bits from RHS.

13) char is a small int of 8 bits.
   char c = 9 // 0000 1001
   If we give character then its ASCII value is stored.

14) Integer Promotion: Whenever small integer (short/char) is used in expression, its implicitly converted to int (32 bits).

15) Type conversion:
   long double
   double
   float
   unsigned long int
   long int
   unsigned int
   int
   char, short

16) Make sure in an expression, all variables are of same type

17) 0 is false, Any value other than 0 is True

18) 0 = '\0' = NULL

19) Without parenthesis, only one line is associated with if/else, loops.

20) if(.)
   if(.)
   else
   } Without parenthesis, else is associated with nearest if.

⇒ Switch:

1) Expression must be int/char/short or expression that evaluates to int.
2) Labels are unique constant can't be variables
3) cases & default stmt can be in any order and break is optional.
4) stmt outside (not in any case) are ignored.
5) It goes to default when no case matches.

NOTE: printf("Hey") ;) → Not an error

1) break takes out of loop/switch.
2) continue takes to next iteration (condition checking) of loop.
3) Shorthand Notation:
   v op= exp ⇒ v = v op (exp)
4) ++(constant/expr) gives error
   ie ++2 , ++(a+b-2)
5) increment, decrement operator is only used with variable.
6) Left shift << (* by 2 if MSB is 0)
   Right shift >> (÷ by 2)

NOTE: $2^{3^2} \neq 2^6$, $(2^3)^2 = 2^6$

7) 1's com (~x) = 2's com (-x) − 1

8) p=(i,j,k) ← value of last exp k is assigned to p.

9) p=i,j ← p=i becoz = has higher precedence than ,

10) !0 = 1
!<non zero> = 0

⇒ Ternary Operator:

exp1 ? exp2 : exp3 ; ⇒ if exp1 exp2 else exp3

- convert nested ternary operator into if-else then solve.

- a?b:c?d:e

```
if a
 ↳ b
 ↳ if c
    ↳ d
    ↳ else e
```

a?b?c:d:e

```
if a
 ↳ if b
    ↳ c
    ↳ else d
 ↳ else e
```

⇒ Coma operator: evaluated from left to right & value of right most expr. is value of combined exp.

(x=10, y=5, x+y) equiv. to 15

((i<=5, j>=0) ← useless) equiv. to j>=0

Precedence Table

	Operators	Associativity
	(), [], →, ., ++, -- (Postfix)	L→R
unary	!, ~, +, -, *, &, sizeof, typecast, ++, -- (Prefix)	R→L
Arithmetic	*, /, %	L→R
	+, -	L→R
Shift	≪, ≫	L→R
Relational	<, <=, >, >=	L→R
	==, !=	L→R
Bitwise	&	L→R
	^	L→R
	|	L→R
Logical	&&	L→R
	||	L→R
Ternary	?:	R→L
Assignment	=, +=, -=, *=, /=, %=, &=, |=, ^=, ≪=, ≫=	R→L
coma	,	L→R

1) exp1 && exp2 exp1 || exp2

Always evaluate exp1 first irrespective of what exp1 & exp2 is. K=(i)||(j||k) ← first

2) Short circuiting:

T || _ = T F && _ = F

↳ Don't evaluate

3) Effect of Postfix operator comes into effect after sequence point (; && ||)

4) In && and ||, put brackets acc to precedence then solve from left to right.

(e1 && e2) || e3 → e1 || (e2 && e3) →

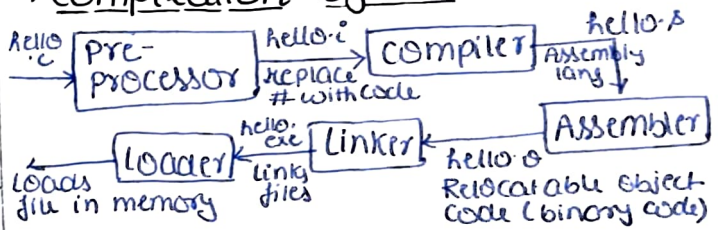
⇒ Function:-

1) Declaration: int mul(int a, int b);
or int mul(int, int);
or mul(int, int); ← By default return type is int

2) main() { fun(); } fun() is not declared

compiler assumes that fun() is fu^n of int return type. Later in defination, if assumption is crct then no error otherwise compile time error.

⇒ Compilation System:



- Linker is a manual task. It by default links required files like printf.o.
- Header file stdio.h only contains declaration of buildin functions. .o file contains definition.
- main() { printf(); } It will compile becoz compiler will assume printf is f^n of int return type which is crct.

NOTE: file1 main() { } file2 main() { } These 2 files can't be merged as a program can't have 2 main function.

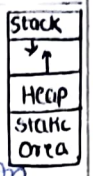
NOTE: main() { int a; { int b; } } ← Block — NO ERROR

NOTE: x = f() + g() is ambiguous. f() & g() can be evaluated in any sequence, order is not fixed.

- Exception (&& || , ?:) order is fix from left to right.

* Sequence points: (; , && || ?:) side effect immediately comes after it.

⇒ Memory layout:



1) Static: compile time allocation (code, global & static variables)
2) Stack: Runtime allocation (Activation record)
3) Heap: Runtime alloc. (using malloc)

⇒ C storage classes:

- auto - register - static - extern
- scope : can i see variable
- Lifetime : is variable in memory.

1) Auto storage class:

- storage: stack
- scope: within block
- Initialize: Garbage
- Lifetime: end of block

auto int a; ⇔ int a;

- Its default storage class for all local var.
- not available to linker.

NOTE: auto & register are always used with local var. If used with global var then compilation error.

2) Register storage class:

- using it we recommend compiler to keep it in register if possible.
- same as auto

3) Static storage class:

- storage: static area
- Initialize: zero
- Scope: within block
- Lifetime: end of program
- Not available to linker as can't be accessed outside file.

static int a; ← executed only once.

NOTE: Global var is also initialized to 0
↳ can be used in other files.
↳ none of these classes.

4) Extern storage class:

- used to refer global var or functions in same file or other.
- no memory is allocated.
- It tells compiler "Don't worry, variable is somewhere present".
- Functions are by default extern.

NOTE: Fu" can't be defined inside block. can be declared or called.

* Declaration: needed by compiler
  extern int x;
  int func(); = extern int func();
* Definition: needed by linker
  int x; extern int x = 5;
  int fun(){printf("Hey");}
* extern int x = 5; It can't be local as extern always points to global variable.

- storage: static
- Initialize: 0
- Scope: Global
- Lifetime: end of program
- Available to linker

* If var/fu" is declared but not defined, it will have linker error only if var is used.
* Function parameter can't be static (compile time error)

NOTE: extern int a; } no error
extern int a;

* Globally we can't have any stmt other than declaration & initialization of variables.

```
int i=10;
i=5; ← compile time error
```

#Pointer

1) Initialized with garbage value.
2) int* p ≡ int *p ≡ int * p
3) We can't have address of constant, expression or register variable.
4) All pointer size is same (By default 8B).

⇒ Array:-

1) int arr[5] = {1,2}; // 1,2,0,0,0
   static int arr[5]; // all ele 0
   char arr[5] = {'a','b'}; // a,b,10,10,10
2) 0 ≡ '\0' ≡ NULL
3) char arr[5] = "abcd"; // a,b,c,d,\0
   char arr[4] = "abcd"; // a,b,c,d
4) char array ending with NULL '\0' is treated as string.
5) int arr[] = {'a','b','c'}; // arr[3]
   int arr[] = "abc"; // arr[4]
   ↳ compiler puts a '\0' char & treat it as a string.

6) int arr[]; ← error
Either provide dimension or declaration

7) Array name can't be used on LHS of =.

8) sizeof() does not evaluate expression
sizeof(i++) ← i will not increment.
It tells size in Bytes.

9) Pointers are incremented by the size of (*ptr) whose addr it's storing.
p+i = p + i * sizeof(*p)

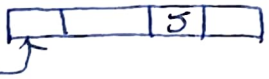
⇒ 1D array:

1) int a[10]; ← Base addr
a points to first ele a[0].

2) a does not have lox so it can't be on LHS of assignment.

3) x = B1 B2 B3 B4
Big endian: B1 B2 B3 B4
Little endian: B4 B3 B2 B1 (Mostly used)

4) p can be used as a i.e. p[2]=5



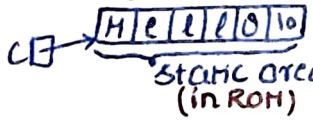
5) When value is fetched from little endian, it's first reversed byte wise then converted to decimal.
Ex: x = d3 d2 d1 d0
Printf("%d", x); decimal of 0 0 0 0 d2 d3

6) a[i] = *(a+i)
&a[i] = (a+i)

7) int a[5];
a+10 ← valid
*(a+10) ← May lead to Runtime error
"Kisi anjan ke ghr me nhi ghus skte"

8) char c[] = "Hi"; → c=c+1 invalid
char *c = "Hi"; → c=c+1 valid as char box, its pointer

9) char *c = "Hello"; ← String constant
It cannot be modified
c[1] = 'x'; ← Invalid



10) int arr[5];
&arr is pointer to array, (&arr+1) will skip entire array.

11) printf("%s", addr); It will print from address to '\0' character.

12) Passing array as parameter:
main()
{ int a[5];
fun(a); }

fun(int *a)
fun(int a[])
fun(int a[15])
} internally same as 1st one
ignored

Only base address is passed.

13) char arr[5]; scanf("%s", arr);
Enter 4 char as %s will put '\0' char at end. If we enter 5 char, it may lead to error.

---

1) strlen(): Counts no. of char except '\0'
It returns unsigned int.
strlen("Hello") // 5
sizeof("Hello") // 6

2) strcmp(): compares char by char
strcmp(s1,s2) → 1 if s1>s2 ad>abc
→ 0 if s1=s2 ab=ab
→ -1 if s2>s1 abc<ad

3) strcat(s1,s2): concatenates s1 & s2 & stores in s1 & returns

4) strcpy(s1,s2): copies s2 to s1 & returns
•s1 size must be big enough otherwise it will lead to segmentation fault.

NOTE: a[5] = 5[a] = *(a+5)

---

1) If p is a double int pointer:
p: address *p: address
**p: int

⇒ 2D Array:

1) in 2D array int arr[5][6]
a → pointer to 1st row
&a → pointer to entire array.

2) In 2D array, ([],*)² will only give int rest all will give addr

3) a[i][j] = *(*(a+i)+j) = *(a+i)[j]

4) int *p; int **ptr; int a[2][3];
p=a; ptr=a; ptr=*a
It will give warning.

5) 2D array & double pointer are not related at all.

---

1) Pointer to an array:
int (*p)[5];
p is pointing to an int array of size 5.

2) Arithmetic on pointers depends only on LHS (ie type of pointer).

3) `int a[5][6]` , `int **p , (*p)[6]` } to get int we may use **, [][], *[]
`int a[5] , int *p` } to get int use *, []

4) Array of pointers : `int * arr[5];`

5) `char * a[2] = {"Hi", "Bye"};` ← String literals in static area
`a[1][2] = 'x'` ← Invalid

NOTE: `int *p = 0` is valid
0 ≡ '\0' ≡ NULL

6) 2D array:
`char arr[2][4] = {"Hi", "Bye"};`
`arr[1][2] = 'x'` ← valid

7) Passing 2D array to a fun":
```
main()
{ int a[5][6];
  fun(a);
}
```
internally same as ↓
`fun(int (*p)[6])`
`fun(int a[5][6])`
`fun(int a[][6])`

8) First dimension in array is optional, rest all dimensions are important.

9) `int a[5][6] ; int (*p)[6] = a;`
'p' can be used as 'a'.
`p[2][3] = 10;`

10) Pointer to 2D array:
`int arr[3][4];`
`int (*p)[3][4] = &arr;`

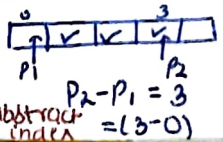
NOTE: "Hello"[3] = l
3["Hello"] = l

11) For pointer 'p' & * cancels each other
`&*(p+1) = p+1` `*&(p) = p`

12) `int arr[2][2];`
arr is a 2D array, but when we do arithmetic its treated as addr.
sizeof(arr) → 16B
sizeof(arr+0) → 8B (size of pointer)

⇒ **Difference between two pointers:**
P1 & P2 must point to elements of same array otherwise result is garbage value or error.

$$P_2 - P_1 = \frac{(P_2 - P_1)}{\text{sizeof}(P_1)}$$



1) If P1 is a pointer to an int array of size 3 then *P1 is an int array of size 3.
sizeof(*P1) = 12.

⇒ **3D Array:-**

1) `int a[2][3][4]`
- a points to 2D array of size [3][4]
- a[], *a → pointer to 1D array
- a[][], **a → pointer to int
- a[][][], ***a → int

1) Valid arithmetics on pointer:
P+3, P-3, P1-P2, P1 <= P2, P1 >= P2
P1 == P2

2) Invalid operations on pointers:
P1+P2, P1*P2, P1/P2, P1 % P2

3) void pointer can hold any type of address without typecast.
can't do *p without typecast.

4) **Nomenclature of pointer:**
* pointer to, [] array of,
() function returning
- Identify identifier, keep going right until you go out of symbol or hit ')'.
- Keep going left until you go out of symbol or hit '('.

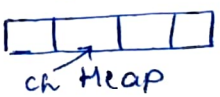
5) **Pointer to fun:-**
`int (fun)(int x)` ← fun: pointer to fun
`{ }`

```
main()
{ int (*p)(int)=fun
}
```

p can be used as fun. ie. p(5)

6) **malloc:**
- malloc(10) allocates 10B memory in heap and returns void pointer.
- `int *p = (int*) malloc(sizeof(int));` Typecasting is optional.
- If malloc fails, it returns NULL. Make sure to test this case.
- malloc makes a system call

7) free(ch) → to free allocated memory
ch is starting addr of allocated chunk, if its not starting addr it will give error.
free(ch) → Error



8) **Dangling pointer:** points to memory that has already been freed.
- Its a prob if itr dereferenced (*p) (Runtime error)

9) **Memory leak:** is memory which hasn't been freed & there is no way to access it or free it now.

10) `char *C; sscanf("%s", C);` Error as memory is not reserved for string.

NOTE: EOF : End of file (special symbol defined in stdio.h).

11) **Macros:** `#define plus(x) x+1` (no semicolon)
Everywhere replace plus(x) by x+1
- If x is expression, don't do any computation, simply replace.

---

1) C programming uses only call by value (never call by reference).

2) **Static scoping:** C uses it (By default)
- use global variable
- done at compile time.

3) **Dynamic scoping:**
- use previous fun" variable
- done at runtime.

#Structure

1) `struct xyz { };` → semicolon is must

2) `typedef unsigned char xyz;`
typedef is used to give name to custom types.
NOTE: `long i;` ≡ `long int i;`

3) `typedef struct stud { } s;` → name is optional
`s s1, s2;`

4) `struct std { int r; int marks; };`
data type ←
`struct std s = { 10, 50 };` ← Memory is allocated
`s.r = 15;` ↖ s is not a pointer

5) **Array of structures:**
`struct std s[10];`
s points to 1st structure in array
- s is not structure, s[0] is structure

6) `struct std (s);`
s is not a pointer (like array name) so, *s is error.

7) **Passing a structure to a fun:**

```
main()
{ struct stud s;
  s.r = 5;
  s.marks = 50;
  fun(s);
}
```

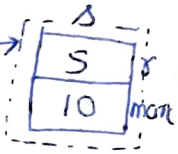
```
fun(struct stud t)
{ t.r = 10; }
```

Here t is a local variable & value of s attributes are copied to t.

8) **Pointer to structure:**
`struct std *ptr = &s;`

9) `ptr = &s` ⇒ `*ptr = s`

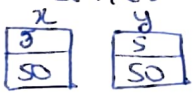
10) `(*ptr).r = 5`
`*ptr.r` will not work as . has higher precedence.



11) `(*ptr).r` ≡ `ptr->r`

NOTE: `strlen("Hello")` // "Hello" is allocated in static area then base addr is given to strlen

12) Copying one struct to another
`struct std x, y;`
`x.r = 5; x.marks = 50;`
`y = x`

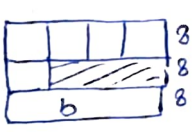


Everything of x gets copied to y.

13) Comparing two struct :- do element by element.
Can't do `(x == y)` → ERROR

14) **Storing struct on memory alignment**
A variable is stored only at places where starting addr is ÷ by size of variable.
- Start with size of biggest variable

15) `struct s { short int a[5]; long b; };` ← Treat it as 5 shorts
→ 2B
→ 8B
`sizeof(struct s) = 24B`
data type



16) If struct is not aligned sizeof(struct s) = 18B

17) By default struct is aligned for optimization.

1) C does not support chaining of operators.

- $a=b=c \Rightarrow a=(b=c)$ = is right to left
- $-10 < i < -1$, $i=-7$ $\Rightarrow (-10<-7)<-1$ $\Rightarrow 1<-1 = 0$ < is left to right

2) Infinite recursion & keep calling f" unless stack is overflow.

⇒ **union:**

- Memory is shared blw variables.
- Memory is created only for largest variable.

⇒ **Enumeration:**

- used to assign names to integers.

```
enum week {Mon, Tue};
enum week w = Tue;
printf("%d", w); //1
```

} enum week {Mon, Tue} w;

- If we do not provide value, by default value is assigned starting from 0.

```
enum week {Mon=1, Tue, Wed=0, Th=0};
                1     2     0     0
```

- Unassigned names gets value previous+1.
- value assigned is some int constant.
- All enum constants must be unique in their scope.

```
enum A {x, y};
enum B {a, x};
```

} can't have same constant in same scope. ERROR

---

1) Goal of structured programming lang is to able to infer flow of control from program text. control passes 1 inst" to another sequentially.

2) ADT (Abstract data type) defines only what operations are to be performed not how its implemented.

- supports only operations which r defined.

3) Abstract class has atleast 1 fun undefined TO instantiate, we need to create subclass define fun" then instantiate.

4) use of goto can result in unstructured code.

5) Dynamic data structure: that changes its size at runtime. Ex: Linked list. Memory is allocated at runtime.

6) In static data struct, size is fixed & memory allocated at compile time.

7) Dynamic memory allocation: allocation at runtime.

8) **Aliasing:** situation where same memory location can be accessed using different names.

```
main()
{ int x=5;
  f(&x, &x); }
```

```
f(int *a, int *b)
{
}
```

9) Assembler uses location counter to give address to each inst".

- Reference count is used by garbage collector to clear memory whose reference count becomes 0.
- Linker loader does relocation of object code.