C-programming :

09-08-21
monday

-> In the whole of c programming Recursion, Pointers and Arrays are very important topics

First C-Program :

```
#include <stdio.h>
int main(void)
{
  printf ("Hello world! \n");
}
```

Identifiers :

-> Every word in c is classified as either keyword or identifier

-> Identifier refers to name of a variable or a function name

* keywords are reserved words and we cant use them for variable or function names

-> In c-programming language in total there are 32 keywords present

Data types :

int (4 Bytes) ; char (1 Byte) , short (2 Bytes)

- signed
- unsigned

- signed
- unsigned

- signed
- unsigned

float (4 Bytes), double (8 Bytes), long double (10 Bytes)

Assigning Values to Variables :

int final_value = 100; , double balance = 75.84

char Yes = 'x'

-> what will be the output of the following c-program

```
# include <stdio.h>
int main (void) {
printf (" if I have %d bills worth %d dollars each
```

then I have %d dollars", 3, 5, 3*5);

return 0;

}

Soln: in general printf expects a string

Here %d will be replaced with respective values

the output will be

if I have 3 bills worth 5 dollars each then I have 15 dollars

→ what will be the output of the following program

```
#include <stdio.h>
int main() {
printf("3+2 equals %d and 3-2 equals %d \n",
                                    3+2, 3-2);
return 0;
}
```

Soln: we must remember that in printf all the characters strings, expression will be treated as a string and only the format specifier will be replaced

So the o/p will be

3+2 equals 5 and 3-2 equals 1

Here %d is format specifier

Format Specifiers:

Char – %c
int – %d
unsigned int – %u
float – %f
Hexadecimal – %x
address – %p
String – %s

Now we will look into signed vs unsigned Arthematic as these will help in understanding the pointers concept better

Unsigned vs Signed :

→ In unsigned we have only +ve values

→ In signed we have both +ve and -ve values

* unsigned are generally represented in normal binary format
* Signed are generally represented in 2's complement form

Now How to compute 2's complement ?

there are two steps involved

① Complement every bit of the number

i.e $0 \rightarrow 1$
$1 \rightarrow 0$

② Add 1 to the resulting number

→ Find the 2's complement of 10100

Soln: ① 10100 → 01011

② 01011
+ 1
01100

so 2's complement of 10100 = 01100
= 01100

Memory representation of numbers :

→ int a = 4;

when we declare 'a' as int by default it is signed integer

So 4 = 0100

↑ This bit important as it represent sign

i.e if MSB = 1 then number is -ve
if MSB = 0 then number is +ve

we know int is 4 Bytes = 32 bits

So 0000 0000 0000 0000 0000 0000 0000 0100

→ int a = -4

-4 is to be represented in 2's complement form

so ① Assume that the given number is +ve and compute the binary format

i.e 4 = 0100

② Compute the 2's complement on the binaray number

i.e ① 0100 → 1011

② 1011
+ 1
1100

'a' is stored in main memory as

1111 1111 1111 1111 1111 1111 1111 1100

* if MSB = 1 then we copy '1' to the remaining 28 bits so it will be as shown above and same goes for MSB = 0 too

→ Signed int a = 4;

Here we dont care about 'a' i.e a is declared as signed or unsigned

we have a = 4 and 4 is +ve so we represent it in normal binary format.

→ unsigned int a = -4;

Here a is declared as unsigned but we dont care about it and we still represent -4 in 2's complement form

This is called fixed procedure. it means the procedure of storing the value in memory is fixed irrespective of unsigned or signed.

*** in 2's complement system

① if the number is +ve then we represent the number in normal binary format

② if the number is -ve then we represent the number in 2's complement form

* in 2's complement system

MSB will indicate the sign of the number

i.e if MSB = 0 it is a +ve number

if MSB = 1 it is a -ve number

→ Represent 5 in 2's complement system

Soln: Since 5 is +ve number we represent it in usual binary format

So 5 = 0101

→ Represent -5 in 2's complement form

Soln: Since -5 is -ve number

-5 = ① 0101 → 1010

② 1010
+1
1011

2's complement of -5 is 1011

* Similarly unsigned int a = -5, Here -5 is -ve and we dont care about unsigned or signed -5 will be stored in memory as 1011

Gate - 2019

→ in 16 bit 2's complement representation the decimal number -28 is

A) 1111 1111 0001 1100 B) 0000 0000 1110 0100

C) 1111 1111 1110 0100 D) 1000 0000 1110 0100

Soln: -28 = ① 28 in binary $= 16 + 4 + 8$

$= 16 + 8 + 4$

$= 011100$

$\downarrow$

100011

② 100011

$+1$

100100

in the remaining 10 bits 1 will be copied

So 1111 1111 1110 0100 ⇒ option c is true

Gate CSE 2000:

→ the number 43 in 2's complement representation is

A) 01010101 B) 11010101 C) 00101011 D) 10101011

Soln: 43 is +ve so we represent in normal binary form so MSB should be '0' so B, D can be eliminated

$43 = 32 + 8 + 2 + 1$

$= 0101011 \approx 00101011$ because it is +ve so we can copy MSB

Gate CSE 2002:

→ the 2's complement representation of the decimal value -15 is

A) 1111 B) 11111 C) 111111 D) 10001

Soln: -15 = ① 15 → 01111 → 10000

② 10000

$+1$

10001

Conversion of 2's complement to Decimal:

There are two methods for converting 2's complement to decimal

method-1 :

we assign -ve weight for MSB

i.e $0101 = -0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

$= 4+0+1 = 5$

$1011 = -1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$

$= -8+2+1 = -8+3 = -5$

method-2 :

① if the given number is +ve (MSB = 0) convert the number into decimal format normally

② if the number is -ve (MSB = 1), take the 2's complement of the number and then convert it to decimal

i.e 0101 → +ve number

$0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

$= 0+4+0+1 = 5$

1011 → -ve number

2's complement → 0100
+ 1
0101

= 5 but since MSB = 1 for the given number

so -5

* when we are given a representation in bit form say 1010 then we are asked to find the corresponding decimal value them

we must ask back wether the given bit form 1010 is signed or unsigned

① if unsigned then

1010 → $1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 = 8+2$
$= 10$

② if signed then

1010 → $-1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 = -8+2$
$= -6$

Copying number to higher bit representation : Lec-2

① copying unsigned numbers (zero extension)

Assume that 8 bit number to be copied to 16 bits

$$10110101 \rightarrow \underbrace{0000\ 0000}_{\text{This is called zero extension}}\ 1011\ 0101$$

So for the unsigned number, copy the LSB 8 bits as it is and Append zeroes in the front as shown in the above

→ Given $n = 5 \rightarrow 101$ copy to 4 bits

Soln: Since 5 is +ve and there wont be any difference in 2's complement and normal binary representation for +ve number $101 \rightarrow 0101$

② Copying signed number (sign extension)

in this we need to copy the sign bit (0 or 1) as it is

i.e if 8 bit representation of some integer is 1001 0110 then 9 bit representation of same integer is 1 1001 0110

like wise

0101 is 4 bit representation of some integer then 5 bit representation is 00101

So

4-bit	8-bit	16-bit	Decimal
0111	0000 0111	0000 0000 0000 0111	+7
1110	1111 1110	1111 1111 1111 1110	-2

* in the case of signed or unsigned integer representation we can remove excessive 0's at MSB as they are

not really useful

* In case of sign representation, atleast keep one zero so that we can identify that the number is +ve

i.e 0000 0011

↑ This '0' cant be ignored in the cased of signed representation as removing this zero the number becomes -ve

in case of unsigned int we can remove this zero too

* In the case of signed representation we can remove excessive 1's too and remember that atleast one '1' should be kept as it will represent the sign of the number

i.e 1111 1100

↑ This '1' cant be removed

Gate CSE 2002:

→ Sign extension is a step in

A) Floating point multiplication B) signed 16 bit addition

c) Arthematic left shift d) Converting signed extension from one size to another

Soln: we know sign extension is required to copy lower bits to higher bits (8 bits → 16 or 32 bits) in case of signed numbers.

If it is unsigned representation then we have zero extension so d is true

Now Comming back to c-programming

→ What are the values that printf statement will print for the following code

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

```
int main () {
  int x = 9;
unsigned int y = 9;
  int p = -9;
unsigned int q = -9;
  printf ("%x", x);
  printf ("%x", y);
  printf ("%x", p);
  printf ("%x", q);
}
```

Soln: we know that irrespective of what the representation is i.e signed or unsigned 9 will be stored in the memory as 0000 0000 0000 0000 0000 0000 0000 1001

and -9 will be stored as

-9 = ① 9 → 01001 → 10110

② 10110
+ 1
10111

So 1111 1111 1111 1111 1111 1111 1111 0111

* The intresting case is

if p = -9 and printf ("%d", p); will give output as -9

but printf ("%u", p); will give $2^{32} - 8 - 1$ as o/p

Similarly

unsigned int p = -9;

printf ("%d", p) will give -9 as output

But

printf ("%u", p) will give $2^{32} - 1 - 8$ as o/p

Now comming back to the question

Here the binary values of 9, -9 are fixed irrespective of signed or unsigned int and when printing the value of o/p depends on the format specifier

printf ("%x", x) will give 0 0 0 0 0 0 0 9

printf ("%x", y) will give 0 0 0 0 0 0 0 9

printf ("%x", p) will give F F F F F F F 7

printf ("%x", q) will give F F F F F F F 7

** the value of binary format depends on signed unsigned only when it is a -ve number

i.e 0101 → signed → 5, unsigned → 5 — decimal value doesnt change

1101 → signed → -3, unsigned → 13 — here decimal values of signed, unsigned changes as 1101 is -ve in the signed representation.

Zero extension and Sign extension:

Extension concept is very important

it is of 4 types

1. unsigned to unsigned
2. unsigned to signed
3. signed to signed
4. signed to unsigned.

1. Unsigned to unsigned:

```
unsigned short int a = 9;
unsigned int B = A;
unsigned short int C = -9;
unsigned int D = C;
```

we know short int has 16 bits or 2 Bytes and int has 32 bits or 4 Bytes

whatever the LHS be 9 will be stored in the memory as 0000 0000 0000 1001

now we are extending it to 32 bits by Assigning A to B

* Here we dont care wether B is signed or unsigned. we call 'a' as source

if source is unsigned we fill the extra bits with '0' and if source is signed then we fill the extra bits with sign bit

Since 'a' is unsigned

B = 0000 0000 0000 0000 0000 0000 0000 1001

Now -9 will be stored in memory as

-9 : 9 = 01001 → 10110
+1
10111

1111 1111 1111 0111

So C = 1111 1111 1111 0111

we are assigning c to D and source 'c' is unsigned so we append zero's

i.e D = 0000 0000 0000 0000 1111 1111 1111 0111

② unsigned to signed

```
unsigned short int A = -9;
        signed int B = A;
```

we dont care about signed or unsigned of 'A'

-9 = 1111 1111 1111 0111 = A

since source is unsigned

B = 0000 0000 0000 0000 1111 1111 1111 0111

③ signed to signed

```
Short int A = -9;
        int B = A;
```

So A = 1111 1111 1111 0111

Here source is signed so we use sign bit for filling the extra bits

So B = 1111 1111 1111 1111 1111 1111 1111 0111

④ signed to unsigned

```
Short int A = -9;
unsigned int B = A;
```

So A = 1111 1111 1111 0111

Since source is signed we use sign bit extension

So B = 1111 1111 1111 1111 1111 1111 1111 0111

** Extension depends on LHS of the source

i.e if LHS of source is

Signed : use signed extension i.e copy MSB (0 or 1) to fill new space

unsigned : use zero extension

Integer Promotion :

→ whenever a small integer type (char or short) is used in an expression then it will be simply converted to int. This is called integer promotion

it is because

Let char c = 'a' [a will be stored in the memory as ascii value]

i.e c can be stored as int and we can perform operations like (c+1, c-1, c*5)

→ now printf ("%d", c) will Print the ascii value of a

printf ("%c", c) will print a

Since c+1 → integer we need to promote char to int for evaluating the expression c+1 so here is the reason for integer promotion

→ Given Short int y = -9;

unsigned int iy = y;

```
Printf ("%d", y);
Printf ("%u", y);
Printf ("%d", iy);
Printf ("%u", iy);
```

Soln: normally -9 = 1111 1111 1111 0111 = y

Since source is signed we use sign extension

iy = 1111 1111 1111 1111 1111 1111 1111 0111

Here comes the intresting part

Printf ("%d", y);

Here y is treated as an expression and y is short int so it will be converted to int due to integer promotion

so y = 16 bits ⟶ 32 bits

Since y is signed we copy the sign bit

so y = 1111 1111 1111 1111 1111 1111 1111 0111

%d is treated as signed

so printf ("%d", y) = -9

Printf ("%u", y);

Here most of the students makes a mistake that they directly take y as short and %u is used to treat y as unsigned so the output will be $2^{16} - 8 - 1$

But it is "WRONG"

Here y is an Expression in printf so using integer promotion y is converted to int and y = 1111 1111 1111 1111 1111 1111 1111 0111

so printf ("%u", y) = $2^{32} - 8 - 1$

and similarly

Printf ("%d", iy) = -9
Printf ("%d", y) = -9 } both are same due to sign bit extension
Printf ("%u", iy) = $2^{32} - 8 - 1$

→ unsigned s
int iy = y
Printf ("
printf ("
printf ("
printf (

Soln: printf

it is

exten
so
so pri
Similarl
Print
Print
Print

→ int x =
unsigned
Printf (
Printf (
Printf (
Printf (

Soln: unsi
i
comp
Here
both
Printf

→ unsigned short int y = -9;

```
int iy = y;
printf ("%d", y);
printf ("%u", y);
printf ("%d", iy);
printf ("%u", iy);
```

Soln: printf ("%d", y);

y is an expression and y is short so it is converted to int

Since it is unsigned we use zero extension

So y = 0000 0000 0000 0000 1111 1111 1111 0111

So printf ("%d", y); = $2^{16}-8-1$

Similarly

printf ("%u", y); = $2^{16}-8-1$

printf ("%d", iy); = $2^{16}-8-1$

printf ("%u", iy); = $2^{16}-8-1$

→ int x = -1;

```
unsigned u = x;
printf ("%d", x);
printf ("%u", x);
printf ("%d", u);
printf ("%u", u);
```

Soln: unsigned u == unsigned int u

i.e Even if we dont declare u as int explicitly compiler will take it as int implicitly

Here Extension concept is not required as both are integers

printf ("%d", x); = -1

printf("%u", x); it will print the value as follows

$-1 = 1 : 0001 \rightarrow 1110$ in 32 bits

$$\begin{array}{r} 1110 \\ +\ 1 \\ \hline 1111 \end{array}$$

1111 1111 1111 1111 1111 1111 1111 1111

So o/p will be $2^{32}-1$

printf("%d", u); will print -1

printf("%u", u); will print $2^{32}-1$

→ what will be the min and max number in the unsigned form with K bits

Soln: Assume $K=4$

we get min value when 0 0 0 0

so min value $=0$

we get max value when 1 1 1 1

so max value $= 2^4 - 1$

So in unsigned form using K bits

min value $= 0$

max value $= 2^K - 1$

→ what will be the min and max number in the signed form with K bits

Soln: Assume $K=4$

we get min value when $(K-1)$ bits are 0's and MSB $= 1$ i.e 1 0 0 0

So min value $= -1 \times 2^3 + 0$

$= -8$

$= -2^{K-1}$

we get max value when MSB $=0$ and the remaining $(K-1)$ bits are 1's

i.e 0 1 1 1

So max value $= -0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

$= 4+2+1$

$= 7$

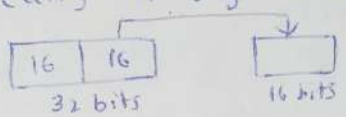
$= 2^{k-1}-1$

So the range will be $[-2^{k-1}$ to $2^{k-1}-1]$

Troncation :

→ Troncation is nothing but moving the higher bits to lower bits i.e 32 bits → 16 bits

→ Regardless of Source or destination, signed or unsigned type, turncation always truncates so this can cause the number to change drastically in sign and value



→ what will the printf prints

```
int x = -3;
unsigned short u = x;
printf("%u", u);
```

Solⁿ: -3 will be stored in memory as

$-3 = 3 : 0011 \rightarrow 1100$
$\quad\quad\quad\quad\quad\quad\quad\quad + 1$
$\quad\quad\quad\quad\quad\quad\quad\quad 1101$

in 32 bits

1111 1111 1111 1111 1111 1111 1111 1101

now we are truncating 32 bits of data to 16 bits so

u = 1111 1111 1111 1101

* in truncation we dont care about signed unsigned representation, we just take 16 bits from LSB and put it in 'u'

now %u is unsigned so

output will be $2^{16}-1-2$

→ what will the printf statement prints

```
char a=30, b=40;
char d = a*b;
Printf ("%d", d);
Printf ("%d", a*b);
```

Soln: $a*b = 30 \times 40$

$= 1200$

1200 in binary format is

0000 0100 1011 0000

it should be stored in d which is a character

So 8 bits from LSB is stored in d

d = 1011 0000

now printf ("%d", d);

Here d is an expression

So we use integer promotion and d is converted to int

So d : 1111 1111 1111 1111 1111 1111 1011 0000

the printf will print -80

Printf ("%d", a*b); will print 1200

→ what will the printf statements print

```
char f = -65;
Printf ("%d", f);
Printf ("%u", f);
```

Soln: $-65 = 65 \rightarrow 64+1 \rightarrow 01000001 \rightarrow 10111110$

$$\begin{array}{r} 10111110 \\ +\ 1 \\ \hline 10111111 \end{array}$$

now printf ("%d", f);

Here f is an expression so we integer promotion.

So f = 1111 1111 1111 1111 1111 1111 1011 1111

%d prints signed value so -65 will be printed

printf ("%u", f);

%u prints unsigned value of f which is

$2^{32} - 64 - 1 = 2^{32} - 65$

Type Conversion: Lec-3

→ Let us have a situation where we have int a, float b and we do a+b

Since a, b are involved in same expression there will be type mismatch so rule is that if we have int, float in an expression then the int is converted to float and final result will also be a float

i.e int a;
float b;
a + b;

int → float, float → float — final result → float

* the hierarchy of type conversion is as follows

↑ long double
double
float
unsigned long int
long int
unsigned int
int
short, char

→ if we have long int, float then long int is converted to float and final result will also be in the float

now printf ("%d", f);

Here f is an expression so we integer promotion

so f = 1111 1111 1111 1111 1111 1111 1011 1111

%d prints signed value so -65 will be printed

printf ("%u", f);

%u prints unsigned value of f which is

$2^{32} - 64 - 1 = 2^{32} - 65$

Type Conversion: Lec-3

→ Let us have a situation where we have int a, float b and we do a+b

Since a, b are involved in same expression there will be type mismatch so rule is that if we have int, float in an expression then the int is converted to float and final result will also be a float

i.e
```
int a;
float b;
a + b;
↓   ↓
int float ] - final result → float
  → float -
```

* the hierarchy of type conversion is as follows

↑ long double
double
float
unsigned long int
long int
unsigned int
int
short, char

→ if we have long int, float then long int is converted to float and final result will also be in the float

→ what are the values that print statements will print

```
int value1 = 10, value2 = 3;
float result;
result = value1/value2;
Printf ("%f", result);
result = (float) value1/value2;
Printf ("%f", result);
result = value1/(float) value2;
printf ("%f", result);
```

Sol^n: the expression value1/value2 is called integer expression as both value1, value2 are integers

So value1/value2 = $\frac{10}{3}$ = 3.33 but since it is integer division we store '3' only

now we assign this value to result which is float

So result = 3.0

Printf ("%f", result) → 3.0

result = $\underbrace{\text{(float) value1}}$ / value2

it is called type casting which means we are converting value1 to float

now value1 = float
value2 = int so value2 is also converted to float

So result = 10.0/3.0 = 3.33

Printf ("%f", result) → 3.33

Same goes for result = value1/(float) value2
= 10.0/3.0 = 3.33

Printf ("%f", result) → 3.33

*→ Now we will see some intresting facts about the floating point numbers

we know int is 4B long so we can represent the values from $[0 \text{ to } 2^{32}-1]$ Assume only the unsigned int

Similarly float is also 4B long

Assume we represent the values $0.0, 1.0, 2.0, \ldots (2^{32}-1).0$

now the question is how can we represent say (1732.4434) likewise numbers using float as all the $(2^{32}-1)$ representations to be precise 2^{32} representations

The explanation is we dont represent every value using float i.e say 1939 is not represented but 1938.999679 may be represented

So if we want 1939 then compile / os will roundoff 1938.999679 to 1939.0 but if we print the 1939.0 to 10 decimal points then it will not print (1939.0000000000) it will print 1938.9996790000

The conclusion is that for every integer we dont have a float representation and this is because we can have infinite number of floating point numbers and we cant represent them in 2^{32} representations

* we can print a float with 10 decimals using

printf ("%.10f", a)

↳ it can print 10 decimal point float

→ what will the printf statements print

```
int main() {
    unsigned int a = 1000;
        int b = -1;
    if (a > b) printf("a is BIG");
    else printf("a is SMALL");
    return 0;
}
```

Sol^n: if (a > b)

↳ it is an expression and a, b are different data types

So b is converted to unsigned int

$-1 = 1 : 0001 \rightarrow 1110$
$\quad\quad\quad\quad\quad\quad\quad\quad + 1$
$\quad\quad\quad\quad\quad\quad\quad\quad 1111$

since int is 32 bit : b = 1111 1111 1111 1111 1111 1111 1111 1111
$= 2^{32} - 1$ (decimal value as b: signed int → unsigned int)

now $a = 1000$
$b = 2^{32} - 1$ so $a > b$ is false

a is SMALL will be printed.

IF statement:

The structure of if statement will be

```
if (condition)
{
    statement_block;
}
```

→ what will the printf statement print

```
#include <stdio.h>
int main ( )
{
    int grade;
    scanf ("%d", &grade);
    if (grade <= 10)
    {
        printf ("You did not study.\n");
        printf ("you like this course!\n");
    }
}
```

Soln: when we write scanf it means we are waiting for the user to provide the input

Assume the user provided '1'

Since $1 < 10$ so we goto inner statements in the if condition

So both printf statements gets executed in the above given case

**

```
if ( grade <= 10)
  printf ("Easy");
  printf ("But tricky");
```

=

```
if (grade <= 10)
{
  printf ("Easy");
}
printf ("But tricky");
```

→ what printf statements does the following program print

```
#include <stdio.h>
int main ( )
{
  if (0)
  printf ("statement 1 \n");
  printf ("statement 2 \n");
  printf ("statement 3 \n");
  return 0;
}
```

Sol^n: in general in if statement the condition is evaluated as true or false

if (← condition →)

but we can directly give true or false to if statement

i.e if (true), if (false)

in programming

true → non zero

false → zero

Since if (false)

so statement 1 wont get printed

** if (-1)

↳ is treated as true

i.e anything other than '0' will be treated as true in the if statement

i.e if (3-3) or if (x=3-3) then they are treated as false as they return '0'

else statement / clause :

→ what will the printf statement print

```
#include <stdio.h>
int main () {
  int i = 20;
  if ( i < 15) {
    Printf (" i is smaller than 15");
  }
  else {
    printf (" i is greater than 15");
  }
  return 0;
}
```

Sol^n: we have initialized i to 20

and if (i < 15) is false as 20 > 15

So we goto else statement

* we only goto else statement only when the if statement is false.

So i is greater than 15 will be printed

Nesting if ... else :

if we donot use braces then consider only one statement inside if statement

↳ the statement can be simple statement i.e one line

(or)

the statement can be compound statement

i.e if (num > 0)

```
[ if (num < 10)
    printf (" num is between 0 and 10");]
```

↳ This is a compound statement

→ we can have

```
if (num
{
  print
}
else
  if (
    {
    }
```

Matching if ...

```
if
```

and only

else state

i.e

```
if (a < 10)
  if (a
    Prin
else
  Printf
```

* *

matter

So if o

Print ar

inner i

→ what will

```
int i=1,
if (i > j
  if
el
```

→ we can have else if statements also

```
if (num > 90)
{
    printf (" You earned an A");
}
else
    if (num > 80)
    {
        printf (" You earned B");
    }
```

Matching if... else parts :

if we have more than one if statements and only one else statement then match the else statement to the nearest if statement.

i.e

```
if (a < 10)
    if (a % 2 == 0)
        printf (" statement-1"); ==
else
    printf (" statement-2");
```

```
if (a<10)
  if (a%2==0)
    printf (" stmnt-1");
  else
    printf ("stmnt-2");
```

** In C-programming indentation doesnt matter

So if a = 11 then the above program doesnt print anything as the else is matched with the inner if statement.

→ what will the printf statement print

```
int i=1, j=5, k=4, number;
if (i > j)
    if (i > k)
        number = i;
else
```

```
        number = k;
    else
        if (j > k)
            number = j;
        else
            number = k;
    Printf ("%d", number);
```

Soln:

Since $i > j$ is false as $1 < 5$

we golo else

now $j > k$ is true as $5 > 4$

So number = 5

Switch statement:

The structure of the switch statement will be

```
Switch (expression)
{
    Case Value-1:
                Block-1;
                Break;
    Case value-2:
                Block-2;
                Break;
    Case Value-n:
                Block-n;
                Break;
    default:
                Block-1;
                Break;
}
```

* Here Break keyword tells us to come out of the Switch statement

→ what will the p

```
# include <
int main
switch (
    C
    }
```

Soln: Since n

Va

Rules for us

→ The

Swi

mos

express

or an

→ the ex

a) floa

Soln: we

an

→ what will the printf statement prints

```
#include <stdio.h>
int main ( ) { int make, num = 8;
switch (num) {
        case 7:
            printf (" value is 7");
            break;
        case 8 :
            printf (" value is 8 ");
            break;
        case 9 :
            printf (" value is 9");
            break;
        default :
            printf (" out of range");
            break;
    }
    return 0;
}
```

Sol^n: Since num = 8 we goto case 8 which will print value is 8

Rules for using switch statement :

→ The expression after switch keyword ie switch (expression) must yield an integer value ie the expression should be an integer or a variable or an expression that evaluates to an integer

→ the expression in switch (expression) can take
a) float b) int c) char d) short (MSQ)

Sol^n: we know char internally is an integer and also char, short are called small integers so b, c, d can be the answer

-> The case statements and the default statement can occur in any order in the switch

-> the case label values must be unique

i.e switch ( ) {

case 1 :

case 2 :

case 2 :

}

it is not valid as case 2 is repeated
↳ label

-> the case label must end with a colon (:)

i.e case 1 :

-> what will be the output of the program

```
#include <stdio.h>
int main ( )
{
    int num = 2;
    Switch (num +2)
    {
        printf (" hey");
        Case 1 :
            Printf (" case - 1");
        case 2 :
            Printf (" case - 2");
        case 3 :
            Printf (" case - 3");
        default :
            Printf (" Default : value is %d", num);
    }
    return 0;
}
```

Soln : * Here printf (" hey"); doesnt belong to any case statements or default statement so it wont get executed and Switch (num +2) = Switch (4)

Assume num = 1

then num + 2 = 3

So switch (3)

Now since there is no break at the end of case 3 the switch statement will print default value also

i.e (case 3 : value is 3
Default : value is 3) both of them gets printed

* This is called fall through

Fall through:

in switch statement if we dont have a break at the end of currently executing case then it will execute till it sees break and this condition is called fall through.

→ what will be the output

```
int month = 2;
int year = 2000;
int numdays = 0;
switch (month) {
  case month :
        printf ("%d", numdays);
}
```

Soln: * in case label :
↳ label should not be a variable
it should be integer constant
So we get error

Gate CSE 2012 :

→ what will be the output of the following C program segment

```
char inchar = 'A';
    switch (inchar) {
        case 'A': printf ("choice A \n");
        case 'B':
        case 'C': printf ("choice B");
        case 'D':
        case 'E':
        default : printf ("No choice");
    }
```

A) No choice B) choice A C) choice A
choice B No choice

D) Program donot give any output because it is erroneous.

Soln: This is fall through

So the output will be Choice A
choice B Nochoice

-> what will be the output

```
** int value = 0;
    switch (value) {
        default:
            value ++;
        case 2 :
            Printf (" Humans are human centric");
            break;
        case 3 :
            Printf (" This is inhuman");
    }
    Printf ("%d", value);
```

Soln: Here value = 0

Since there is no case 0 we goto default and there is no break in default so we execute case 2 and there is break in case 2 so o/p will be : Humans are Human centric !

* if default is
then o/p will
the logic
we goto default
since there is
till we see th

Loops in c:

while loop :
The st
while (co
statemen
repeate
}

eg: while
{
su
co
}

-> int s = 0;
int a;
scanf ("%d
while (!(
s = s + a;
scanf ("
}

Soln: the wh
gives
!(tr

For loop :
Th

* if default is between case 2 and case 3 then then o/p will be : this is inhuman.

the logic here is since there is no case 0 we goto default and start executing from that since there is no break in default we will execute till we see the break

Loops in c :

while loop :

The structure of the while loop is

```
while (condition) {
  statements to be executed
  repeatedly
}
```

eg:

```
while (count != 10)
{
  sum = sum + count;
  count = count + 1;
}
```

```
-> int s = 0;
   int a;
   scanf("%d", &a);
   while (!(a==-1)) {
   s = s + a;
   scanf("%d", &a);
   }
```

Soln: the while loop will terminate only when the user gives -1 as input because -1 == -1 is true and !(true) = false so we come out of while loop

For loop :

The structure of the for loop is

```
for (initialization ; condition ; increment) {
      statements to be executed repeatedly
}
```

eg: for (i=0 ; i<5 ; i++)

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

the flow of execution of the for loop is

① we initialize i=0

this step only occurs once ie we initialize i=0 only once in the entire loop

② we check wether i is less than 5

if i<5 then we go to the body of the for loop which is printf and execute

③ we increment i i.e i++ ≈ i=i+1

now i=1, again we check that i<5 or not

if i<5

we again print the value of i

else

we come out of for loop

②, ③ will repeat as long as for loop is executing

** for loop has three parts

for (①, ②, ③)

{

}

① = initialization

② = condition

③ = increment

we can skip any of the 3 parts in the for loop i.e for (; ②; ③) is valid

for (①; ; ③) is valid

→ what will be the output of the following code snippet

```
for( P = 10 ; P > 0;)
{
    P = P-1;
}
printf ("%d", P);
```

Soln: we can write the code snippet as

```
int p;
for ( p=10 ; p>0 ;)
  p = p-1;
printf ("%d", p);
```

the value of p that will be printed is '0'

→ what will be the output of the following code snippet

```
# include <stdio.h>
Void main ( )
{
  int i=0;
  for ( ; i<=9; )
  {
    i++;
    Printf ("\n value is : %d", i);
  }
}
```

Soln: ** Even if ①, ③ parts are missing in the for loop the flow of execution doesnt change so the o/p will be

1, 2, 3, 4, 5, 6, 7, 8, 9, 10 as we are printing the incremented value

→ what will be the output of the following code snippet

```
# include <stdio.h>
int main ( )
{
  int i;
  for ( i=0 ; i<=3 ; i++);
  Printf ("%d", i);
  return 0;
}
```

A) Compilation error B) 1, 2, 3 C) 4 D) 0, 1, 2, 3

Soln: Here we have semicolon at the end of for loop

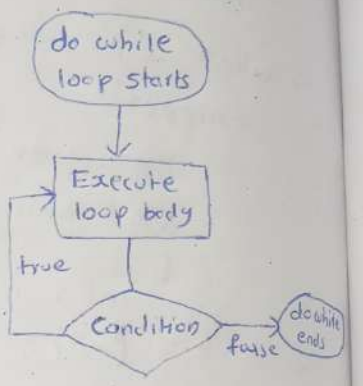
So for (i=0; i<=3; i++); == for (i=0; i<=3; i++)
{
}

the o/p will be '4'

Do while loop:

the structure of the do while loop is

```
do
{
  statements to be
  executed repeatedly
} while (condition);
```



* in do while loop we execute the statements in the loop without checking the condition for the first time then after executing the statements we then check the condition

eg:
```
# include <stdio.h>
main ()
{
  int n=0, m=1;
  do
  {
    printf ("%d", m);
    m++;
  } while (m<=n);
}
```

Here from the beginning m > n but m is printed for one time and next we check the condition

Break statement:

** when we execute the Break statement which is present inside the loop then we immediately come outside of the loop

i.e
```
int a;
while (1) {
  scanf ("%d", &a);
  if (a==0) break; }
```

while(1) → Always true and this loop will terminate only when user gives '0' as input

⇒ what will be the output of the following code snippet.

```
#include <stdio.h>
int main (void) {
    int hours = 10;
    int i = 0;
    for (i=0; i< hours; i++) {
        if (i==5) {
            break;
        }
        printf ("%d", i);
    }
    return 0;
}
```

Soln: the for loop breaks when i==5 so until i=5 we print every value of i so 0,1,2,3,4 will be printed.

Continue statement:

when we execute the continue statement then we continue executing the loop for the next iteration without executing the statements below the continue statement

i.e

```
for (i=0; i<10; i++) {
    if (i==5)
        continue;
    Printf ("%d", i);
}
```

Here 0,1,2,3,4 are printed and when i=5 we execute continue statement so we dont execute printf which is below the continue statement and executes the next iteration of the for loop so o/p will be 0,1,2,3,4,6,7,8,9

TIFR 2017

→ Consider the following pseudocode fragment where y is an integer that has been initialized

```
int i = 1
int j = 1
while (i<10):
    j = j * i
    i = i + 1
    if (i==y):
        break
    end if
end while
```

Consider the following statements

I) (i==10) or (i==y)

II) if (y>10) then i==10

III) if j=6 then y==4

which of the following statements above is/are true at the end of while loop? choose from the following options:

A) I only B) III only C) II and III only D) I, II, III E) None of the above

Soln: we get out of while loop when

i>=10 or i==y

So in I) i==10 or i==y is true

in 2) if y>10 say y=11 then while loop only breaks when i==10

So 2 is true

in 3) j=6 i.e

i=1	i=2	i=3
j=1×1	j=2×1	j=3×2
=1	=2	=6
i=i+1	i=i+1	i=i+1
=2	=3	=4

when j=6, i=4 so y==4 as we get j=6 when we break at this point

so 3 is true

Operators:

Lec-4

→ we will see the operators in C-programming

Some of the operators like "&" (ampersand) will be discussed while studying pointers.

In general operators are divided into

1. Relational operators
2. logical operators
3. Assignment operators

Relational operators :

These operators tells the relation between two variables

i.e $<, <=, >, >=, ==, !=$ etc

* if (m=n)

↳ Here what this expression will return is we assign the value of n to m and then the value of n gets returned

if (m=n) → will be false only when n==0

Logical operators :

"&&" is logical AND , "&" is bitwise AND

"||" is logical OR , "|" is bitwise OR

"!" is logical NOT.

** The relative precedence of the relational and logical operators are

Highest → ! (NOT)

>, >=, <, <

==, !=

&&

Lowest → || (Logical OR)

Assignment operators :

→ we can represent m = m+1 in short hand form as m+=1

* if we have x += y+1 then it will be evaluated as x = x+(y+1)

and not as $x = (x+y)+1$, it may not make much difference if we have addition but if we have $x * = y+1$ then $x = x * (y+1) \neq x = (x*y)+1$

Similarly

$\rightarrow a = a+1 \Rightarrow a+=1$

$\rightarrow a = a-1 \Rightarrow a-=1$

$\rightarrow a = a*(n+1) \Rightarrow a*=n+1$

$\rightarrow a = a/(n+1) \Rightarrow a/=n+1$

$\rightarrow a = a\%b \Rightarrow a\%=b$

Increment and Decrement operators:

++m, --m, m++, m--

++m is equivalent to m = m+1 or m+=1

--m is equivalent to m = m-1 or m-=1

→ ++m = pre Increment of m

m++ = Post Increment of m

→ --m = pre decrement of m

m-- = Post decrement of m

Given n = 5

x = n++

Here we first assign n to x

So x = 5

now we increment n

So n = 6

x = ++n

Here we first increment value of n

So n = 6

now we assign n to x

So x = 6

* we cannot pre increment or postincrement Constants i.e (++2) → it gives error as we can write

$++a = (a = a+1)$ like wise

$++2 = (2 = 2+1)$ is meaningless

* we Cannot preincrement or post increment the expressions

i.e $\underbrace{++(a+b*2)}$ let $a=1, b=2$

↳ it will result in error because

Expressions dont have dedicated memory

i.e $(++a)$ Here a is variable and it has dedicated memory

the result of expression will be stored in a temporary register

→ In operators we have precedence and the Associativity

Addition follows Associativity

i.e $(a+b)+c = a+(b+c)$

Division doesnt follow Associativity

i.e $(a/b)/c \neq a/(b/c)$

in general division follows left to right Associativity

i.e $a/b/c = ((a/b)/c)$

Similarly

the power operator follows right to left Associativity

i.e $a^{b^c} = \left(a^{(b^c)}\right)$ let $a=2, b=5, c=2$

$2^{5^2} = \left(2^{(5^2)}\right) = 2^{25}$

→ Left to right associativity is called left associativity

Right to left associativity is called Right associativity

Precedence of Arthematic operators :

* High priority : *, /, %
* Low priority : +, -

→ The basic evaluation procedure follows two left to right passes through the expression

- During the first pass the high priority operators are evaluated
- During the second pass the low priority operators are evaluated

→ Arthematic operators are left associative

Let

$a = 9,\ b = 12,\ c = 3$

$x = a - b/3 + c*2 - 1$

Here /, * are high priority operators

So $x = 9 - 12/3 + 3*2 - 1$

in first pass :

$x = 9 - 4 + 6 - 1$ [$12/3 = 4$, $3*2 = 6$ are evaluated]

in second pass :

$x = 5 + 5$

$= 10$

Gate CSE 2015 :

→ Consider the following c program

```
#include <stdio.h>
int main ( )
{
  int i, j, k=0;
  j = 2*3/4 + 2.0/5 + 8/5;
  k - = --j;
```

```
for (i=0; i<5; i++)
{
    switch (i+k)
    {
        case 1:
        case 2: printf ("\n %d", i+k);
        case 3: printf ("\n %d", i+k);
        default: printf ("\n %d", i+k);
    }
}
return 0;
}
```

The number of times printf statement is executed is —

Soln: Given $j = 2 * 3/4 + \underbrace{2.0/5}_{\text{float division}} + \underbrace{8/5}_{\text{integer division}}$

$j = 6/4 + \underbrace{2.0/5}_{\text{it is converted as } 2.0/5.0 = 0.4} + 8/5$

$= 0.4$

So $j = 1 + 0.4 + 1$ * $\frac{6}{4} = 1.5$ but $\frac{6}{4}$ is int division so $\frac{6}{4} = 1$

$j = 2.4$ but since $j =$ integer

Similarly $\frac{8}{5}$

Since we have 0.4 so $1 \to 1.0$ so $1.0 + 0.4 + 1.0 = 2.4$ so $j = 2$

now $k - = --j$

$= k = k - --j$

$= 0 - (1)$

$k = -1$

when $i = 0$	$i = 1$	$i = 2$
Switch $(0-1)$ so -1 one printf	$i+k = 0$ one printf	$i+k = 1$ 3 printf
$i = 3$ $i+k = 2$ 3 printf	$i = 4$ $i+k = 3$ 2 printf	$i = 5$ for loop will terminate

no of times printf will be occuring will be

$1 + 1 + 3 + 3 + 2 = 10$

Bitwise operators :

Some of the Bitwise operators are

- → Bitwise AND (&)
- → Bitwise OR (|)
- → Bitwise EXOR (^)
- → Left shift (<<)
- → Right shift (>>)
- → One's Complement (~)

operand 1 (op1)	operand 2 (op2)	op1 & op2	op1 | op2	op1 ^ op2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

* EXOR operator is also called (mod 2) operator

Bitwise AND (&) :

```
      x → 0000 0000 0000 1101
      y → 0000 0000 0001 1001
x & y = z  z → 0000 0000 0000 1001
```

** '&' is a multipurpose operator, we have to use it based on the Context

→ one application of Bitwise AND is it is used to check wether the given number is even or odd i.e

```
int test = 1;
if (number & test)
    printf (" number is odd ");
else
    printf (" number is even");
```

and with unit place
⇒ 0 & 1 = 0 ⇒ Even
⇒ 1 & 1 = 1 ⇒ Odd.

Another way of chec
if (nomber
printf ("
else
printf (

→ How to check if 3
not

Soln: we can say 3

Bitwise OR (|) :
x → 00
y → 00
x|y → 0

Similarly x^y → 0
↳ Exclusiv

Left shift (<<) :
Let x i
now x <<
x by 3 Pla
at LSB
So x << 3 =

Right shift (>>
Let x
now x >>
to the ri
at LSB
So x >>

** Shift oper

Another way of checking number is even or odd is

```
if (number % 2 == 0)
    Printf ("Number is even");
else
    Printf ("Number is odd");
```

→ How to check if 3rd least significant bit is 1 or not

Soln: we can say 3rd LSB is '1' if number & 4 == 4

Bitwise OR (|):

$x \rightarrow$ 0000 0000 0000 1101

$y \rightarrow$ 0000 0000 0001 1001

$x|y \rightarrow$ 0000 0000 0001 1101

Similarly $x \wedge y \rightarrow$ 0000 0000 0001 0100

↳ Exclusive OR

Left shift (<<):

Let x is 0100 1001 1100 1011

now $x << 3$ means we shift the bits in x by 3 places to the left and append 0's at LSB

So $x << 3$ = 0100 1110 0101 1000

Right shift (>>):

Let x is 0100 1001 1100 1011

now $x >> 3$ means we shift every bit in x to the right side by 3 places and append 0's at LSB

So $x >> 3$ = 0000 1001 0011 1001

** Shift operators can be dangerous when it

Comes to signed numbers

i.e 1100 0111 = x

now $x >> 1$ = 0110 0011

Here x is -ve number and $x >> 1$ becomes positive number

→ There are two types of right shift operators

① unsigned Right shift → fill MSB with 0's

② Signed Right shift → Depends on System fill MSB with 0s or sign bit

Signed right shift is also called arthematic right shift.

→
```
# include <stdio.h>
int main ( ) {
  int x, y=10;
  x = y <<1;
  Printf ("%d", x);
}
```

Soln: we know in binary representation left shift by one bit means multiplying the given number with '2'

So $x = 2 \times 10 = 20$

Similarly left shifting by 2 bits means multiplying the given number with 4

Right shifting by 1 bit means dividing the given number with '2'

i.e $x = y >> 1$ will give $x = 5$

One's Complement operator (~):

in 1's complement we change 1 → 0
0 → 1

x = 1001 0110

$\sim x$ = 0110 1001

→
```
int main ( ) {
  int x;
  x = 8; /* 000
  printf ("%d", ~
}
```

Soln: in the Questio

16 bits

0000 0000

in printf ("%d

as Signed num

Conditional operator

exp1 ?

what it

then we goto

is false then

i.e int m

{

int

a

Pri

re

}

Here

Gate CSE 2008:

→ which Combin

2 makes the

following ex

$x = 1001\ 0110\ 1100\ 1011$

$\sim x = 0110\ 1001\ 0011\ 0100$

```
int main () {
    int x;
    x = 8; /* 0000 0000 0000 1000 */
    printf ("%d", ~x);
}
```

Soln: in the Question they are saying take int as 16 bits

0000 0000 0000 1000 → 1111 1111 1111 0111

in printf ("%d", ~x) we see the binary format as signed number so it will print -9

Conditional operator or terenary operator:

exp 1 ? exp 2 : exp 3

what it means is if expression 1 is true then we goto expression 2 and if expression 1 is false then we goto expression 3

i.e
```
int main ()
{
    int a = 0;
    a = 5 < 2 ? 4 : 3;
    printf ("%d", a);
    return 0;
}
```

Here 5 < 2 is false so 3 is assigned to 'a'

Gate CSE 2008:

→ which combination of the integer variables x, y and z makes the variable 'a' get the value 4 in the following expression

$a = (x > y) ? ((x > z) ? x : z) : ((y > z) ? y : z)$;

A) $x=3, y=4, z=2$ B) $x=6, y=5, z=3$ C) $x=6, y=3, z=$

D) $x=5, y=4, z=5$

Soln: we go by option verification

Let $x=3, y=4, z=2$

$3 > 4$ is false so we goto

$y > z ? y : z$

So option 1 is true $4 > 2$ = true so 4 will be o/p

* if we observe carefully we can see that the tevenary operator is checking for the maximum value among x, y, z

Comma operator:

Let

value = $(x=10, y=5, x+y)$;

→ in comma operator we evaluate the expression from left to right and the value of the rightmost expression will be the value of the whole expression

So $x = 10$
$y = 5$
$x+y = 15$

finally value = 15

* one famous application of comma operator is we can perform swapping easily

i.e $t=x, x=y, y=t$;

→ # include <stdio.h>

** int main() {

int i, j = 2;

for (i=0; i<=5, j>=0; i++) {

Printf ("%d", i+j);

j--;

}

return 0;

}

Soln: in for loop

(i<=5,

expression

the fact th

j>=0 the

Also

will break

Now j

i=

i+

j

j>=0 fa

so o/p of

→ # include <

int main

int

for

{

}

re

}

Soln: Here

using

so i=0

i+j

```
    j--;
  }
  return 0;
}
```

Soln: in for loop condition checking part we have $(i<=5, j>=0)$ so the final output of this expression depends on $j>=0$ i.e we dont consider the fact that $i<=5$ it means even if $i>5$ and $j>=0$ the for loop will continue executing

Also if $\underset{T}{i<=5}$ and $\underset{F}{j>=0}$ then for loop will break

Now $j = 2 >= 0$

$i=0, j=2$	$i=1, j=1 >= 0$	$i=2\ j=0 >= 0$
$i+j = 2$	$i+j = 2$	$i+j = 2+0$
		$= 2$
$j-- = j = j-1$	$j-- = j = j-1$	$j-- = j = j-1$
$= 2-1$	$= 1-1$	$= 0-1$
$= 1$	$= 0$	$= -1$

$j>=0$ fails when $j=-1$

so o/p of the program is 222

→
```
#include <stdio.h>
int main() {
    int i, j=2;
    for (i=0; j>=0, i<=5; i++)
    {
        printf("%d", i+j);
        j--;
    }
    return 0;
}
```

Soln: Here we dont consider $j>=0$ condition as we are using a comma operator

so $i=0\ j=2$ $\quad$ $i=1, j=1$ $\quad$ $i=2, j=0$

$i+j = 0+2 = 2, j=1$ $\quad$ $i+j=2, j=0$ $\quad$ $i+j=2, j=-1$

$i=3, j=-1$	$i=4, j=-2$	$i=5, j=-3$	$i=6$ Then $i>5$ So the condition fails
$i+j = 3-1 = 2$	$i+j = 4-2 = 2$	$i+j = 5-3 = 2$	
$j=-2$	$j=-3$	$j=-4$	

So o/p will be 222222

```
-> #include <stdio.h>
   int main ( ) {
   int i, j = 2;
   for ( i=0, j=2 ; j >=0 && i<=5 ; i++, j--)
   {
      printf ("%d", i+j);
   }
```

Soln:

$i=0, j=2$	$i=1, j=1$	$i=2, j=0$	here $j>=0$ condition will fail
$i+j = 2$	$i+j = 2$	$i+j = 2$	
$i=1, j=1$	$i=2, j=0$	$i=3, j=-1$	

So the output will be 2,2,2

Highest and 2nd highest precedence operators:

-> These operators are very important when it comes to gate exam as many tricky Questions can be formed using these operators

Operators	Associativity
1) ( ), [ ], ->, . , ++, -- (Post fix)	left to right
2) sizeof, &, *, +, -, ~, ! typecast, ++, -- (Prefix)	right to left

Here & -> ampersand (address of)

* -> dereference operator (used in pointers)

+, - -> unary operators

~ -> 1's complement

Also, a++, a-- have high priority than ++a, --a

All operators with precedence and Associativity:

Now we will see all the operators with their precedence and Associativity

( )	Paranthesis
[ ]	Brackets
.	Dot or m
->	Arrow op
++, --	Post incr
++, --	Pre incre
+, -	unary
!, ~	NOT oper
(type)	typeca
*	Indirect
&	Addres
size of	Deter
*	Multi
/	divi
%	Mo
+, -	Additi
<<, >>	Bitw
<, <=	Rel
>, >=	Re
==, !=	Re
&	
^	
|	

s then i >5 so the
ndition fails

t, j--)

here j >=0
condition will
fail.

tors:
mpoitant when
tricky Questions
tors

Associativity
left to right
right to left
)
in pointers)

than ++a, --a
ativity:
ators with
Associativity

( )	Paranthesis or function call	
[ ]	Brackets or Array subscript	
.	Dot or member selection operator	Left to right
->	Arrow operator	
++, --	Post increment, Post decrement	
++ --	Pre increment, Pre decrement	
+, -	Unary plus, Unary minus	
!, ~	NOT operator, Bitwise complement	
(type)	Type cast	Right to Left
*	Indirection or dereference operator	
&	Address of operator	
size of	Determine size in Bytes	
*	Multiplication	left to right
/	division	
%	Modulo	
+, -	Addition and subtraction	left to right
<<, >>	Bitwise left shift and Right shift	left to right
<, <=	Relational less than, less than equal to	left to right
>, >=	Relational greater than greater than equal to	
==, !=	Relational equal to, not equal to	left to right
&	Bitwise AND	left to Right
^	Bitwise XOR	left to right
|	Bitwise OR	left to right

Short Circuiting in C :

```
#include <stdio.h>
int main() {
    int a=1, b=1, c=1;
    if (a==b || c++) {
        Printf("%d", c);
    }
}
```

Soln : we might be thinking that

in a==b || c++ post increment has the higher precedence

i.e c++ should be evaluated first

so c = 2

and a==b is true

so Printf will print '2'

to right

to right

ht to left

ght to Left

left to right
priority

nent has the
cedence
first

2'

But it is wrong because

say int a, b;
a = 3+5;
b = 3*5;

Here even though 3*5 has high priority than 3+5 we still execute a = 3+5 first

This is called the order of evaluation

Similarly
order of evaluation of && , || says that

**Always execute the left expression first and then execute the right expression

i.e expression 1 && expression 2 then expression 1 is evaluated first

→ exp 1 && exp 2 || exp 3

then the order of evaluation is

&& has high priority than || so

(exp 1 && exp 2) || exp 3

to be precise

((exp 1) && exp 2) || exp 3

→ exp 1 || exp 2 && exp 3

Since && has high precedence than ||

exp 1 || (exp 2 && exp 3)

Now the intresting point is

order of evaluation comes first than the precedence

So here exp 1 will be evaluated first

→ exp 1 && exp 2 || exp 3 && exp 4

Since && have high precedence and it is left to right associativity

so (exp 1 && exp 2) || (exp 3 && exp 4)

we know the order of evaluation has high priority than Precedence

So exp 1 will be evaluated first
exp 2 will be evaluated second
exp 3 will be evaluated third
exp 4 will be evaluated fourth

** we have a term called Compiler dependant (or) undefined

undefined means it is not defined by the language so the compiler has the liberty to choose which way it wants

i.e say a = i++ + i--

now (i--) here i should take the incremented value or the original value is not defined by the language

So the compiler takes the liberty and chooses whatever it wants

This is called Compiler dependant as Compiler is choosing which way to follow independent of the language

Now comming back to the Question

(a==b) || c++

According to the order of evaluation

(a==b) will be executed first

since a==b is true

true || anything = true

so second expression wont get evaluated at all

* True || we dont care = True , False || we care

* False && we dont care = False , True && we care

→ #include <stdio.h>
int main() {
int a=1, b=1, c=
if (a != b || c
printf ("%d",
}

Soln: Here According
a != b w
we goto c++
c=0 is retu
So if (0) =
the output.

Gate CSE 2021 :

→ #include <std
int main() {
int i, j, count
count = 0;
i = 0;
for (j = -3
{
if ((j
Cou
}
count = co
Printf ("
return 0;
}

which one of th

A) The Program
B) The program
be present w

high priority than

th

npiler dependant

ined by the

s the liberty to

take the

nted value or the

l value is not

l by the language.

as compiler is

dependent of the

on

evaluation

st

nt get evaluated

se || we care

ve && we care

```
# include <stdio.h>
int main() {
  int a=1, b=1, c=0;
  if (a != b || c++) {
    printf ("%d", c);
  }
```

Soln: Here According to order of execution

a != b will be executed since a != b is false

we goto c++ here c=0 so in if statement

c=0 is returned first and then c gets increment

So if (0) = false so nothing gets printed in

the output.

Gate CSE 2021 :

```
# include <stdio.h>
int main() {
  int i, j, count;
  count = 0;
  i = 0;
  for (j = -3 ; j <= 3 ; j++)
  {
      if ((j >= 0) && (i++))
          count = count + j;
  }
  count = count + i;
  printf ("%d", count);
  return 0;
}
```

which one of the following options are correct

A) The program will not compile successfully

B) The program will compile successfully and output 10 will be present when executed.

c) The program will compile successfully and output 3 when executed.

d) the program will compile successfully and output 13 when executed.

Soln: we know from j = -3 to -1 if statement won't be executed as j >= 0 is false

when

j = 0	j = 1	j = 2	j = 3	j = 4
i = 1	i = 2	i = 3	i = 4	for loop will break
** count = 0 + 0 = 0	count = 0 + 1 = 1	count = 1 + 2 = 3	count = 3 + 3 = 6	

→ This case wont happen because

if ($\underbrace{j >= 0}_{T}$ && $\underbrace{(i++)}_{F}$)

as i=0 will return and then i will get increment

finally count = 6, i = 4

now count = 6 + 4 = 10

* if count is initialized to '1' then we get wrong answer, Try to focus :)

Functions in c:

Lec-5

→ The structure of the function will be

```
function return type function name (Parameter list)
{
    Local variable declaration;
    executable statement-1;
    executable statement-2;
    :
    return statement;
}
```

ex:

```
float mul (float x, float y)
{
    float result;
    result = x * y;
    return result;
}
```

(float → return type; mul → function name; (float x, float y) → parameter list)

** if we write
we simply
or we sk

eg: void sum
{
printf (
return
}

→ main ( )
{
int y;
y = mul(10,5)
printf ("%d
}

mul (10,5) wi
the value re
so y = 50

Function decla
→
function
Functio
• fu
• fu
• p
• r

ex: in
Equally
functio

cessfully and output 8

cessfully and output 13

to -1 if statement wont
s false

j = 2 j = 3 j = 4
i = 3 i = 4
ont = 1+2 count = 3+3 for loop
= 3 = 6 will break

This case wont happen
because

if ($\underbrace{j>=0}_{T}$ && $\underbrace{(i++)}_{F}$)

as i=0 will return
and then i will
get increment

Lec-5

he function will be

name (Parameter list)

)
er list

** if we write the return type as void then we simply write return in return statement or we skip the entire return statement

eg:
```
void sum (int a, int b)
{
    printf (" sum = %d", a+b);
    return;
}
```
(or)
```
void display( )
{
    printf ("Hey");
}
```

```
main ( )
{
    int y;
    y = mul(10,5); -> Function call
    printf ("%d\n", y);
}
```

```
int mul (int x, int y)
{
    int p;
    p = x * y;
    return (p);
}
```
Function definition

mul (10,5) will be replaced by the value returned by the function
so y = 50

Function declaration :

→ Function declaration is also called function prototype

Function declaration consists of
- function return type
- function name
- Parameter list
- Terminating semicolon

ex: int mul (int m, int n);

Equally acceptable forms of declaration of a function are

int mul (int, int)
mul (int a, int b)
mul (int, int)

→ if we skip the return type of a function then by default compiler will take return type as int

→ in parameter list we have to specify the data type of the variables like int, float but the variable names can be skipped.

Using a function:

```
#include <stdio.h>
int fun (int x, int y); function declaration
main ()
{
    int a;
    a = fun(2,3);
    printf ("%d", a);
}
int fun (int x, int y) function definition
{
    return x+y;
}
```

→ will the following code snippet compile

```
#include <stdio.h>
main ()
{
    int a;
    a = fun (2,3);
    printf ("%d", a);
}
int fun (int x, int y) {
    return x+y;
}
```

Soln: Here we didnt declare the function so when the compiler reaches the line a = fun(3,2);

it doesnt know
will return so by
fun (2,3) will re
Now when
definition it sees
if fun() has
wont throw an
but if fun() i
type say float
throw an error
Since
wont get com

→ will the code

```
#include
main()
{
    char a
    a = fun
    printf
}
Char fun
{
    return
}
```

Soln: Here fun
sees fun
it assume
when it
to know
not int
an error

it doesnt know what fun(2,3) is and what it will return. So by default compiler assumes that fun(2,3) will return int.

Now when the compiler reaches function definition it sees the function return type and if fun() has void return type then compiler doesnt throw an error and continues to work but if fun() is having another function return type say float then in this case compiler will throw an error.

Since fun() return type is int the code wont get compilation error.

→ will the code snippet compile

```
#include <stdio.h>
main()
{
    char a;
    a = fun(2,3);
    printf("%c", a);
}
Char fun (int x, int y)
{
    return x+y;
}
```

Soln: Here fun() is not declared so when compiler sees fun() for the first time at a = fun(2,3) it assumes that fun() will return int and when it goes to function definition it came to know that fun() is returning char and not int so in this case the compiler will throw an error

The following above code gets compile time error

→ Will the following code works

```
#include <stdio.h>
main ()
{
    float a = foo ();
}
float foo ()
{
    return 3.5;
}
```

Soln: Here also we didnt declare the function and compiler assumes it as int and in the definition it sees foo () is float return type so compiler will throw an error.

The above code is having compile time error

Gate CSE 2005:

Consider the following c-program

```
double foo (double); /* Line-1 */
int main () {
    double da, db;
    // input da
    db = foo (da);
}
```

```
double foo (double a) {
    return a;
}
```

The above code compiled without any error or warning. if Line-1 is deleted, the above code will show

A) No compiler warning or error

B) Some compiler warnings not leading to unintended results

c) Some compiler warnings due to type mismatch leading to unintended results

D) compiler error

Soln: when Line-1
be no function
function

Compiler had ass
compile time er

Activation record:
Acti

stores all the
call including
values, location
The
is allocated w
deallocate whe
record gets d
by it will be
record gets
* main
also having

→ what will be
Void swap
int temp =
x = y;
y = temp
}

Soln: Actions H
Called will
main c
3 a
4 b

Soln: when Line-1 is deleted it means there will be no function declaration

function return type is double so what the compiler had assumed is wrong so we get the compile time error.

Activation record:

Activation record is a stack which stores all the necessary information about the function call including parameters, local variables, return values, location currently being executed in the function. The Activation record is temporary and it is allocated when the function is called and will deallocate when the function returns. once the activation record gets deallocated then the memory occupied by it will be available for use when new activation record gets created.

** main() is called by INIT or start() in O.S

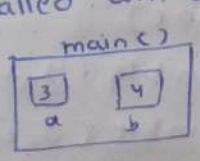
* main() is also a function and it is also having its own activation record

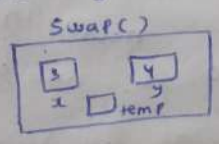
→ what will be the output for the following code

```
void swap (int x, int y) {
int temp = x;
  x = y;
  y = temp;
}
```

```
int main (void) {
  int a = 3, b = 4;
  swap (a, b);
  printf (" a = %d, b = %d", a, b);
}
```

Soln: Actions that are taken place when main() is called will be creating Activation record so





Since a, b are not global variables and x, y are not global variables, what

Changes made to x, y donot reflect to a, b

Since $temp = x \Rightarrow temp = 3$

$x = y \Rightarrow x = 4$

$y = temp \Rightarrow y = 3$

only the values of x and y are swapped not a, b

so o/p will be 3, 4

Gate CSE 2019

→ Consider the following C program

```
#include <stdio.h>
int Jumble (int x, int y) {
    x = 2*x+y;
    return x;
}
int main () {
    int x=2, y=5;
    y = Jumble (y,x);
    x = Jumble (y,x);
    printf ("%d\n", x);
    return 0;
}
```

The value printed by program is ——

Soln: Given $x=2$, $y=5$

in Jumble let $x=a$, $y=b$

so $a = 2 \times a + b$

$a = 2 \times 2 + 5 = 9$ So $y = 9$

now $x = Jumble(9, 2)$

** This is wrong as we have taken Jumble (x, y) but not Jumble (y, x)

The actual answer is

x in Jumble $= a$ now in main()

y in Jumble $= b$ $y = Jumble(y, x)$

$= Jumble(5, 2)$

So $a = 5$, $b =$

now $a = 2 \times$

$= 2 \times$

$y = 12$ now

The output

→ Will the followin

```
int fun (fl
  float temp
  return ten
}
main()
{
  fun (2.5
}
```

Soln: * Execution

but compi

during co

main() o

its work

Line-1 itse

Here Return

when co

declared

float to in

so

retur

in main

which

So $a = 5$, $b = 2$

now $a = 2 \times a + b$

$= 2 \times 5 + 2 = 12$

$y = 12$ now $x = \text{Jumble}(12, 2)$

$a = 12$, $b = 2$

$a = 2 \times a + b$

$= 24 + 2 = 26$

The output will be 26.

→ Will the following c code works

```
int fun (float x, float y) {
  float temp = x x y;
  return temp;
}
main()
{
  fun (2.5, 3.5);
}
```

Soln: * Execution of the program starts from the main() but compilation will start from Line-1 itself and during compilation Compiler doesnt care about the main() or other function it just starts executing its work which is compiling the program from Line-1 itself which is int fun (float x, float y)

Here Return type is int

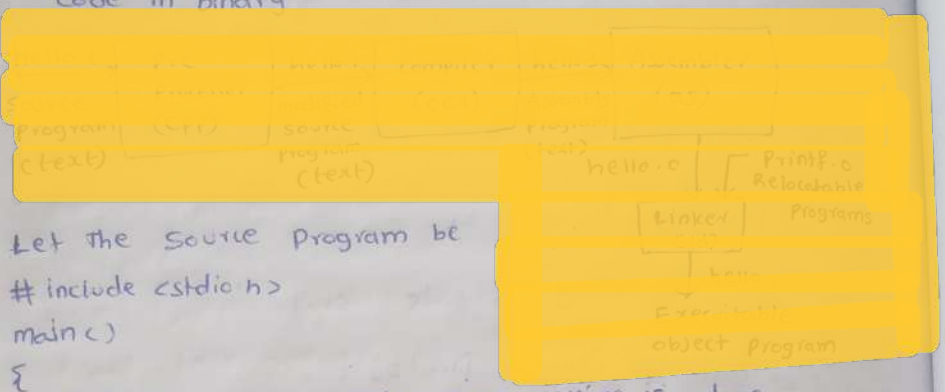
when compiler sees return temp but temp is declared as float so compiler will typecast the float to int

so $x * y = 2.5 * 3.5$

$temp = 8.75$

return temp so only 8 will be returned in main() we are not catching the returned value which is perfectly fine; * This code works.

** Now we are going to see the various phases of program from program in text file to executable code in binary



Let the source program be

```
# include <stdio.h>
main()
{
  int x, y;
  y = 5;
  x = y;
}
```

Now after preprocessing is done hello.c is converted to hello.i

This is called (hello.i) modified source program

in modified source program # include <stdio.h> is replaced with 40 to 50 lines of code

-> The difference between hello.c and hello.i is that 40 lines of code which is replaced to the place of # include <stdio.h>. Apart from this both are same

-> When hello.i is sent to compiler the output will be assembly code like mov R1, 2000 or load R1, 1000

Virtual memory :

in one of the books it is mentioned that "virtual memory is most successful idea in the computer science"

virtual memory provides an abstraction to compiler. it says that there is only one program running with memory of full size and compiler is free to store/use any memory location

** Please see the video for how compiler depends on virtual memory.

what does

he

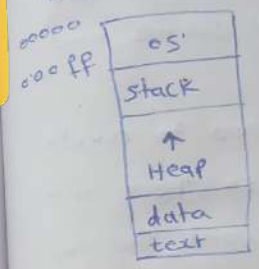
a de

the

say we are

and base add

now in virtual



in memory

-> say we

only physical

in

if we have

Compiler h

program has

offset so

careful

Assume

now 000

* 00

logical ad

Assum

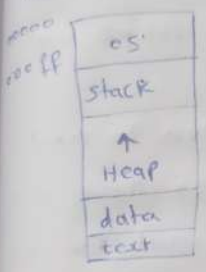
memor

what does Load R_1, 1000 mean

here 1000 can be memory address or a decimal value and this depends on the addressing mode that we use

say we are using base register addressing mode and base address will be present in base register

now in virtual memory



operating system will be present and we cant touch it so the base address will be 00ff now since compiler thinks that we have all the memory it will generate the offset 1000

so Load R_1 1000 means Load the value in memory 00ff + 1000 into Register R_1

→ say we dont have virtual memory and we only physical memory

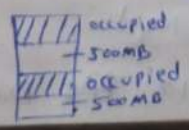
in the above 00ff + 1000 is also virtual memory

if we have only physical memory then compiler has to see before what offset does the program has access to and then generate the offset so in physical memory we need to be careful

Assume 1000 is available to the program so now 00ff + 1000 will be physical memory address

* 00ff + 1000 is called Effective address or logical address

Assume that the current state of physical memory is



The program that we are compiling needs 1 GB of memory but we dont have 1 GB memory contiglously

Load Ri offset → say we need more than 500 MB and less than 1 GB

in this case compiler cant execute or compile the program But operating system says to the compiler that i can help you execute the program and you generate two base addresses and put them in two base registers first base address is 1st 500 MB and second base address is 2nd 500 MB

But this is not a good idea to create different base addresses dynamically so there are some limitations using physical memory and this case is one of them.

→ Now the Assembler takes the Assembly code and each line to binary

i.e mov R_1, R_2 we know mov has fixed bit string as well as R_1, R_2 has

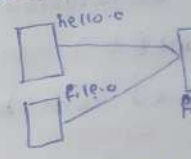
so in Assembler mov is replaced with that bit string along with R_1, R_2

Say we have 100 lines in hello.i then in hello.s we have more than 100 lines for sure say 300 lines then hello.o Also has 300 lines as Assembler converts every line in hello.s to binary.

→ Linker helps to link two programs i.e say hello program has int a and file program is using a say a+=1 now we need to link hello, file programs so at hello.o, file.o they are linked

* Both hello, file programs are compiled

Independently



→ Even if there have only hello.o default files like hello.o

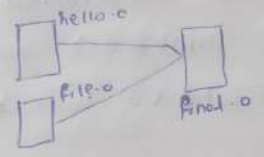
→ will this program

```
main()
{
printf("Hi");
}
```

Soln: The compiler printf is n assume that since n defined so defined in current pro

Now compiler a say printf checks if file.c if pri linker pro throw an ins function

independently



→ Even if there are no shared variable and we have only hello.o then also we need linker as some default files like printf.o needs to be linked with hello.o

→ Will this program compile

```
main()
{
printf("Hi");
}
```

Assume #include <stdio.h> is not written

Soln: The compiler sees the function printf() and since printf is not declared by default compile will assume that return type of printf() is int

Since no where in the program printf() is defined so compiler thinks that printf() may be defined in another program that will link to the current program.

Now during linking the linker sees that the compiler assumed printf() return type is int and say printf() is defined in file.c now the linker checks if printf() return type is int or not in the file.c

if printf() return type is int in file.c then linker proceeds further and if not the linker will throw an error

instead of printf() we can have any function like fun() and the compiler thinks same way as above

* stdio.h contains only the declaration of printf() not the definition of printf()

The definition of printf() will be present in .lib - file

So even if we are not including stdio.h it means we are not declaring the printf()

```
main()
{
printf("Hello");
}
```

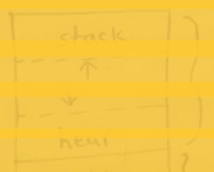
This will work because we are lucky enough that printf has "int" return type.

→ How to share a function among two files/programs

As a programmer we dont need to do any extra work as compiler will take care of it

* if we need to share a variable among two files/programs then we use "extern" keyword

memory layout of C program:



The Actual code of the program will be present in the static part of memory

Along with code, static part has global variables and static variables

when executing the program Activation record gets created and that Activation record gets pushed into the stack.

* There are no
and heap space

→ in stack memo
local vari
function c
Heavi

→ in Heap memo
malloc()
free()
The

→ in static memo
static var
Global va

```
#include <stdio.h>
#include <stdlib.h
int x;
int main()
{ int y;
  char *str;
  y=4;
  str = malloc(
  str[0] = 'm';
  free(str);
  return 0;
}
```

Storage classes:

There
① Aut
② reg
③ sta
④ ex

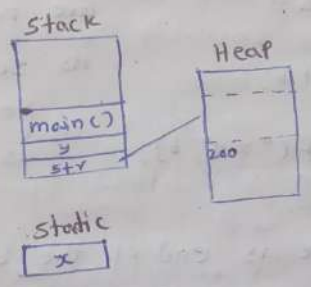
and heap space as we use them at runtime

> In stack memory we have
local variables Run time
function calls
Heavily used when using recursion

> In Heap memory we have
malloc() to allocate memory Run time
free() to deallocate memory
These are dynamic memory allocation

> In static memory we have
static variables compile time
Global variables

```
#include <stdio.h>
#include <stdlib.h>
int x;
int main ()
{ int y;
  char *str;
  y=4;
  str = malloc(100 * sizeof (char));
  str [0] = 'm';
  free (str);
  return 0;
}
```



Storage classes :

There are four storage classes

1. Auto
2. register
3. static
4. extern

Auto storage class:

in c programming variables can be present at 3 memory locations ① Heap ② stack ③ static

Auto int a;

① storage will be in stack

② Initial value will be Garbage value

③ Scope will be within the block

```
fun ()
{
  Auto int a;
  . . .
}
```

block ←

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

This code wont work because we are printing value of b outside of its scope
So we can say scope of b is with in the block

④ Life time is end of the block
life time of the variable tells that variable is present in the memory or destroyed.

⑤ Since stack is used at run time; Auto int a will be runtime allocation

⑥ Auto storage class is the default storage class for all local variables

⑦ objects with the auto class are not available to the linker

file1.c
int main ()
{ int a;
}

file2.c
{ → file2 cant be linked to variable
} a in file1 as a is local variable

and its scope is within

→ auto int a; /* illegal --
main ()
{
auto int b; /* vali
if (a > b)
{ auto int a = c
}
}

Static storage class:

* Static will b
class for "GA

① storage will be in
② Initial value will be
③ Scope will be with

i.e static int a;
main ()
{
}
fun ()
{
}

we can use 'a' anya
in the program eve
fun () too

④ life time will be

i.e main ()
{
int b = fun ();
printf ("%d",
fun ();
printf ("%d",
}

and its scope is within the block

```
-> auto int a ; /* illegal -- auto must be within block */
   main ()
   {
     auto int b ;  /* valid auto declavation */
       if (a > b)
         {
           auto int a = a+b  /* valid inner block declaration */
         }
   }
```

Static storage class :

* Static will be the most important storage class for "GATE"

① storage will be in static memory

② Initial value will be zero

③ Scope will be within the block

i.e

```
static int a;
main ()
{
}
fun ()
{
}
```

we can use 'a' anywhere in the program even in fun () too

```
main ()
{
  static int a;
  ---
}
fun ()
{
}
```

Here we can use 'a' within the main() and it can be used in fun ()

④ life time will be till the end of program

i.e

```
main ()
{
  int b = fun();
  printf ("%d", b);
  fun ();
  printf ("%d", b);
}
```

```
fun ()
{
  static int a =5;
  a++;
  return a;
}
```

when we call int b = fun()

a = 5
a++ => a = 6

So b = 6

Here when we return from a function the activation record will be destroyed so all local variables will be destroyed

but static 'a' it is global and it cant be destroyed

So when we again call fun()

a will take value 6 from starting

so a++ => a = 7

now the value of a is 7

This is the difference between scope and lifetime

static variable will only be initialized only once

and

we cant access 'a' in main() as its scope is within fun()

and

Even when we return from fun() a is in the memory as 'a' has lifetime till the end of program

→ static can be local and global

i.e

```
Static int a;
main()
{
}
```

Here a is global
i.e static is global

```
main()
{
  static int a;
}
```

Here static is local

→ Even though stati
accessable to other fi
to linker.

→

```
# include <stdio.h>
void change_static()
{
  static int s;
  s = s+2;
  printf("%d", s);
}
int main()
{
  change_static
  change_static
  change_static
}
```

Soln: initial value

change_stati
change_stat
change_sta

Extern storage cl

→ Extern

to share a var

① storage will be
② Initial value
③ Scope is Glo
④ lifetime is til
⑤ objects with
linker

Extern in

function the activati
ocal variables will be

it cant be destroyed

starting

Scope and lifetime
ialized only once

main() as its

n fun() 'a' will
has lifetime is

not available to

another file/
program

;

is local

-> Even though static could be Global it is not accessable to other files/programs so not available to linker.

```
-> #include <stdio.h>
   void change_static (void)
   {
     static int s;
      s = s + 2;
     printf ("%d", s);
   }
   int main ()
   {
     change_static ();
     change_static ();
     change_static ();
   }
```

Soln: initial value of s = 0 since it is static

change_static () -> o/p : 2

change_static () -> o/p : 4

change_static () -> o/p : 6

Extern storage class: Lec-6 : 22-8-21

-> Extern keyword is used when we want to share a variable

1. storage will be in static memory
2. Initial value will be zero
3. Scope is Global (we can use anywhere in the program)
4. lifetime is till the end of program
5. objects with Extern class are available to the linker

Extern int a; it will be the default declaration of extern

```
-> int max;
   main ()
   {
     int len;
     extern int max; -> L1
     printf ("%d", max);
     max = 5;
   }
```

Assume that L_1 is removed then when compiling compiler sees max as global variable so it will Print '0'

* Even if L_1 is present then also '0' will be the o/p and there wont be compilation error

```
-> main ()
   {
     int len;
     extern int max; -> L1
     Printf ("%d", max);
     max = 5;
   }
   int max;
```

* Here we used declaring instead of defining for variables as we dont know the difference but we learn in detail when we go further.

if L_1 is not present then compiler will throw an error at printf ("%d", max) as max is not declared or defined

* if L_1 is present then compiler wont throw an error

extern int max;

it means we are not declaring a variable max, when we use extern int max we are saying to compiler that max is some where in the program so dont throw error

* Since we are n
extern in
no memory

-> it is a Commo
of all global Varia
file and omit all
Then why do

The answer is to
Present in another
i.e

```
/* main.c */
# include <stdio.h>
extern int gint;
Void change_extern
int main ()
{
Printf ("%d\n", gin
change_extern ();
gint = 5;
change_extern ();
Printf ("curr val of
return 0;
}
```

-> in extern.c
we compile
because to
so extern.c

main.c also
declared gint
extern.c

en when compiling
variable so it will

n also '0' will be
npilation error

Here we used declaring
ead of defining for
riables as we dont
ow the difference but
learn in detail when
go further.

ompiler will throw
) as max is not

Compiler wont throw

not declaring a
e extern int max
that max is some
nt throw error

* since we are not declaring max in
extern int max;
no memory will be allocated.

→ it is a common practise to place definitions of all global variables at the beginning of source file and omit all extern declarations.

Then why do we need extern keyword?

The answer is to share a variable which is present in another file.

i.e

```
/* main.c */
#include <stdio.h>
extern int gint;
Void change_extern(void);
int main()
{
Printf ("%d\n", gint);
change_extern();
gint = 5;
change_extern();
Printf ("curr val of gint %d\n", gint);
return 0;
}
```

```
/* extern.c */
#include <stdio.h>
int gint;
void change_extern()
{
Printf ("change_extern:
curr_val %d\n",
gint);
gint = 10;
Printf ("change_extern:
changed_val %d\n",
gint);
}
```

→ in extern.c there is no main() and when we compile extern.c there will be no error because to compiler main() is just a function so extern.c will be successfully compiled.

main.c also wont throw an error as we declared gint as extern and gint is defined at extern.c

→ Compiler has access to the file which it is curr
Compiling but linker has access to all the files
During linking the linker sees that gint is defin
in extern.c so now linker will combine both
files as a single file

i.e

```
#include <stdio.h>
int gint;
void change_extern (void);
int main ( )
{
  - - -
  - - -
  - - -
}
void change_extern (void)
{ - - -
  - - -
  - - -
}
```

→ Command to link main.c, extern.c is

gcc main.c extern.c

The o/p of above file is

```
0
change_extern : Curr-value 0
change_extern : Change-value 10
change_extern : Curr_value 5
change_extern : change_value 10
curr value of gint : 10
```

```
/* File-1 */
→ extern int i;
main ()
{
  printf ("%d", i);
}
```

```
/* File-2 */
fun ()
{ int i;
  i=1;
}
```

Does This have Linker error?

we know File-1 and
independently
File-1 dont have
File-2 dont have
but during linking
int i in file-2
have access to it
file-2 according to
work so linker w

→ /* File-1 */
main ()
{
extern int i;
printf ("%d", i)
}

Does thi

we can write
its scope is
but it show
another file a
variable and
So there

Declaration vs

→ Declaratio
accept referen
ext
ext

do
* f

we know File-1 and File-2 are Compiled independently

File-1 dont have compilation error

File-2 dont have compilation error

but during linking

int i in file-2 is auto class so linker dont have access to it and int i is not defined in File-2 according to linker so extern int i wont work so linker will throw an error.

->

Declaration vs Definition :

-> Declaration is what the Compiler needs to accept references to that identifier

```
extern int bar;
extern int g (int, int);
```

we can use extern to functions also

```
double f (int, int);
```

* function declarations are by default extern.

→ Definition is what linker needs to know in order to link references to those entities.

int bar;

in definition of variable we create memory for the variable

int g (int lhs , int rhs) { return lhs * rhs;}

double f (int i, int d) { return i+d ; }

in short we can say

Declaration : " somewhere , there exists foo "

foo can be function or variable

Definition : "and here it is!"

→ How to define a function without declaring it first ?

we just need to define the function before main ()

i.e
```
fun ( )
{ - - -
  - - -
}
main ( )
{ - - -
  - - -
}
```

→ How to declare a variable without defining it?

Int var; → it is definition as memory will be created for var

so use extern

i.e extern int var; → it is declaration as no memory is created for var

→ How to define a variable using extern keyword (local and global variables)

Simply initialize i

i.e extern int a

→ is there any diffe

declarations

1: extern int fun

2: int fun ( );

A) Both are identical

c) First declaration is wrong

D) Difference is Co

Soln: Both are iden

extern when

but for varia

int var; /

* Note: Funct

because c does

inside other f

i.e fun1 (

{

- - -

fun 2

{

}

}

→ main ( )

{

extern int i

int i = 0;

Printf ("Y

eds to know in
ese entities

riable we create
iable

{ return lhs * rhs;}
return i+d ; }

e exists foo "
foo can be function
or variable

hout declaring it

e function before

thoot defining it?
on as memory will
r var

is declaration as
memory is created
r var

extern keyword
riables)

Simply initialize it to a value

i.e extern int a = 5;

here memory will be created for a

-> is there any difference between following declarations

1: extern int fun();

2: int fun();

A) Both are identical

B) No difference, except the extern int fun() is in another file

C) First declaration is wrong

D) Difference is compiler dependent

Soln: Both are identical as functions are by default extern when declaring

but for variables

int var; , extern int var are completly different as int var has memory allocated

* Note: Functions themselves are always extern because C doesnot allow functions to be defined inside other functions

i.e
```
fun1 ()
{
  ---
  fun2 ()
  {
    ---
  }
}
```
it is not allowed in C

-> main ()
```
{
  extern int i;
  int i = 0;
  Printf ("%d", i); }
```
it throws an Error because extern should alway refer to a global variable not to a local variable.

* Sometimes Extern is also called as External

Register storage class:

Register is used for local variables Just like auto

① storage will be in the cpu register

② Initial value will be garbage value

③ Scope is within the block

④ Lifetime will be End of the block.

```
Void main ()
{
    register int i=0;
    printf ("%d", i);
}
```

we are suggesting the compiler to put i in the register within the cpu

→ Compiler can ignore our suggestion and can keep i in memory

→ most modern compilers will automatically keep variables that are heavily used in registers

* if our suggestion to keep i in register is ignored then i will automatically convert to auto

→ Does global variables in C are static or extern by default?

if global variables are by default static then it means we would be able to access them only in the same file but global variables can be accessed by other files too

does this mean global variables are extern by default

extern int a;

Also we have

be accessed by

can be global

Auto, regis

So there is

global varia

→ By default

static memo

static stor

→ #include <st

int main ()

{

auto int

{

aut

Pri

}

Printf

return

}

A) 10 10 B)

Soln: First p

a=15

Second

the ans

→ #include

main ()

{

extern

num = 20

Printf (

extern int a; is not same as int a

Also we have seen that static variables cant be accessed by linker even though static variable can be global

Auto, register are used for local variables so there is no default storage class for the global variable.

→ By default global variables gets stored in static memory and available to linker so static storage and External linkage

→
```
#include <stdio.h>
int main()
{
    auto int a=10;
    {
        auto int a=15;
        Printf ("%d", a);
    }
    Printf ("%d", a);
    return 1;
}
```

A) 10 10 B) 10 15 C) 15 10 D) Compiler error

Soln: First printf will print 15 because local variable a=15 has high precedence over int a=10 and Second printf will print 10 so 15 10 will be the answer

→
```
#include <stdio.h>
main()
{
extern int num;
num=20;
Printf ("%d", num); }
```

a) compile time error
b) linker error
c) runtime error
d) No error

Soln: when compiling extern int num; Compiler will assume that num is present in some other file i.e num = 20 is there since it is not defined compiler will assume num is defined in some other file but there is no other file so linker will throw an error * when executing or compiling n = 20 Let num=n compiler wont say I didnt see num before as we have declared num as extern so compiler think num is defined as global in another file.

```
-> #include <stdio.h>
   extern int i = 10;
   int main()
   {
     extern int i;
     printf("%d", i);
     return 0;
   }
```

A) Garbage value B) 10 C) Compiler Error D) Linker Error

Soln: Here we are defining i as global variable so in main() extern int i; will be useless so o/p will be 10

```
-> #include <stdio.h>
   int main()
   {
     extern int i = 10;
     printf("%d", i);
     return 0;
   }
```

A) Garbage Value
B) 10
C) Compiler Error
D) Linker Error

Soln: Here we are defining i in local scope but extern says that i should be global so we get compile time error.

```
-> #include <stdio.h>
   extern int i;
```

t num; Compiler will
ent in some other
e since it is not
ne num is defined
is no other file
or * when executing or
iling n = 20 Let num = n
iler wont say I didnt
num before as we have
ared num as extern
compiler think num
defined as global in
er file.

er Error b) Linker Error
as global variable
will be useless

Value

Error
rror

local scope but
global so we

```
int main ()
{
  i = 5;
  printf ("%d", i);
  return 0;
```

Soln: Here compiler thinks that i will be defined in another file but there is no other file so we get linker error.

```
#include <stdio.h>
void fun1()
{
  auto int x = 1;
  static int y;
  register char z = 'F';
  printf ("%d %d %c", x, y, ++z);
int main ()
{
  fun1();
  return 0;
}
```

A) 1 0 F B) 1 1 G C) 1 0 G
D) 1 1 F1

Soln: z = 'F' so z++ or ++z will be 'F' + 1 = 'G'

So 1 0 G will be the o/p

Recursion:

→ A function calling itself is called Recursion

we can also say functions defined interms of itself are called recursive functions

i.e

```
f()
{
  ---
  f()
}
```

and

```
f1()
{
  ---
  f2(n)
}
```

it is not a recursion

→ we know

$n! = n \times n-1 \times n-2 \times \cdots \times 1$

$n! = n \times (n-1)!$

we can write

$f(n) = n \times f(n-1)$

in programming we can write

```
factorial (int n)
{
    int fact;
    if (n==1)
        return (1);   ] Base case
    else
        fact = n * factorial (n-1);
    return (fact);
}
```

→ int recsum (x) {

```
    if (x==1)
        return x;
    else
        return x + recsum(x-1);
}
```

now 15 recsum (5)

5 + 10 recsum(4)

4 + 6 recsum(3)

3 + 3 recsum(2)

2 + 1 recsum(1)

* This is called Recursion tree method

recsum (5)

5 + recsum (4)

5 + (4 + recsum (3))

5 + (4 + (3 + recsum (2)))

5 + (4 + (3 + (2 + recsum (1))))

5 + (4 + (3 + (2 + 1)))

15

* This method is called unrolling

Gate CSE 2004:

→ Consider the following C-Function

```
int f (int n)
{
    static int i=1;
    if (n>=5) return n;
    n= n+i;
    i++;
    return f(n);
}
```

The value returned by f(1) is

A) 5 B) 6 C) 7 D) 8

Soln: we have a static variable which is shared by all recursive functions

$\boxed{\emptyset}$ ~~1~~ 2 3
i

f(1)	f(2)	f(4)	f(7)
i=1	n<5	n<5	n>5
n=1+1	n=2+2	n=4+3	return 7
=2	=4	=7	
i=2	i=3	i=4	
return (f(2))	return (f(4))	return (f(7))	
7	7	7	

so f(1) will return 7

Gate CSE 2007:

→ Consider the following C-program

```
intf (int n)
{
    static int r=0;
    if (n <= 0) return 1;
    if (n > 3)
    {
        r = n;
        return f(n-2) + 2;
    }
    return f(n-1) + r;
}
```

what will be the value printed / returned by f(5)

A) 5 B) 7 C) 9 D) 18

Soln:

$\boxed{\cancel{0}}$ 5
r

f(5)
5 > 3
r = 5
return f(3) + 2
16

f(3)
3 > 3 false
return f(2) + 5
11

f(2)
return f(1) + 5
6

f(1)
return f(0) + 5

f(0) return 1

f(5) = 16 + 2 = 18

Gate CSE 2016:

→ what will be the output of C-program

```
Void Count (int n)
{
    static int d=1;
    Printf ("%d", n);
    Printf ("%d", d);
    d++;
    if (n>1) Count (n-1);
    Printf ("%d", d);
}
void main ( ) {
    Count (3);
}
```

A) 3 1 2 2 1 3 4 4 4

B) 3 1 2 1 1 1 2 2 2

C) 3 1 2 2 1 3 4

D) 3 1 2 1 1 1 2

Soln:

$\boxed{4}$ 2 3
d

C(3)
3, 1,
d = 2
C(2)
4

C(2)
2, 2,
d = 3
C(1)
4

Gate CSE 2017:

→ Consider the foll

```
Void fun1 (int n) {
    if (n==0) retur
    Printf ("%d", n)
    fun2 (n-2);
    Printf ("%d",
}
```

The output printed

A) 5 3 4 2 3 1 2 2 2 3 3

C) 5 3 4 2 3 1 2 2 1 3 2

Soln: fun1 (5)
fun2 (3)
Print (5)

fun1 (3)
fun2 (1)
print (3)

* fun (static
{
}

So this

Soln: $\boxed{+}$ 3 Print: 3, 1, 2, 2, 1, 3, 4, 4, 4
d

c(3)	c(2)	c(1)
	2, 2,	1, 3
3, 1,	d = 3	d = 4
d = 2	c(1)	n>1 is false as 1>1
c(2)	4	4
4		

Gate CSE 2017:

→ Consider the following two functions

```
Void fun1(int n) {
    if (n==0) return;
    printf("%d", n);
    fun2(n-2);
    printf("%d", n);
}
```

```
Void fun2 (int n) {
    if (n==0) return;
    printf("%d", n);
    fun1(++n);
    printf("%d", n);
}
```

The output printed when fun1(5) is called is

A) 5342312223344 5 B) 5342312011223 3

C) 5342312213243 5 D) 5342312021324 3

Soln: fun1(5) Print: 5342312223344 5

fun2(3)	fun2(3)	fun1(4)	fun2(2)
Print(5)	fun1(4)	fun2(2)	fun1(3)
	print(4)	print(4)	print(3)

fun1(3)	fun2(1)	fun1(2)	fun2(0)
fun2(1)	fun1(2)	fun2(0)	↓ return
print(3)	print(2)	print(2)	

int fun (static int i)
{
}

it is not allowed to pass i to a function as static

```
-> fun (int n)
   {
     if (n==0)
       return;
     Printf ("%d", n);
     fun(n--);
   }
```

what will be the o/p when fun(4) is called

Soln:

```
fun (4)    fun (4)
pf (4)     pf (4)     . . . . . .
fun (4)    fun (4)
```

it will call fun(4) infinite times as we are having post decrement

so o/p will be 44444----- infinite times.

```
-> fun (int n)
   {
     if (n==0)
       return;
     Printf ("%d", n);
     fun (--n);
     fun (n--);
   }
```

what will be o/p when fun(3) is called

321121

Soln:

```
fun (3)    fun (2)    fun (1)
pf (3)     pf (2)     pf (1)
fun (2)    fun (1)    fun (0)
fun (2)    fun (1)    fun (0)
```

The o/p will be

3 2 1 1 2 1 1

Try to be careful if we have increment or decrement operators

Gate CSE 2000:

-> The value of J at the end of execution of the following C-Program

Lec-7 25-08-2

the o/p when fun(4)
ed

e times as we are

infinite times.

o/p when
ned

e increment or

execution of the

```
int incr (int i)
{
    static int count = 0;
    count = count + i;
    return (count);
}
main() {
    int i, j;
    for (i=0; i<=4; i++)
        j = incr(i);
}
```

C = 0

i = 0 ⇒ C = 0

i = 1 ⇒ C = 1

i = 2 ⇒ C = 3

i = 3 ⇒ C = 6

i = 4 ⇒ C = 10

A) 10 B) 4 C) 6 D) 7

Soln:

i = 0	i = 1	i = 2	i = 3	i = 4
j = incr(0)	j = incr(1)	j = incr(2)	j = incr(3)	j = incr(4)
count = 0	count = 0+1 = 1	count = 1+2 = 3	count = 3+3 = 6	count = 6+4 = 10
j = 0	j = 1	j = 3	j = 6	j = 10

So value of j = 10 at the end of execution

Gate IT 2004

→
```
# include <stdio.h>
int funcf (int x);
int funcg (int y);
main ()
{
    int x=5, y=10, count;
    for (count = 1; count <= 2; ++count)
    {
        y += funcf(x) + funcg(x);
        printf ("%d", y);
    }
}
```

```
funcf (int x) {
    int y;
    y = funcg(x);
    return (y);
}
```

```
funcg (int x) {
    static int y = 10;
    y += 1;
    return (y + x);
}
```

A) 43 80

B) 42 74

C) 33 37

D) 32 32

Soln: Count = 1 :

$Y = 10 + func f(5) + func g(5)$

$\boxed{10} \to 11 \to 12 \to 13$
Y

func f(5)
let Y = a
a = func g(5)
a = 16

func g(5)
Y = 10
Y = 11
ret (11+5) = ret(16)

func f(5) return 16

func g(5)
Y = 11
Y = 12
ret (12+5) = ret(17)

so Y = 10 + 16 + 17
= 43

Count = 2 :

$Y = 43 + func f(5) + func g(5)$

func f(5)
let Y = a
a = func g(5)

func g(5)
Y = 12
Y = 13
ret (13+5) = ret(18)

Y = 43 + 18 + 19

Y = 80

So 43, 80 will be printed

(* if we have

y = fun1() + fun2() + fun3()

Then which function should we call first is not defined by the language so it is compiler dependent

i.e we can call fun1() first then fun2() then fun3()

or any other order may be possible

Pointers in c programming :

int quantity = 179 ; ⇒ $\boxed{179}$ → value
quantity → variable name

from now onwards

Quantity → variable name
179 → value
5000 → address

→ Pointer is nothing but a variable that contains an address

* int quantity = 179;

it means first we are assigning 179 to quantity and then we are telling that quantity is integer type.

Let p be a pointer

now p = &quantity;

address of (operator)

Variable:	Value:	Address:
quantity	179	5000
p	5000	5048

* when we print the address of quantity then the address that is in output is the Logical address not physical address

→ Physical address is only known to the operating system.

→ we can treat address as an unsigned integer so we use "%u" as format specifier.

But the problem is

if address is 64 bit long and integer is 32 bits then if we use %u then we get first 32 bits of 64 bit address or last 32 bits of 64 bit address

So the solution is using "%p" which will always work.

Simple program to print address of data types

```
main()
{
  char a;
  int x;
  float p, q;
  a = 'A';
  x = 125;
  p = 10.25, q = 18.76;
  Printf("%c is stored at address %u", a, &a);
  printf("%d is stored at address %u\n", x, &x);
  printf("%f is stored at address %u\n", p, &p);
  printf("%f is stored at address %u\n", q, &q);
}
```

Output:

A is stored at address 4436

125 is stored at address 4434

10.250000 is stored at address 4442

18.760000 is stored at address 4438

-> int *p; /* integer pointer */

-> float *x; /* float pointer */

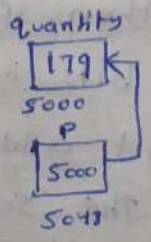
-> char *pc; /* character pointer */

int *p; ⇒ [ ] p → Points to unknown location

contains garbage

General way of using a pointer is

```
int quantity = 179;
int *p;
p = & quantity;
```



Pointer declaration style :

```
int* p;   /* style-1 */
int *p;   /* style-2 */
int * p;  /* style-3 */
```

mostly style-2 is used because

int *p, a we can declare many variables at a time i.e p is a integer pointer
a is normal integer variable

illegal uses:

→ we cannot point at constants

&125 is invalid as constants dont have memory address

→ we cannot point at expressions

&(x+y) is invalid

→ register int x;

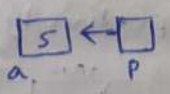
int *p = &x; is invalid because we are storing x in a register which dont have address.

* we cannot take an address of register variable even if compiler decides to keep variable in memory rather than in addressless register.

* register keyword is not at all useful in C-programming because compilers are smart enough to decide which variables are to be kept in the register.

Accessing a variable through its pointer:

Let a = 5 , p is a pointer pointing to a



now we can access the value of a using *p " Think of 'a' as a house and

writing p means we are standing outside the house so using *p we get inside the house and we access the value of a"

Here * is key to house so using the key we can enter into the house.

→ int a, b;

int *p; it is equivalent to b = a;

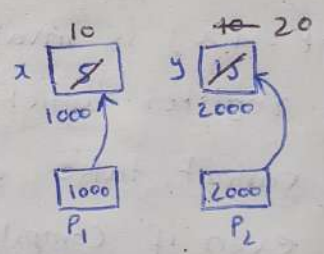
p = &a;

b = *p; * if p is pointing to 'a' then where ever 'a' is present we can replace it with *p

* we can use "%d" for address as compiler will treat the address as -ve number if m of the address is 1

→ what is the output of the following code snippet

```
int x=5, y=15;
int *P1, *P2;
P1 = &x;
P2 = &y;
*P1 = 10;
*P2 = *P1;
P1 = P2;
*P1 = 20;
Printf("%d %d", x, y);
```



*P1 = 10 ⇒ x = 10

*P2 = *P1 ⇒ y = x

P1 = P2 ⇒ P1 pointing to y

*P1 = 20 ⇒ y = 20

so 10, 20 will be printed

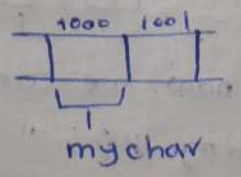
→ char *mychar;

short *myshort;

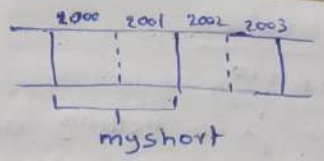
int *myint;

Suppose these pointers point to locations 1000, 2000, 3000 respectively

Since char is 1 Byte



Short is 2 Bytes



int is 4 Bytes

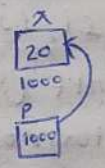
Passing pointer or address to a function:

when we pass an address to a function then in the function we should have someone to catch the address

only pointer has the capacity to hold the address so we define or declare a pointer in the parameter list of the function

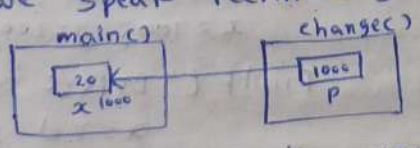
i.e

```
main()
{
    int x;
    x = 20;
    change (&x);
    printf ("%d\n", x);
}
change (int *p)
{
    *p = *p + 10;
}
```



* C-Programming doesnt have call by address or call by reference, it only has call by value

if we speak technically using activation record



x is in activation record of main()

p is in activation record of change()

So we can access the value of x in change()

using *P

so *P = *P + 10;

it means x is getting added to 10

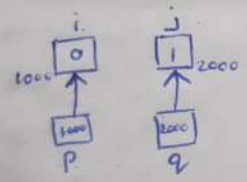
so value of x is 30

* Since x is local variable in main() we can access x when we are in the Chang() will be using *P

Gate 2010:

```
-> #include <stdio.h>
   Void f (int *p, int *q)
   {
      p = q;
      *p = 2;
   }
   int i=0, j=1;
   int main()
   {
      f(&i, &j);
      printf ("%d %d\n", i, j);
      return 0;
   }
```

Soln:



P=q ⇒ P is pointing to j

P=2000 ⇒ *p = 2 ⇒ j = 2

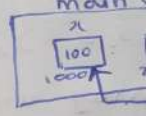
So o/p or final values of i,j are 0, 2

```
-> swap (int *a, int *b)
   {
      int *t;
      t=a;
      a=b;
      b=t;
   }
```

```
main()
{
   int x, y;
   x = 100; y=200;
   Printf (" Before call:
            x = %d y = %d \n", x, y);
   Swap (&x, &y); /* call */
   Printf (" After call:
            x = %d y = %d \n", x, y);
```

Soln: let us s

main



t = a

a = b

b = t

we are not

main()

So o/p

-> if

swa

{

in

t

*

*

}

So wher

it is not

Array:

-> Array

a collecti

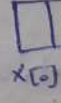
data type

-> All the

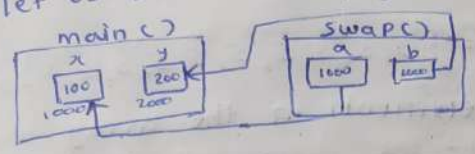
same na

-> Indivi

the in



Soln: let us solve it using activation record



$t = a \Rightarrow t = 1000$

$a = b \Rightarrow a = 2000$

$b = t \Rightarrow b = 1000$

what this is doing is changing the pointers i.e a points to y b points to x

we are not touching x, y in main()

So o/p will be 100, 200

→ if we have swap function as

```
swap (int *a, int *b)
{
  int t;
  t = *a;
  *a = *b;
  *b = t;
}
```

Then

$t = *a \Rightarrow t = 100$

$*a = *b \Rightarrow x = y \Rightarrow x = 200$

$*b = t \Rightarrow y = t \Rightarrow y = 100$

So when printing x, y 200, 100 will be printed

it is not "FAKE SWAP"

Lec-8 23-8-21

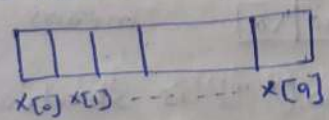
Array:

→ Array is a data structure which can represent a collection of data items which have the same data type (float/int/char/...)

→ All the items constituting the group share the same name

```
int x[10];
```

→ Individual elements are accessed by specifying the index



x is a 10 element one dimensional array

Array initialization:

→ int myarray [10] = {5, 5, 5, 5, 5, 5, 5, 5, 5, 5};

All the elements in the array are '5'

→ int myarray [10] = {1, 2}; // initialize to 1, 2, 0, 0, 0, ...

it means first element = 1

second element = 2

remaining 8 elements are 0's

→ int my array [10] = {0}; // all elements are 0

first element = 0

remaining 9 elements = 0

→ Static int my array [10]; // all elements are 0

→ int a [10];

Then by default all the elements in the array 'a' are garbage values

→ in case of character array default value is "null character" represented as '\0' (ASCII value is 0)

char t[5] = {'a', 'b', 'c', 'd'}; // initialize to 'a', 'b', 'c', 'd', '\0'

char t[4] = {'a', 'b', 'c', 'd'}; // initialize to 'a', 'b', 'c', 'd'

char s[5] = "abcd"; // initialize to 'a', 'b', 'c', 'd', '\0'

char s[4] = "abcd"; // Initialize to 'a', 'b', 'c', 'd'

char a[4] = {'a', 'b'} char b[3] = "abc"

means | 'a' | 'b' | '\0' | '\0' | means | 'a' | 'b' | 'c' |

* writing in double quotes (" ") is shortcut of writing each character

* writing in double quotes doesnt always means there is null character in the end

only for strings there will be null character in the end

* There is no data type called string in the C programming.

string is a bunch of characters

char s[5] = "abcde"

↓ we are asking for a space of 5 characters

↓ we are supplying 5 characters so there isn't be space for the 'null character' this is the reason we say writing in double quotes doesn't always mean there is a null character

-> if dimension is not specified the compiler will deduce the dimension from the initializer list

i.e int myarray[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }

which means

int myarray [9] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }

compiler thinks that we are interested in only 9 elements so even if we dont mention the dimension, compiler will deduce it

* char s[5] = "abcd" is equivalent to

char s[5] = { 'a', 'b', 'c', 'd', '\0' }

i.e s - | 'a' | 'b' | 'c' | 'd' | '\0' |

when we dont mention about the dimention of char array compiler always wants to end it with

with null character.

→ How to force compiler to not to put null character at the end of string

Simply specify the dimension of the char array

i.e char a[5] = "abcde";

* char a[] = "abcd"

Here compiler assumes "abcd" as string

So when dimension is not specified it treats it as string not as char array

but

char a[] = {'a', 'b', 'c', 'd'}

Here there will be no null character at the end as it represents char array

* char a[5] = "ab"

Here compiler puts 3 null characters at the end of 'a', 'b' i.e

'a'	'b'	'\0'	'\0'	'\0'

not extra null character is added because we have specified the size i.e we need 5 character space

→ char s1[4] = "abc";

char s2[4];

s2 = s1; /* ERROR */

The array name cant be used in the left side of assignment, more about this we will see later.

Sizeof();

Size of () will take data type as input and returns the size that datatype

occupies in

Printf (" size o

printf (" size

printf (" size

printf (" size

→ we can also

i.e char c;

double d;

Printf (" si

printf (" s

→ for simpl

int = 4

→ int a[10];

Printf ("%d

Printf ("%d

Soln: a is an

size 4B

a[0] is i

→ #include <std

int main () {

short i = 20;

char c = 97;

Printf ("%d

return 0;

}

Soln: c + i he

so cho

So int

occupies in Bytes as output

```
printf (" size of (int) = %d\n", size of (int));
printf (" size of (char) = %d\n", size of (char));
printf (" size of (float) = %d\n", size of (float));
printf (" size of (double) = %d\n", size of (double));
```

→ we can also write the variable name in sizeof()

i.e
```
char c;
double d;
printf (" size of variable c = %d\n", size of (c));
printf (" size of variable d = %d\n", size of (d));
```

→ for simplification assume that

int = 4B, char = 1B, float = 4B, double = 6B

→
```
int a[10];
printf ("%d", size of (a));
printf (" %d", size of (a[0]));
```

Soln: a is an array with 10 elements each of size 4B so total size of a = 4×10 = 40B

a[0] is int so size of a[0] = 4B

→
```
#include <stdio.h>
int main () {
short i = 20;
char c = 97;
printf (" %d, %d, %d", size of (i), sizeof (c), sizeof (c+i));
return 0;
}
```

Soln: c+i here char is involved in an expression so char should be promoted to int

So int + short so short will be promoted

to int

so int + int = int

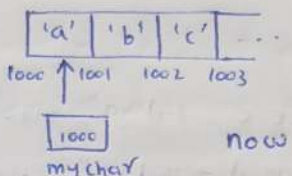
output will be 2 1 4

Pointer arthematic :

char *mychar ;

Here mychar is a pointer pointing to a character

Say

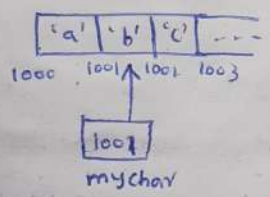


now mychar ++ means we are incrementing the pointer to point to next character i.e 1001 as size of char = 1B

So mychar ++ = mychar + 1(1B)

= 1000 + 1 = 1001

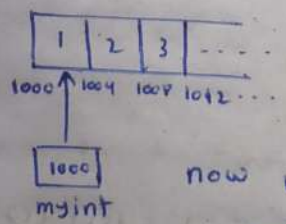
So



int * myint ;

Here myint is a pointer pointing to an integer

Say

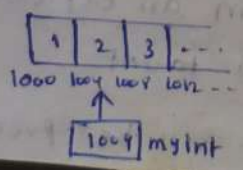


now myint ++ means we are incrementing the pointer to point to next integer i.e 1004 as size of int = 4B

So myint ++ = myint + 1(4B)

= 1000 + 4 = 1004

So



* Let q be an integer pointer

int *q

and q is pointing to address say 2000

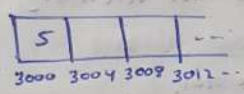
now q++ => q is now pointing to 2004

now (char *) q + 1 will point to ? Assume q = 2000

Here we are typecasting int pointer to

char pointer so q + 1 = 2000 + 1(1B) = 2001

-> int a[6]



now int *P = & a[0]

it is perfectly valid

as we have seen *P contains value

P contains address

* int *P = & a[0]

it doesnt mean we are assigning address

of a[0] to *P.

it means *P is an integer pointer and

P = & a[0]

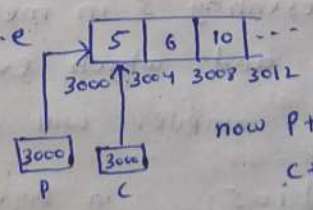
i.e int *P = & a[0] is same as int *P;

P = & a[0];

now

char * C = & a[0]

So C = 3000 i.e



now P+1 = 3004

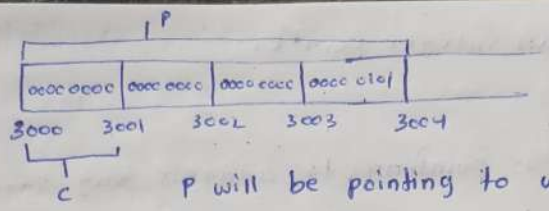
C+1 = 3001

We know memory is Byte addressable *P = 5 now *C = ?

So 5 is stored in memory as 0000 0000 0000 0000 0000 0000 0000 0101

now *C will be having the value of 1st Byte

i.e

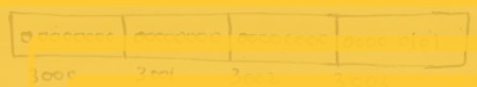


p will be pointing to whole 4B as it is integer pointer

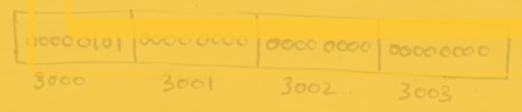
c will be pointing to 3000 byte as c is a char pointer.

So $*c = 0$

→ We have a concept of little endian and big endian way for representing bytes in memory



it is big endian representation



it is little endian representation

most of the systems uses little endian representation

Now Assume

```
int i=5;
p=&i;
```

now the question is what is the value of $*p$ when 5 is represented in big endian and little endian

The answer is that compiler knows which way we are representing 5 in memory so it takes care of the stuff and which ever representation the system is using compiler will always return $*p = 5$

** int i=5; in big endian:

```
char *c;        *c = 0 as first byte has all 0's
c = &i;         in little endian:
                c = 5 as first byte has 5
```

→ Assume short *s; now what will be the values of
int a = 5; *s incase of little endian and
s = &a; big endian

incase of big endian Compiler wont do anything it will just print the decimal value as it is

i.e $\underbrace{00000000 \quad 00000000}_{*s}$ 00000000 00000101

so *s = 0

In case of little endian Compiler will first reverse the bytes pointed by 's' and print the decimal value

i.e $\underbrace{00000101 \quad 00000000}_{*s}$ 00000000 00000000

*s → will be reversed i.e 00000000 00000101

now *s = 5

* Compiler will reverse the bytes not bits so the order of bytes will change as the memory is Byte addressable.

→ #include <stdio.h>

```
int main (void)
{
  int i = 511;
  char *p = (char *) &i;
  printf ("%d", *p);
}
```

Soln: unless specified try to answer in little endian system

511 in binary is

00000000 00000000 00000001 11111111

* char *p = (char *)&i

even if we write

char *p = &i; both of them convey the same meaning as char pointer can only point to 1B

So in little endian

$\underbrace{11111111}_{p}$ 00000001 00000000 00000000

even if we reverse 1B it wont change anything

So printf ("%d", *p) will be -1 as we use %d as format specifier which tells compiler to see 11111111 as -ve number as MSB = 1

→ #include <stdio.h>

```
main()
{
  unsigned int i = 255;
  char *p = &i;
  int j = *p;
  printf ("%d\n", j);
  unsigned int k = *p;
  printf ("%d", k); }
```

Soln: Binary

00000000

* unless

by defau

char *p =

so $\underbrace{11111111}_{p}$

int j = *

Here *p

j

so we

extension

j = 111

we are p

same go

we only lo

k = 1111

So o/p

Pointers and

int *p;

int a[10];

p = & a[0]

now

printf ("%d

Soln: Binary representation of 255 is

00000000 00000000 00000000 11111111

* unless specified try to take little endian system by default even in gate

char *p = &i

so 11111111 00000000 00000000 00000000

p

int j = *p

Here *p = 8 bits

j = 32 bits

so we must use extension

since char *p is signed we use sign extension

j = 11111111 11111111 11111111 11111111

we are printing j as "%d" so o/p will be -1

same goes for k as we know in extension we only look for source

k = 11111111 11111111 11111111 11111111

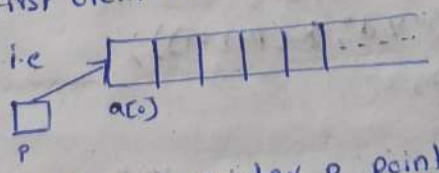
so o/p of k will be '-1'

Pointers and arrays:

int *p; it means p is a pointer pointing to the

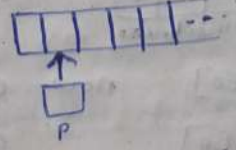
int a[10]; first element in the array

p = & a[0]; i.e



now p+1 = pointer p pointing to a[1]

i.e



now

printf ("%u", a) a is same as & a[0] here a is array name but we can think it as a is pointing

to first element of the array.

So we can write

```
int *p;
p = &a[0];
p = a;
```

Both are same

```
-> #include <stdio.h>
   int main()
   {
    int arr[] = {10, 11, 12, 13, 14};
    int i, *var;
    var = arr;
    for(i=0; i<=4; i++)
    {
      printf("%d\n", *var);
      var++;
    }
    return 0;
   }
```

Soln: it is the code to print all the 5 elements in the array.

```
-> #include <stdio.h>
   int main()
   {
    int a[] = {1,2,3,4,5,6};
    int *ptr = &a+1;
    printf("%d", *(ptr-1));
    return 0;
   }
```

Soln:

a	1	2	3	4	5	6
	2000	2004	2008	2012	2016	2020

2024

char *c = &a[0] now c+1 = 2001 = 2000 + 1(1)

short *s = &a[0] s+1 = 2002 = 2000 + 1(2)

int *p = &a[0] p+1 = 2004 = 2000 + 1(4)

say $p+1 = 2000 + 1(\underbrace{4B})$

↳ it is size of int in array

we can write

$\&a[0]+i = 2000 + i \times (\text{size of } (a[0]))$

similarly

$\&a+i = 2000 + i * (\text{size of } (a))$ if 2000 is starting address of a

so size of (a) = no of elements × 4B

$= 6 \times 4 = 24B$

so $\&a+1 = 2000 + i \times 24$

if $i=1$

$\&a+1 = 2000+24$

$= 2024$

* if p is a pointer pointing to an integer then

$p+1 = p+1(\text{size of int})$

$= p+1(4B)$

if p is a pointer pointing to an array then

$p+1 = p+1(\text{size of array})$

$= p+1((\text{no of elements}) * 4B)$

→ we know

$a = \&a[0]$ so $a+1 = \&a[0]+1$

$= \&a[1]$

Uncovering a[i]: Lec-9 24-08-21

Let us assume 'a' is an array

now

→ a represents address of first element in array

→ a+0 represents address of first element in array

→ a+1 represents address of second element in array

say $p+1 = 2000 + 1(4B)$

↳ it is size of int in array

we can write

$\&a[0]+i = 2000 + i \times (\text{size of } (a[0]))$

similarly

$\&a+i = 2000 + i * (\text{Size of } (a))$ if 2000 is starting address of a

so size of (a) = no of elements x 4B

$= 6 \times 4 = 24B$

So $\&a+i = 2000 + i \times 24$

if $i=1$

$\&a+1 = 2000+24$

$= 2024$

* if p is a pointer pointing to an integer then

$p+1 = p+1(\text{size of int})$

$= p+1(4B)$

if p is a pointer pointing to an array then

$p+1 = p+1(\text{size of array})$

$= p+1((\text{no of elements}) * 4B)$

→ we know

$a = \&a[0]$ so $a+1 = \&a[0]+1$

$= \&a[1]$

Uncovering a[i]: Lec-9 24-08-21

Let us assume 'a' is an array

now

→ a represents address of first element in array

→ a+0 represents address of first element in array

→ a+1 represents address of second element in array

→ a[i] is same as *(a+i)

we know a+i is address of ith element and *(a+i) gives value of ith element.

→ int *pa;

pa = &a[0]; /* or we can write pa = a */

Can we say pa is pointing to array?

We cant say pa is pointing to array but we can say pa is pointing to first element of the array

we can say pa is pointing to array when we write First we need to declare the pointer as pointer pointing to array i.e int (*ptr)[5] → Here PTR is a Pointer pointing to an array of 5 elements

Now ptr = ptr+1 will point to the next array

→ #include <stdio.h>

```
int main()
{
  char string[] = "BADGERS";
  char *ptr = string;
  *ptr = *ptr +2;
  printf("%c", *ptr); ptr = ptr +2;
  ptr--;
  printf("%c", *ptr);
  ptr = string;
  printf("%c", *ptr);
  return 0;
}
```

Sol^n: Since we didn't mentioned size of array "BADGERS" is taken as string

'B'	'A'	'D'	'G'	'E'	'R'	'S'	'\0'

↑ ptr

*ptr = *ptr +2

= 'B' + 2 = 'D', ptr = ptr+1

'D'	'A'	'D'	'G'

↑ ptr

*ptr will p

ptr--

'D'	'A'	'D

↑ ptr

*ptr will

ptr = strin

*ptr will

so DAD will

→ we know

a[i] = *

2[a] =

Now if p is pointing to p then

p[i] = *

Array vs pointe

int a[5]

int *p =

now p[

the exact

array 'a' i

→ we c

i.e a

but p

→ we can dire without any

'D'	'A'	'D'	'G'	'E'	'R'	'S'	'\0'

↑ ptr

*ptr will print 'D'

ptr - -

'D'	'A'	'D'	'G'	'E'	'R'	'S'	'\0'

↑ ptr

*ptr will print 'A'

ptr = string; so ptr points to first element

*ptr will print 'D'

so DAD will be printed.

→ we know

$a[i] = *(a+i)$

$2[a] = *(2+a) = *(a+2) = a[2]$

Now if P is an array pointer i.e a pointer pointing to first element of an array then

$P[i] = *(P+i)$ and $a[i]$ is equivalent to $P[i]$

Array vs pointer:

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

now $P[2] = 3 = a[2]$

The exact difference between Pointer P and array 'a' is,

→ we cant change the base address of 'a'

i.e $a = a+1$ is invalid

but $p = p+1$ is valid

→ we can directly assign a pointer to a string without any effort

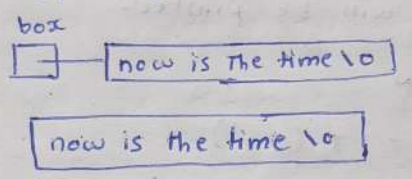
```
char c[] = "now is the time";
char *t = c
```

```
char *a = "now is the time"; [it is valid]
```

Earlier we have seen that array name should not be present to the left of the assignment operator because

```
char amessage[] = "now is the time";
char *pmessage = "now is the time";
```

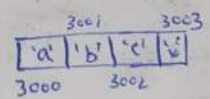
Diagramatically speaking



in pmessage we have a box/space to store what we assign to it so we can write it on leftside of the assignment

in amessage we dont have any box/space to hold that we want to assign so we cant have an array to the left side of the assignment.

So `char s1[4] = "abc";`




```
char s2[4];
s2 = s1; /* ERROR */
```

Here s_1 is address which is 3000 and we cant assign address to an array.

is valid]

name should
assignment

ne";

;

o store what
it on leftside

space to hold
have an array

array.

(*ptr) = &(3000) = address of element value 5
= 2000

&(&ptr) = *(3000) = 2000

so in case of pointer if P is a pointer then

&*ptr = *&ptr

(&, * will get cancelled)

→ int array[] = {4, 6, 12, -5, -7, 3, 1, 0, -10};

int *ptr1, *ptr2;

ptr1 = array + 2;

ptr2 = &ptr1[5];

① what is the value of *(ptr1 + 1)

ptr1 will point to element 12 as ptr1 = array+2

now *(ptr1+1) = ptr1[1] = -5

② what is the value of *(ptr2 - 3)

ptr2 will point to element 0 as ptr2 = &ptr1[5]

ptr1[5] = *(ptr1 + 5) = 0

now *(ptr2 - 3) = -7

→ Let int a[6] = {1, 2, 3};

int *p = a;

now size of (a) = 6 × (size of (int))

= 6 × 4 = 24B

size of (P) = size of (int) (as p is integer pointer)

= 4B

Gate CSE 2019:

→ Consider the following C program

```
#include <stdio.h>
int main() {
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 5};
int *ip = arr + 4; printf("%d\n", ip[1]); return 0; }
```

Soln: ip = arr + 4

So *ip = 5

ip[1] = *(ip+1)

= 6

Gate CSE 2011 :

→ what does the following c program print?

```
char c[] = "GATE2011";
char *p = c;
Printf ("%s", p + p[3] - p[1]);
```

A) GATE2011 B) E2011 C) 2011 D) 011

Soln: %s will print the characters till it sees null

since c[] size is not mentioned we have null at the end

'G'	'A'	'T'	'E'	'2'	'0'	'1'	'1'	\0

p[3] = *(p+3) = 'E'

p[1] = *(p+1) = 'A'

So 'E' - 'A' = 4

Printf ("%s", p+4); = 2011

→ * Let us Assume we have a System where address is 32 bit or 4B address

then

Sizeof int *a = 4B

sizeof char *ch = 4B

→ This is because integer pointer or character pointer both have to store address right and since address is 4B sizeof char *ch = 4B

Similarly size of float *f = 4B

Size of long int

Size of short i

Size of double

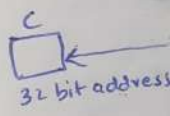
→ char c;

char *ch = &c

sizeof (c)

sizeof (*

i.e



Passing 1D arr

int main

{

int a

fun

}

in main fu

the array w

of the array

Now

the function

* For us

fun (int b

array

But c

fun (int b

to fun (

s

to fun (i

the array

size of long int *li = 4B

size of short int *st = 4B

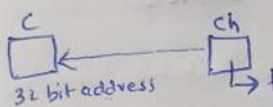
size of double *d = 4B

→ char c;

char *ch = &c;

sizeof (c) = 1B as char is 1B

sizeof (*ch) = 4B if the system is generating 32 bit addresses

i.e 

→ to store 32 bit address size of *ch should be 4B

Passing 1D array to a function:

```
int main ()                    fun (int *p)
{                              {
 int a [3] = { 1, 2 };          ----
 fun (a);                      }.
}
```

in main function we are passing the address of the array which is the address of starting element of the array

Now to catch this we need a pointer in the function as pointer is able to handle the address

* For user convinence we can also write fun (int b [3]) or fun (int b [ ]) to catch the array

But even if we write fun (int b [3]) or fun (int b[ ]) compiler internally converts them to fun (int *a)

Since compiler is converting fun (int b[]) to fun (int *a) we can use whatever size in the array i.e b [10] or b [1000] also we dont need to

mention the size also i.e int b[] in the function Parameter.

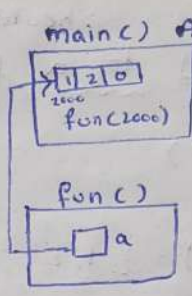
So

```
void fun1 (int *a)
{
}
void fun2 (int a[])
{
}
void fun3 (int a[10])
{
}
```

All of them mean the Same to catch an array starting address

→ main ()

```
{
  int a[3] = {1,2};
  fun (a);
}
fun (int a[3])
{
}
```



when we pass the address of array to fun() in the Activation record of fun() we only have a pointer Pointing to Starting address of array which is 2000

→ is the Program below valid

```
main ()
{
  int a[5] = {1,2}
  fun (a);
}
```

```
fun (int a[10])
{
  a = a+1;
  *a = 10;
}
```

Soln: in fun ( ) we dont have an array, we only have a pointer pointing to Starting address of array so a = a+1 is valid because a is a pointer but if we write a = a+1 in main() it is invalid as a is an array in main() so array cannot be present to the left of assignment

→ if we write
char c[
scanf ("
and we i
then s
sees null
so H
c
This t
malloc () w

strlen ( ) :
strlen
return the
Now let

main ()
{
char c[]
strlen (c
}

Now Let us
happening h
ma
H e l
at i=0 c
i=1 c
i=2
i=3
i=4
i=5
now len

if we write

```
char c[5];
scanf("%s", c);
```

and we input Hey

then scanf continous to take input until it sees null

so

H	e	y	\0	\0

c

This thing gets clarified when we study malloc() which is dynamic memory allocation.

strlen ( ) :

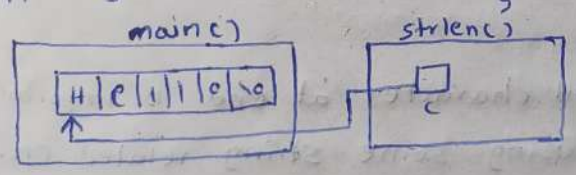
strlen() takes string as input and will return the length of the string

Now let us see the function for strlen()

```
main()
{
 char c[] = "Hello";
 strlen(c);
}
```

```
int strlen(int c[] or
           char ← int c[10] or
           int *c)
  ↳ it is actually char not int
{
 int i=0
 while (c[i] != '\0'){
   i++;
 }
 return i;
}
```

Now let us see what is happening here



at i=0 c[0] = H ≠ \0

i=1 c[1] = e ≠ \0

i=2 c[2] = l ≠ \0

i=3 c[3] = l ≠ \0

i=4 c[4] = o ≠ \0

i=5 c[5] = \0 = \0 So loop will break here

now length of string "Hello" = 5

→ We can implement strlen() using char pointer too

i.e int strlen (char *c)

```
{
    int i=0;
    while (*c!='\0')
    {
        i++;
        c++;
    }
    return i;
}
```

strlen() function in dennis Ritche is as follows

```
int strlen (char *s)
{
  int n;
  for (n=0; *s!='\0'; s++)
    n++;
  return n;
}
```

→ strlen() is already defined in c so we dont need to define it again.

null character at the end of string is very important because if we want to calculate the length of string we will parse till we see the null character

→ when having null character at end is important

if we are using some string related operations like strlen(), strcpy() then it is important

→ when having null character at end is not important

when we are treating character arry as just a character array not as string and not using any string operation

* → printf ("%d", strlen ("Hello"));

the o/p will b

character H

character wont

→ printf ("%d", size

in the

| H | e |

so in total

o/p will be '6

→ write a func

Let char c[]

char d[]

/* Function to

strcpy (char

{

while (*d

{

*c = *

d++;

c++;

}

}

Gate 2004 :

→ char p[20]; int

Char *s = "Str

int length = s

for (i=0; i<

P[i] = s[le

Printf ("%s

the o/p will be 5 as when we reach the null character the loop will break so the null character wont get counted

→ printf ("%d", sizeof ("Hello"));

in the memory Hello will be present as

H	e	l	l	o	\0

so in total there are 6 characters so the o/p will be '6'

→ write a function to copy one string to another

Let
```
char c[] = "LoL";
char d[] = "Hello";
```

/* Function to copy c to d */ or d to c anyone is fine

```
strcpy (char *c  char *d)
{
  while (*d != '\0')
  {
    *c = *d;
    d++;
    c++;
  }
}
```

Gate 2004 :

→
```
char p[20]; int i;
Char *s = "string";
Int length = strlen(s);
for (i=0; i< length; i++)
  p[i] = s[length - i];
Printf ("%s", p);
```

The output of the program is

A) gnirts B) string

c) gnirt D) no output

Soln: we have

s	t	r	i	n	g	\0

↑ s

strlen(s) = 6

so for loop will run from i=0 to i=5

P[0] = S[6-0]

P[0] = S[6] ⇒ P[0] = \0

P[1] = S[5] ⇒ P[1] = g

P[2] = S[4] ⇒ P[2] = n

P[3] = S[3] ⇒ P[3] = i

P[4] = S[2] ⇒ P[4] = r

P[5] = S[1] ⇒ P[5] = t

So P[0] = {'\0', 'g', 'n', 'i', 'r', 't'};

when we use '%s' it will stop printing when it sees a null character

so first character is null character

There will be no output that will be printed

Character string vs character pointer:

Char str1[] = "Hello world!";

char * str2 = "Hello world!";

* str2 is called string constant and it will be stored in Read only memory (Rom) present in static area of the memory

it means we can read the string const but we cant write to it

i.e str1[1] = 'u'; it is valid

str2[1] = 'u'; it is invalid - May crash the Program

Pointer Arthemati

int a[10];

int * P1 = a;

int * P2 = P1 +

Printf ("%d"

1	2	3	4	5

2004 2012 2000 2008 2016

p q

now q-p = 2012

= 12 /

= 12 /

* Valid arthem

P1 + 3 - A

P1 - 3 - S

p - q - D

p<q, p>q,

* Invalid arthem

P1 + P2

P1 * P2

P1 % P2

P1 / P2

** when two

Possible wh

Same array

if not

value or er

Pointer Arthematic (difference between two pointers):

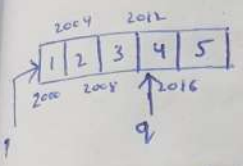
```
int a[10];
int *P1 = a;
int *P2 = P1+3;
printf ("%d", P2-P1);
```

we know

$P_2 = P_1 + 3$

$\Rightarrow P_2 - P_1 = 3$

it means the difference between two pointers is no of elements between them



now $q - p = 2012 - 2000$

$= 12 / \text{size of (int)}$

$= 12 / 4 = 3$

* Valid arthematic operations on pointers:

$P_1 + 3$ — Addition

$P_1 - 3$ — Subtraction

$p - q$ — Difference between two pointers

$p < q, p > q, \ldots$ — Comparision operations

* Invalid arthematic operations on pointers:

$P_1 + P_2$ — $P_1 + P_2$ is not allowed because

Lec-10 25-08-21

Double pointer :

Double pointer is a pointer which will point to another pointer

```
int x, *P1, **P2;
x = 100;
P1 = &x;   /* address of x */
P2 = &P1;  /* address of P1 */
Printf ("%d", **P2);
```

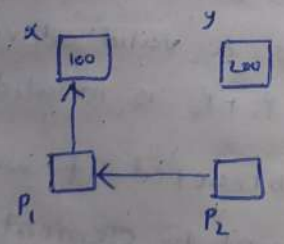
→ we can access the value of x using x, $*P_1$, $**P_2$

for triple pointer we use $***p$

→ what will be the output

```
main()
{
 int x, y, *P1, **P2;
 x = 100; y = 200;
 P1 = &x ; P2 = &P1 ;
 Printf ("%d", **P2);
 *P2 = &y;
 Printf ("%d", **P2);
}
```

Sol^n:



$**P_2$ will print 100

now

$*P_2 = \&y \Rightarrow P_1 = \&y$

so P_1 is pointing to y

so $**P_2$ will print 200

Gate 2008

→ what will be the output

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

int f (int

{ int y, z

** ppz

*py + =

x + = 3

return

}

void main

{

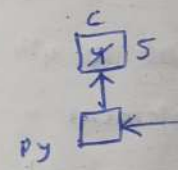
int c, *

c = 4;

Printf

}

Sol^n:



Po

2D arrays :

int

we

o

we kno

Similarly

Lec-10 25-08-21

which will point

using

*p

100

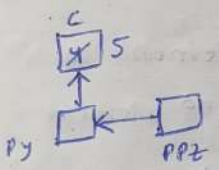
&y

to y

nt 200

```
int f(int x, int *py, int **ppz)
{ int y, z;
  **ppz += 1; z = **ppz;
  *py += 2; y = *py;
  x += 3;
  return x+y+z;
}
void main()
{
  int c, *b, **a;
  c=4; b=&c; a=&b;
  printf("%d", f(c,b,a));
}
```

Soln:



$**ppz = 4+1$, $z = 5$
$= 5$

$*py = 5+2$; $y = 7$
$= 7$

$x = 7+3 = 10$ so $x+y+z = 10+7+5$
$= 22$

it is wrong

Because $x = 4$ as it is the value not the pointer so final value $= 4+5+7+3 = 19$

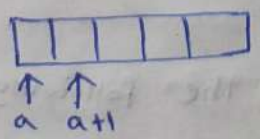
2D arrays:

int a[4][3] is 2D array

we can relate 2D array to a matrix as

a[4][3] 'a' is having 4 rows and 3 coloumn

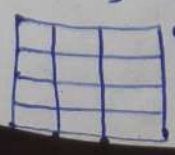
we know in 1D array



Here a represents the address of a[0]

a+1 represents the address of a[1]

Similarly in 2D array



a[4][3]

a represents one full row

a+1 represents second full row

Similarly $P+4$ will represent 4th row / 5th row if array starts with 0 index

Say we want to get into fourth row i.e inside the fourth row

$(P+4)$ will represent / point to entire 4th row

now inorder to point to only first element in the 4th row then we use $*(P+4)$ now we are pointing to the single element i.e first element in 4th row

** $P+4$, $*(P+4)$ both point to same address location but pointer arthematic changes

i.e $((P+4)+1)$ will point to 5th row / 6th row if array starts with 0 index

$*(*(P+4)+1)$ will point to 2nd element in 4th row and we can access the 2nd element

So $*(P+4)+1$ will have address of 2nd element in 4th row / 5th row if array starts with 0 index

* one trick is, in 2D array we need to have two '*'s to get the value

2000	2004	2008	2012
1	2	7	9
3	10	-1	2
0	6	8	4
3	11	13	-4

now $P = 2000$

$P+1 = 2016$ $\quad *(P+1) = 2016$

$(P+1)+1 = 2032$ $\quad *(P+1)+1 = 2020$

* $P+1$ will be pointing to 1D array

* $*(P+1)$ will be pointing to one integer

Gate CSE 2015:

→ what will be the output of the following c-code

Assume address of $x = 2000$ and integer requires 4B of memory

```
int main ()
```

```
{
unsigned int x[4][3] =
        {{1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}};
printf("%u, %u, %u", x+3, *(x+3), *(x+2)+3);
}
```

A) 2036, 2036, 2036 B) 2012, 4, 2204 C) 2036, 10, 10

D) 2012, 4, 6.

Soln: $x = 2000 \Rightarrow x+1 = 2000 + 3(4B)$

$= 2012$

$x+3 = 2000 + 9(4B)$

$= 2036$

$*(x+3) = 2036$, $(*(x+2)+3) = 2000 + 6(4) + 3(4)$

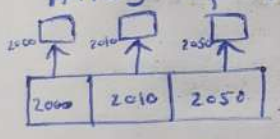
$= 2000 + 6(4B) + 3(4B)$

$= 2036$

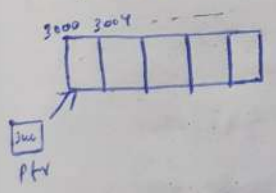
Array of pointers vs pointer to array:

↳ int *x[] ↳ int (*y)[]

Array of pointers will look like



Pointer to array will look like



Pointer to array:

```
int main()
{
  int (*a)[5];
  int arr[5][5] = {{1,2,3,4,5}, {6,7,8,9,10}};
  a = arr;  ++a;
  printf("%d", **a);
}
```

Soln: int (*a) [5];

it means a is a pointer pointing to an array of 5 elements

now let arr = 2000 and a = arr ⇒ a is pointing to first row

now ++a will goto second row of arr

now **a ⇒ *(*a)

↳ we went into second row so it became pointer to integer

now **a will print first element in second row

So 6 gets printed

* Even if we write

a = *arr then also we dont get any error as arr = 2000

*arr = 2000 but here the arthematic of arr changes

if a = *arr

Then arthematic of 'a' doesnt change and a++ will goto next row

now *

→ #include <st

int main ()

{

int (*p) [4]

int a [4] =

p = &a;

for (int i =

printf ("

return 0;

}

Soln: here p

now p =

now *p

so *(*

Here

Array of poin

int a

int *

The differ

int *

in in

Priority tho

a will be

in (*a

but left

a pointer

* if we

now $*(p+v-1) = *(\&a[5])$
$= 6$

```
#include <stdio.h>
int main ()
{
  int (*p) [4];
  int a[4] = {1,2,3,4};
  p = &a;
  for (int i=0; i<4; ++i)
    printf ("%d\n", *(*p +i));
  return 0;
}
```

Ptr to arr p
↓
int *p
↓
actual val **p

Soln: here p is a pointer to an array

now $p = \&a \Rightarrow p+p+$ = pointer pointing to next array address

now *p will make pointer p as pointer to integer

so $*(*p+0) = 1$, $*(*p+1) = 2$ ....

Here we are printing the array elements.

Array of pointers:

int a[10] is normal array

*a[10] is array of pointers

a[0] to a[9] are elements

int *p[10] is array of pointers

*p[0] to *p[9] are elements

if we are dealing with pointer to an array

then it is important to know the size of the array which we are pointing to

i.e int (*p) [5]

Here p is a pointer pointing to array of 5 elements and when we do p++ the pointer will skip 5 elements and point to 6 th element if present

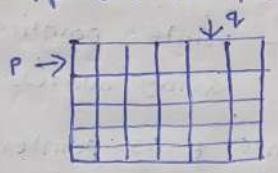
if we have

int (*p) [4]

int arr [5] [6]; p = arr;

then

if we do p++ let q = p++

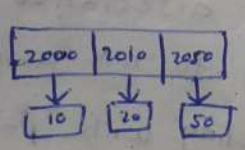


So size of the array helps us to skip how many elements to perform p++

```
-> int main()
   {
    int * arrop[3];
    int a=10, b=20, c=50, i;
    arrop [0] = &a;
    arrop [1] = &b;
    arrop [2] = &c;
    for (i=0; i<3; i++)
    {
       Printf (" Address = %u\t value = %d\n", arrop[i],
                                                 * arrop [i]);
    }
    return 0;
   }
```

Soln: * arrop [3] => arrop is an array of 3 pointers

Say



So at i=0 : o/p 2000 10

i=1 : 2010 20

i=2 : 2050 50

-> char * sports [5]

sports [i][0]

Soln: char * spor

Here spor

pointing to s

So

sport[0]	1000
sport[1]	1005
sports[2]	1012
sports[3]	1021
sports[4]	1029

sports [i][0]

sports [3] [3

-> char *a = "g

a[0] = 't';

is the above

Soln: it is not o

which will

so we can

Passing 2D arre

when

Then write i

as main()

{ int a[s

} fun (a

e of the array

to array of
the pointer
6 th element

array helps
any elements

arrop[i],
* arrop[i]),

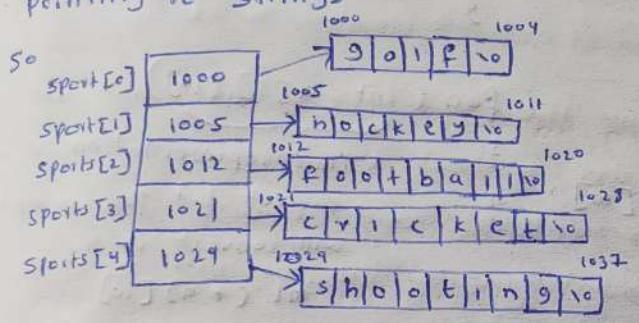
3 pointers
o/p
2000 10
2010 20
2050 50

→ char *sports[5] = {
"golf",
"hockey",
"football",
"Cricket",
"Shooting"
};

sports [1][0] = ?

Soln: char * sports [5]

Here sports is an array of character pointers pointing to strings



sports [1][0] = * (1005 + 0)
= h

sports [3][3] = * (1021 + 3)
= c

→ char *a = "golf";
a[0] = 't';
is the above assignment ok or not?

Soln: it is not ok because "golf" is a string constant which will be present in Rom in static area so we cant modify string constant

Passing 2D array to function:

When we pass a 1D array to a function Then write int *a in parameter list of function as

```
main()
{ int a[5];
} fun(a);

void fun (int *a)
{
}
```

in fun (a)
↳ 'a' acts as pointer to an integer so we write int *a in parameter list

in case of 2D arrays:

fun (a)
↳ 'a' will act as pointer to 1D array so we will write int (*a) [5] in parameter list

i.e

```
main()                    void fun ( int (*a) [5])
{                         {
  int a[4][5]               ...
  fun (a);                  ...
}                         }
```

→

```
main()                    fun ( int a[5] [10])
{                         {
  int a[5] [10];          }
  fun (a)
}
```

This representation is also correct but the compiler will convert it to int (* a) [10]

other representations are

fun (int a [ ] [ ]) ×

Void pointer :

when we assign cross referencer pointer without typecasting then compiler will raise warning

i.e say

```
char c[5] = "abcde";
int *p = c;
```

Since c is char array and p is int pointer compiler will raise warning but execute the compiler without error

if we use void pointer then compiler wont raise warning

i.e

```
int main()
{
  int a = 7;
  void *p;
  p = &a;
  Printf("Int variable is %d", *((int*)p));
}
```

in void pointer typecasting the pointer is very much important while dereferencing because if we simply write

printf("variable is %d", *p);

then compiler doesn't know upto how much we have to dereference

i.e if we have int variable then 4B
char variable then 1B

Since we wrote void *p we have to specify how many Bytes to dereference while printing so type cast is compulsary.

Heap memory :

Malloc :

Malloc is the short form for memory allocation

we use malloc to create memory during the run time

malloc will return the starting address of the newly created memory

to be precise malloc will return the pointer pointing to the starting address of the memory. malloc will be used in the function as

→ int *ip;

ip = (int *) malloc (size of (int) * 10); /* Allocate 10 integers */

ip[6] = 42; // set 7th element to 42

→ malloc returns the void type pointer and we use this pointer without typecast.

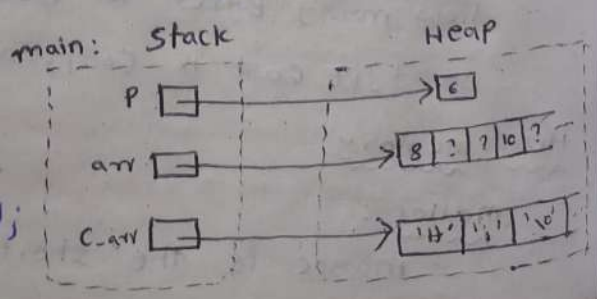
→ malloc returns void because malloc doesn't know for what purpose this memory is being used

→ when malloc fails to create memory then it will return Null

→ Sample program for malloc

```
int *p, *arr;
char *c_arr;
p = (int *) malloc (size of (int));
arr = (int *) malloc (size of (int) * 10);
c_arr = (char *) malloc (size of (char) * 5);
if ((p == NULL) || (arr == NULL) || (c_arr == NULL))
    printf (" malloc error \n");

*p = 6;
*arr = 8;
arr [3] = 10;
strcpy(c_arr, "He");
c_arr[1] = 'i';
```



Free ( ) :

Lec-11 25-08-21

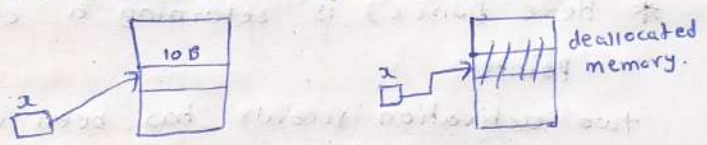
Free ( ) is used to release / free up the used space

Let x = malloc (10)

now we are allocating 10 B of memory and we give the starting pointer to 'x'

if we use free (x) then we are deallocating the 10 B memory and now this deallocated memory can be allocated to some other task.

x = malloc (10) now free (x)



* when we free up the memory, pointer x will still point to starting address of the memory

we know going to the unknown address is not a problem

i.e say int a[10];

int *p; This wont be a problem

p = a+50; as we can goto whatever address we want

the problem occurs when we want to access the data in that unknown location

like (*p) if p = a+50

it is like standing outside celebrity's house is not a problem but entering into the house creates the problem

Dangling pointer :

Dangling pointer is a pointer that points to memory that has already been freed

in the above scenerio x is a dangling pointer

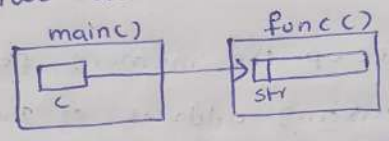
as x is still pointing to the memory that has already been freed.

other example can be

```
char * func()
{
  char str[10] = {'H', 'e', 'l', 'l', 'o'};
  return str;
}
main() {
  char *c;
  c = func();
}
```

* Here func() is returning a character pointer

two activation records has been created



when func() returns the activation record gets destroyed so c is pointing to unknown location so we can say Dangling pointer is a pointer which points to the location that we dont have access to

→ int *c = malloc (size of (int));
free (c);
*c = 3; // writing to the freed location

here were still trying to access the location that had been freed so c is a dangling pointer

memory leak:

memory leak is a memory which hasn't been freed, there is no way to access it (or free it) now

now let us see an example for memory leak

main()
{
int *x;
x = (int *)
x[1] = 3;
int a = 3;
x = &a;
}

Here
is wron
address
now x

now

There
in the
memory

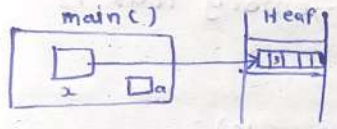
let us see
main()
{
mal
int
int
x
}

→ is there
main()
{
int *
x =
x =
}

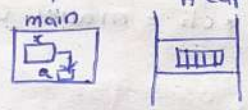
```
main ()
{
int *x;
x = (int *) malloc (sizeof(int) * 4);
x[1] = 3;
int a = 3;
x = &a;
}
```

Here x is pointing to an array of integer is wrong convention, x is pointing to starting address of the array

now x [1] = 3 so we put second element = 3



now x = &a ⇒



There is no way to access the memory or array in the Heap so we cant free it up this is called memory leak

let us see another example

```
main ()
{
  malloc (10);
  int *x;
  int a = 3;
  x = &a;
}
```

This is also memory leak as malloc ( ) will return void pointer but we are not catching it so we cant access the newly created memory which results in the memory leak.

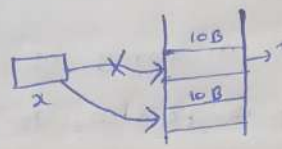
→ is there a memory leak in the given code

```
main ()
{
  int *x;
  x = malloc (10);
  x = malloc (10);
}
```

Soln: initially we are creating memory of 10B and x is pointing to it

now we are again creating memory of 10B and again we are using x to point to memory

so we cant access the first created memory it means there is a memory leak.



*"memory leak is crime so dont do it"

one best example of memory leak is

```
void func () {
    char *ch = malloc (10);
}
main () {
    func ();  // ch is not valid outside func () so
              // no way to access malloced memory
```

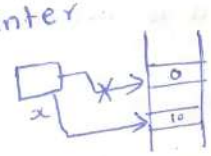
Gate CSE 2017

→
```
# include <stdio.h>
int * assign val (int *x, int val)
{
    *x = val;
    return x;
}
void main () {
    int *x = malloc (sizeof (int));
    if (NULL == x) return;
    x = assign val (x, 0);
    if (x) {
        x = (int *) malloc (sizeof (int));
        if (NULL == x) return;
        x = assign val (x, 10);
```

```
}
printf("%d\n", *x);
free(x);
}
```

is there memory leak or Dangling pointer

Sol^n: function assign val ( ) is returning an integer pointer



Here we are assigning newly created memory pointer to x which is already pointing to dynamically created memory

So clearly it is a memory leak.

Gate CSE 2001

→ [P_1]

```
int *g [void]
{
  int x = 10;
  return (&x);
}
```

[P_2]

```
int * g(void)
{
  int *px;
  *px = 10;
  return px;
}
```

[P_3]

```
int *g (void)
{
  int *px;
  px = (int *) malloc (sizeof (int));
  *px = 10;
  return px;
}
```

which one of the functions are likely to cause problem with pointer

A) only P_3 B) only P_1 and P_3 C) only P_1 and P_2 D) P_1, P_2, P_3

Sol^n: In P_1 x is local variable and after returning from P_1 x will be gone so in main( ) we catch the address of x will cause dangling pointer

we are defining a pointer

*px

and we directly assigning value to it

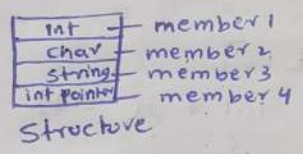
it will cause error because when we declare a pointer it will initially point to unknown location and writing to unknown location is not allowed

P_3 wont cause a problem.

Structures in C-Programming :

Logically speaking structure is packing different types of variables in one box

i.e



Structure is used for

→ handling a group of logically related data items

i.e student name, roll number, marks

Real part and Complex part of a Complex number

→ Helps in organizing complex data in a more meaningful way

→ The individual structure elements are called members

* Just like arrays, members in the structure are consecutive in memory.

The idea

we want to describe a box so for that we need width, length, height

for this we can have 3 variables but wouldn't it be better if we have a single variable to describe a box ?

That variable can have three parts width, length and height.

So we can write

```
#include <stdio.h>
struct box1
{ int width;
  int height;
  int length;
};
struct circle
{
  double radius;
}
int main()
{
  struct box1 b;
  struct circle c;
}
```

here b is a varible of type struct box1

Just like a is a varible of type int in int a;

```
→ struct book_bank
  {
    char title[20];
    char author[15];
    int pages;
    float price;
  }; → This semicolon is easy to forget so always remember to put it
  struct book_bank book1, book2, book3;
```

Here we have 3 variables book1, book2, book3 of type struct book_bank.

Declaring struct variables :

there are two methods for declaring struct variables

now we will see both of them.

method-1

```
struct employee
{
    int id;
    char name[50];
    float salary;
};
struct employee e1, e2;
```

method-2

```
struct employee
{ int id;
  char name[50];
  float salary;
} e1, e2;
```

typedef:

Typedef is a way in c to give a name to a Custom type

i.e if we write

```
typedef int dollars;
```

then in the program we can write dollars in place of int

so int array[10]; ≅ dollars array[10];

like wise

```
typedef unsigned char Byte;
unsigned char Byte1; ≅ Byte Byte1;
```

in general

```
typedef type newname;
```

↳ This can be int, float, double .... whatever data type that is supported in C.

* typedef is only used for data types.

typedef is more often used struct datatype

i.e

```
struct bk
{
    char a;
    int book_id;
};
typedef struct bk Books;
Books Book1, Book2;
```

Also we can

typedef struc

char a;

int book_

} Books;

Books Book1,

Accessing struc

Given

using . oper

struct

int a

long i

};

struct

r1.i =

Array of S

we C

struct

{

char

floa

int

};

struc

So

type

i.e

→ Sample pr

#includ

struct

{

cha

Also we can write typedef as

```
typedef struck bk {
    char a;
    int book_id;
  } Books;

Books Book1, Book2;
```

Accessing structure members :

Given a struct we can access its members using . operator

```
struct rec {
  int a[4];
  long i;
  };
struct rec r1;
r1.i = 10;
```

Array of structures :

we can define array of structures as

```
struct book
{
  char name;
  float price;
  int pages;
};
struck book b[10];
```

So here each element in the array is of type struct book

i.e

Pages	pages	- - -
Price	Price	
name	name	
b[0]	b[1]	

→ Sample program for array of structures

```
#include <stdio.h>
struct person
{
  char name[20];
```

```
    int age;
};
struct person p[3] = { {"mike", 29}
                       {"John", 33}
                       {"Kate", 20} };
```

So here

p[0].name = mike , p[0].age = 29

p[1].name = John , p[1].age = 33

p[2].name = Kate , p[2].age = 20

Pointer as member of struct:

we can use pointer as just another member of struct

i.e

```
struct test
{
    char name[20];
    int *ptr;
};
struct test t1;
```

now if we want to access character in array then we use t_1.name[2] if we want 3rd character

Say ptr is pointing to variable 'a' [5] a 2000

now if we want to access value of a then

$*(t_1.ptr)$ as $t_1.ptr = 2000$ (address of a)

$*(t_1.ptr)$ = value of a

Since . is having higher priority than * we can write $*t.ptr \cong *(t.ptr)$

Lec-12 26-08-21

→ what will be the output of the following program

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

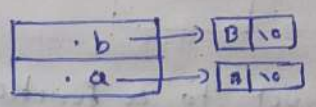
```
typedef struct {
    char *a;
    char *b;
  } t;
void f1(t s);
main( )
{
 t s = {"A", "B"};
 printf ("%s %s", s.a, s.b);
 f1(s);
 printf ("%s %s", s.a, s.b);
 printf ("%c %c", *s.a, *s.b);
}
void f1(t s)
{
   s.a = "U";
   s.b = "V";
   printf ("%s %s", s.a, s.b);
}
```

Sol^n: Here s is a variable of type struct

now ts = {"A", "B"}

it means s has two char pointers pointing to string constants "A", "B"

i.e



s.a is having address of "A"

s.b is having address of "B"

* "%s" needs the starting address of the string and it will print until it sees '\0' in the string

Earlier we have seen a question where

```
main()
{
 char c[] = "GATE 2022";
 char *p = c;
 printf("%s\n", p+p[3]-p[1]);
}
```

Here $p[3] = 'E'$ so $p[3] - p[1] = 4$

$p[1] = 'A'$

$\underbrace{p+4}$

↳ it is an address

So %s will start printing from p+4

" %s expects address of the string"

Comming back to the question

printf("%s %s\n", s.a, s.b) will print A B

* since "A", "B" are string constants

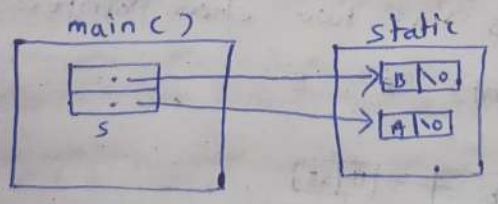
char *t = "Hello" is valid because

t is expecting address

and assigning the string constant "Hello"

means we are assigning address of H to

t so this is legal assignment



→ Now Passing struct to a function

it is Just like passing a variable to a function

i.e

```
main()        foo(int b)
{             {
 int a;
 foo(a);      }
}
```

now in arrays

say a[5] name of array acts like a pointer to the array

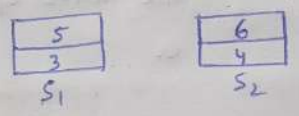
i.e $*(a+3) = a[3]$

but in case of structures it wont be the case i.e name of the structure wont act like pointer to the structure.

Also if we have two structures of type t say S_1, S_2

now $S_1.i = 3$ $S_1.j = 5$

$S_2.i = 4$ $S_2.j = 6$

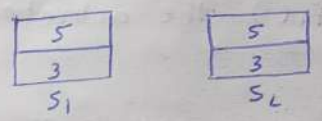


performing

$S_2 = S_1$; is perfectly valid incase of structures

so S_1 will be completly copied to S_2

i.e

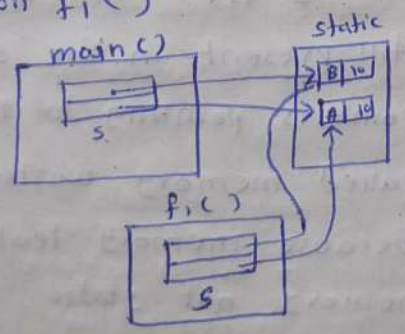


* incase of arrays this wont be happening as $a_2 = a_1$ is invalid because array shouldnot be on the left side of the assignment operator.

Now

$f_1(s)$ means we are sending structure s to function $f_1()$

so



* we cant change the content of the structure s in

main( ) using f_1( ) as only a copy s is present in f_1( )

it is same as

```
main ( )        foo(int a)
{               {
 int x;         }
 foo (x);
}
```

Here whatever we do to a wont reflect on x

But this can change when we pass address to the function

i.e

```
main ( )        foo (int *a)
{               {
 int x;         }
 foo (&x);
}
```

now we can change the value of x using a

in f_1( )

s.a = "u"; So Printf will print u v
s.b = "v";

after returning from f_1( ) the activation record of it will get destroyed

so Prinf in main again will print A B

Printf ("%c", *s.a) will print A

Since s.a is address we need *s.a to get the value

* Even after returning f_1( ) the string constants "u", "v" are still present in the static area and we lost pointers pointing to them

* This is called memory wastage not the memory leak because memory leak is related to dynamic memory not static memory

* struct names

i.e struct exa
int foo;
int bar;
};
struct ex
Here e

Pointer to struc
struct
{
int
int
int
};
struc
Here Pb
*PB

Point

→ Fill the blank

struct Box
{
int width,
int length
int height
}
int main( )
{
struct
struct
PBox =
b. wid
Printf
}

* struct names are not pointers to anything

i.e struct example {

```
    int foo;
    int bar;
};
struct example e;
```

Here e is not a pointer to anything.

Pointer to struct:

```
struct Box
{
    int width;
    int length;
    int height;
};
struct Box Pbox, *PBox;
```

Here Pbox is a variable of type structBox

*PBox is a pointer of type structBox

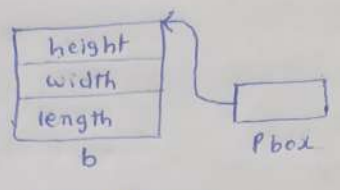
Here PBox is not struct, it is just a pointer pointing to struct

→ Fill the blank to print width of box using PBox

```
struct Box
{
  int width;
  int length;
  int height;
}
int main()
{
    struct Box b;
    struct Box *PBox;
    PBox = &b;
    b.width = 3;
    printf("%d", ____);
}
```

Soln: PBox is not a structure it is a pointer to a structure

So



Given b.width = 3

we know in pointers

int a;

int *b = &a;

now we can use *b in place of a

likewise PBox is pointing to b

so we can use *PBox inplace of b

we can write (*PBox).width = 3 to access b.width.

Here ( ) → Paranthesis is necessary as *PBox.width . has high precedence over *

it will give error as PBox is not a structure so PBox.width is invalid.

* Always writing paranthesis for a pointer will be cumbersum so we use arrow operator

i.e (*PBox).width ≅ PBox → width

* Arrow operator is used for struct pointer type variable

Arrow operator for struct pointers:

once PBox points to a structure variable the members can be accessed as

PBox → width // same as (*PBox).width

PBox → length // same as (*PBox).length

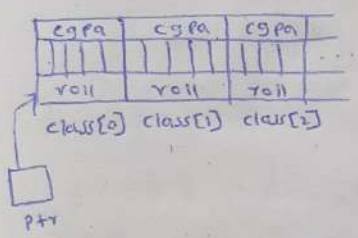
PBox → height // same as (*PBox).height

This method is much more redable and intutive

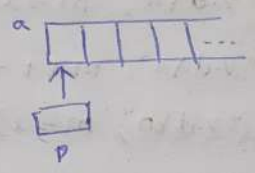
→ structures are just like integers

```
struct stud {
 int roll;
 char dep_code [25];
 float cgpa;
} class [10], *ptr;

ptr = class;
```




```
int a[10], *p;
p = a;
```



to access cgpa of 2nd element of array class[1]

we can use class[1].cgpa (or)

(++ptr) → cgpa

→ when the pointer ptr is incremented by one (++ptr)

→ The value of ptr is increased by sizeof(stud)

→ it is made to point to the next struct in the array

→ [illegible]

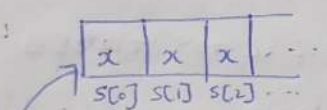
so ++ptr → roll increments roll not ptr

as → has high precedence than ++

→ struct student
{
 int x;
};

```
Void main( )
{
   Struct student s[10];
   s[0].x = 10;
   s[1].x = 1;
   Struct student *m;
   m=s;
   Printf ("%d\n", m->x++);
   Printf ("%d\n", m->x);
   Printf ("%d\n", m++->x);
   Printf ("%d\n", m->x);
   Printf ("%d\n", ++m->x);
   Printf ("%d\n", m->x);
}
```

Soln:



m->x++ arrow operator, postincrement has same precedence so we go for Associativity

They are left to right associative

so ((m->x)++) so m->x = (10)++

so printf ("%d\n", m->x++); ⇒ o/p = 10

printf ("%d\n", m->x); ⇒ o/p = 11

* printf ("%d\n", m++->x);

here m++->x we go from left to right

Since m++ is post fix

first → (m->x)
second → (m=m+1)

so o/p will be 11

printf ("%d\n", m->x) o/p will be 1

printf ("%d\n", ++m->x);

++m->x arrow operator has high precedence than prefix operator

so (++(m->x)) so o/p will be 2

printf ("%d\n", m->x); o/p = 2

→ void main()

```
{ int *ptr;
  int a[10] = {1, 6, 9, 5, 2, 7, 4, 10, 3};
  ptr = a;
  printf("%d", *ptr++);
  printf("%d", *ptr);
  printf("%d", ++*ptr);
  printf("%d", *ptr);
  printf("%d", *++ptr);
  printf("%d", *ptr);
}
```

Soln: printf ("%d", *ptr++)

Postfix has high precedence

so (*(ptr++)) but we know *ptr++ = *ptr then ptr = ptr+1

so o/p = 1

printf ("%d", *ptr); o/p = 6

printf ("%d", ++*ptr);

both ++, * have same priority but we go from right to left so (++(*ptr))

so o/p = 7

printf ("%d", *ptr); o/p = 7

```
Printf ("%d", *++p+r);
```

we do (*(++p+r))

so o/p = 9

```
Printf ("%d", *p);   o/p=9
```

→

```
#include <stdio.h>
typedef struct {
 char *a;
 char *b;
} t;
void f2 (t *p);
main() {
 ts = {"A", "B"};
 Printf ("%s %s \n", s.a, s.b);
 f2(&s);
 Printf ("%s %s \n", s.a, s.b);
}
void f2 (t *p)
{
 p->a = "v";
 p->b = "w";
 printf ("%s %s \n", p->a, p->b);
 return;
}
```

Soln: Here we are passing address of structure s to the function f2(), we might change values pointed by a, b

Here we are changing characters pointed by a, b from "A", "B" to "v", "w"

So o/p will be A B
v w

→

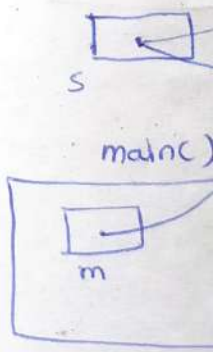
```
struct student
{ char *na
};
struct student
struct student
{ s.name = "r
 printf ("%s
 s.name = "
 return s;
}
void main()
{
 struct stu
 printf ("%
 m.name =
 printf ("%
}
```

a) newton alan

c) alan alan n

Soln: Here m is

fun() is



Gate IT 2006 :

→

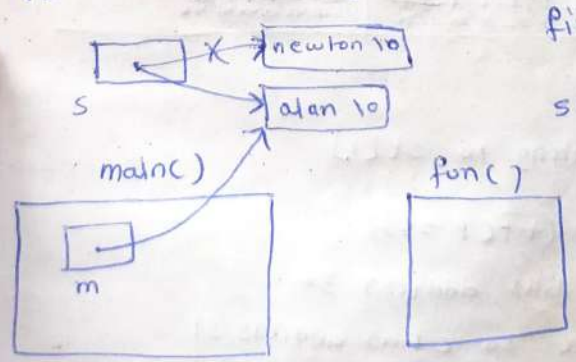
```
#include <st
struct test
```

```
struct student
{   char *name;
};
struct student s;
struct student fun(void)
{
  s.name = "newton";
  printf("%s\n", s.name);
  s.name = "alan";
  return s;
}
void main()
{
  struct student m = fun();
  printf("%s\n", m.name);
  m.name = "turing";
  printf("%s\n", s.name);
}
```

a) newton alan alan b) alan newton alan

c) alan alan newton d) compile time error

Soln: Here m is copying what is present in s as fun() is returning s



first newton will be printed

second alan will be printed

Third alan will be printed.

Gate IT 2006:

```
#include <stdio.h>
struct test {
```

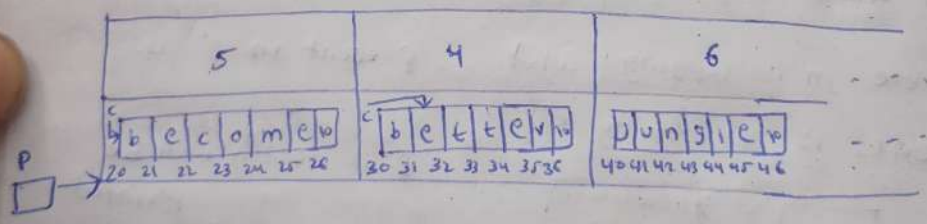
```
{
    int i;
    char *c;
} st[] = {5, "become", 4, "better", 6, "Jungle", 8, "ancestor",
          7, "brother"};

main()
{
    struct test *p = st;
    p+=1;
    ++p->c;
    Printf ("%s,", p++->c);
    Printf ("%c,", *++p->c);
    printf ("%d,", p[0].i);
    Printf ("%s\n", p->c);
}
```

A) Jungle, n, 8, nclastor B) etter, u, 6, ungle

c) cetter, k, 6, Jungle D) etter, u, 8, ncestor

Soln: for questions like these always draw diagrams



Given $p = st$

$p+=1 \Rightarrow p$ is pointing to $st[1]$

now $++p \to c$ so $(++(p \to c))$

c has address 30

++c so c has address 31

Printf ("%s", p++ → c); $\begin{pmatrix} p \to c \\ p++ \end{pmatrix}$ so etter will be printed

now p is pointing to st[2]

printf ("%c", *++p->c);

(p->c) so c has address 40

*(++40) = *(41) = U will be printed

printf ("%d", p[0].i);

p[0] == st[2] so p[0].i = 6

printf ("%s", p->c);

we print from address 41 so ungle will be printed

Gate it 2006

```
# include <stdio.h>
int a1[] = {6, 7, 8, 18, 34, 67};
int a2[] = {23, 56, 28, 29};
int a3[] = {-12, 27, -31};
int *x[] = {a1, a2, a3};
void print (int *a[])
{
    printf ("%d,", a[0][2]);
    printf ("%d,", *a[2]);
    printf ("%d,", *++a[0]);
    printf ("%d,", *(++a)[0]);
    printf ("%d\n", a[-1][+1]);
}
main ()
{
    print(x);
}
```

A) 8, -12, 7, 23, 8

B) 8, 8, 7, 23, 7

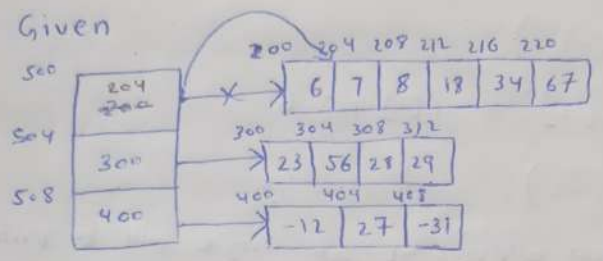
C) -12, -12, 27, -31, 23

D) -12, -12, 27, -31, 56

Soln:

* when we encounter problems like these just draw the diagrams

Given



X[] is having starting address as 500

Printf ("%d,", a[0][2]);

$a[0][2] = *(*(a+0)+2)$

$= *(200+2)$

$= *(200+2(4))$

$= *(208) = 8$

Printf ("%d,", *a[2]);

$*a[2] = *(400)$

$= -12$

Here 504, 508 are not correct addresses but we can take like that

Printf ("%d,", *++a[0]);

$*++a[0] = *++(200)$

$= *(++200) = *(204)$

$= 7$

Printf ("%d,", *(++a)[0]);

$*(++a)[0] = *(504)[0]$

$= *(300) = 23$

* now a = 504 Dont forget to change pointer when updated

Printf ("%d\n", a[-1][+1]);

$a[-1][+1] = *(*(a-1)+1)$

$= *(204+1)$

$= *(208)$

$= 8$

so o/p will be 8, -12, 7, 23, 8

→ main ()

```
{
    struct S1
    {
        char *z;
        int i;
        struct S1 *p;
    };

    struct S1 *ptr = a;
    printf("%s", ++(ptr->z));
    printf("%s", a[(++ptr)->i].z);
    printf("%s", a[--(ptr->p->i)].z); }
```

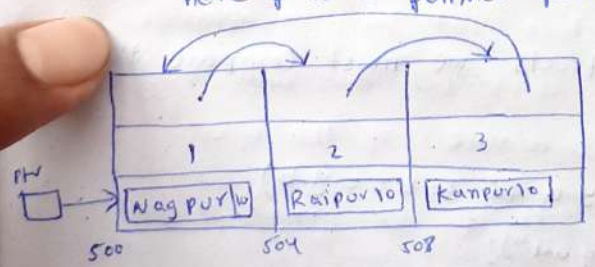
```
static struct S1 a[] = {
    {"Nagpur", 1, a+1},
    {"Raipur", 2, a+2},
    {"kanpur", 3, a}
};
```

A) Raipur, Raipur, Raipur B) Jaipur, Jaipur, Jaipur

C) Raipur, Jaipur, kanpur D) none of these

Soln: struct S1 *P

here p is a pointer pointing to a structure



printf("%s", ++(ptr->z));

say

N	a	g	p	u	r	\0
200	201	202	203	204	205	206

(ptr->z) = 200

++(200)

So z = 201 and o/p = agpur

ptr = 500

printf("%s", a[(++ptr)->i].z);

(++ptr) => ptr = 504

a[2].z = kanpur

printf("%s", a[--(ptr->p->i)].z

a[--(3)].z = a[2].z

= kanpur

Comparing two structs in C: Lec-13 27-09-21

Directly comparing two structures will give an error in 'c' as it is not allowed

i.e struct name {

```
    int a;
    char b;
  } x, y;
int main ()
{
  x.a = 12;   y.a = 12;
  x.b = 'a';  y.b = 'a';
if (x == y)
   printf("equal");
else
   printf("unequal");
}
```

This will give an error as c-programming doesnt allow comparing structs like this

So to compare structs we must compare them memberwise

i.e if (x.a == y.a && x.b == y.b)

```
      printf("equal");
   else
      printf("unequal");
```

Gate-2018:

→ #include <stdio.h>

```
struct ournode {
    char x, y, z;
};
int main()
{
    struct ournode p = {'1', 'o', 'a'+2};
    struct ournode *q = &p;
    printf("%c, %c", *((char *)q+1), *((char *)q+2));
    return 0;
}
```

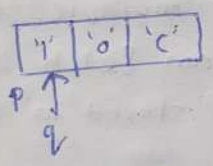
27-08-21

The output of the program is

A) 0, c B) 0, a+2 C) '0', 'a'+2 D) '0', 'c'

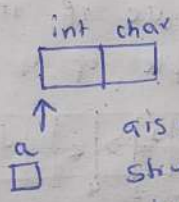
Soln: we know in structure members are present in a contigeous manner in memory

so



p is not a pointer to struct

now we have say



a is a pointer of type struct pointing to a structure

now a+1 will point to another structure but if we want to point it to char in the current struct we do type casting

Since we need to cross integer

we do (int *)a +1

now we are pointing a to char but to get the value of the char we again need to typecast it as *a will print 4B but we need 1B so we do *((char *)(int *a)+1))

(or) Since int is 4B do *((char *) a + 4) will give 'char'

Now comming back to question

q+1 will point to next struct so they did typecast

((char *) q +1) will point to y

so * ((char *) q +1) = '0'

* ((char *) q +2) = 'c'

now printf ("%c, %c", * ((char *) q+1), * ((char *) q+2)))

will give 0, c as o/p

Storing struct in memory - allignment:

(NOT REQUIRED FOR GATE)

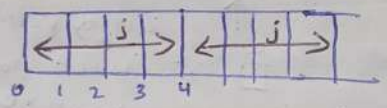
Say we have int j = 5;

now 5 is stored in 4B in memory

but if we are storing integer then int will get stored at the address of multiples of 4

this is because cpu can fetch the integer in one clock cycle if int is stored in the address which is multiple of 4

i.e



→ Similarly if we want to store long then we store it in the address which is multiple of 8 i.e 0, 8, 16, 24, ...

we can store int, long in addresses which are not multiples of 4, 8 but this will lead to the sub optimal performance of the cpu

→ what will be the size of the struct

```
struct stud {
    char c;
    int a;
}
```

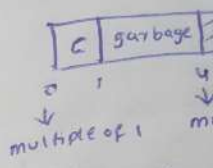
we know that c should be at multiples of 1

and a sho

* Always assume

'0'

so



in order

are filled w

as padding.

so size of

Shortcut:

→ Size of the

struct stu

int i;

char c;

};

Step-1: instea

them

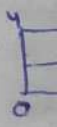
Step-2: sin

take

int and

upwards

i.e



→ struct stu-b

long l;

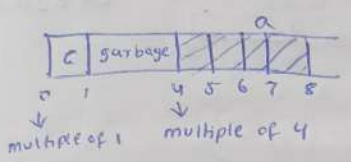
char c;

};

and a should be at multiples of 4

* Always assume that we are starting from address '0'

so



in order to store int at location 4, location 1 to 3 are filled with garbage values which is also called as padding.

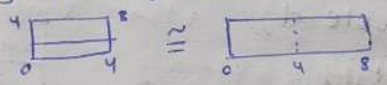
So size of (stud) = 8B (0-8)

Shortcut:

→ Size of the struct will be ?

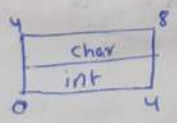
```
struct stu_a {
    int i;
    char c;
};
```

Step-1: instead of writing memory side by side, write them upside i.e



Step-2: since we have two members take the size of largest data type which is int and construct the memory in levels growing upwards in multiples of the size

i.e since int has 4B and we have 2 memb.



char only uses 1B and the remaining 3B are wasted and will be padded.

→ struct stu_b {

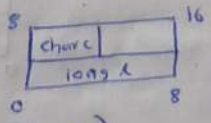
long l;

char c;

};

long is having 8 B

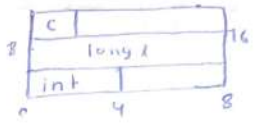
So



7B are Padded, so wasted.

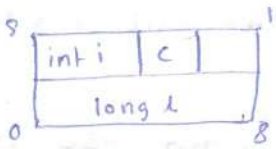
So Sizeof (stu_b) = 16B

→ struct stu_c {
int i;
long l;
char c;
};



24 Bytes

→ struct stu_d {
long l;
int i;
char c;
}



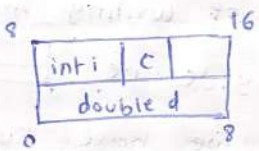
16 Bytes

order matters in struct

in the above first we have int and (0-4) B are occupied by int so there is no space for long as it should start from multiples of '8'. But here we can write as shown above.

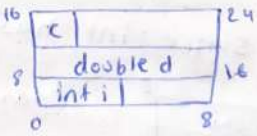
* order of members matters

→ struct stu_e {
double d;
int i;
char c;
};



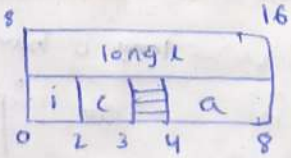
16 Bytes

→ struct stu_f {
int i;
double d;
char c;
};



24 Bytes

→ struct stu_x {
short i;
char c;
int a;
long l;
}



16 Bytes

→ struct stu_y {
short i;
int a;
char c;
long l;
};

** There is
* array as
type
i.e int

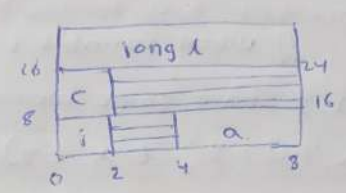
unions :

→ union is
so therefore
as structure
i.e unio

→ Overall
maximum
i.e union

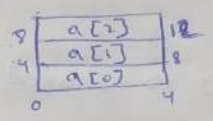
→ what will be
Union test {
}; int x, y;

→ struct stu_v {
short i;
int a;
char c;
long l;
};



24 Bytes

** There is no need for padding in the case of array as all the members / elements are of same type
i.e int a[3]



12 Bytes

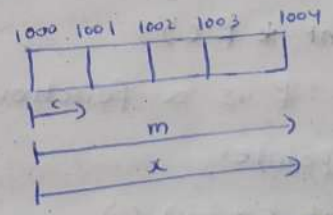
unions :

→ union is a concept borrowed from structures so therefore it follows the same syntax / sentences as structures

i.e union item {
int m;
int n;
float x;
char c;
} c;

→ Overall size of a union will be equal to maximum size of its members

i.e union item {
int m;
float x;
char c;
} code;



in union memory will be shared by all the members

→ what will be the output for the following code
Union test {
int x, y;
};

```
int main()
{
    union test t; // union variable t
    t.x = 2; // x,y share same memory so y = 2
    Printf("%d = x, %d = y", t.x, t.y);
    t.y = 10; // t.x is also updated to 10.
    Printf("x = %d, y = %d", t.x, t.y);
    return 0;
}
```

Since the memory is shared x, y will have same outputs at any time

so o/p will be 2 = x, 2 = y
x = 10, y = 10

Complex declarations:

```
int *p(int);
```

Here p is a function which takes int as input and returns integer pointer.

```
int (*P) [5]
```

P is a pointer to an array of 5 integers

```
int *a[10]
```

a is an array of pointers to int

```
int * f()
```

f is a function which returns integer pointer

```
int (*pf) ()
```

Pf is a pointer to a function which returns int

```
int (*pf) (int)
```

Pf is a pointer to a function which takes int as argument and returns int

Gate CSE

→ what does the following c-statement declare

int (*f)(int *)

A) A function that takes an integer pointer as argument and returns an integer

B) A function that takes an integer as argument and returns an integer pointer

C) A pointer to a function that takes an integer pointer as argument and return an integer

D) A function that takes an integer pointer as argument and returns a function pointer

Soln: option c is the Answer.

int * (* (* fp1) (int)) [10];

fp1 is a pointer to function that takes int as argument and returns a pointer to an array of 10 integer pointers

Pointer to function (Not required for gate)

```
int my_fun (int x)
{
  return x+1;
}
main()
{
  int (*p)(int) = my_fun;
  Printf ("%d", p(2));
}
```

Here we have declared p as a pointer to a function which returns int

So my_fun (2) ≅ p(2)

Getchar():

Getchar() is used to read a single

Character

* Scanf is overkill just to read one character

→ #include <stdio.h>

```
int main()
{
    int c, n1; char c1;
    n1 = 0;
    c1 = getchar();
    while (c1 != 'z')
    {
        n1++;
        c = getchar();
    }
    Printf ("%d", n1);
}
```

we are waiting for the user input

This loop runs until the user had entered 'z' as input

End of File (EoF):

EoF is a special symbol which tells the cpu to stop taking inputs

in windows: control + z

in linux: Control + D

in programs we can use EOF as

```
while (c1 != EoF)
{
}
```

Putchar():

Putchar() function writes a character to screen

Putchar(c) is equivalent to printf("%c", c);

getchar() is equivalent to scanf("%c", &c);

Putchar() will print single character onto the screen

```
int main()
{
    char ch = 'a';
    putchar(ch);
}
```

```
int main()
{
    char ch;
    ch = getchar();
    putchar(ch);
};
```

-> what will be the output for the input abc <Enter>

```
void fun(void)
{
    int c; char c1;
    if (((c1 = getchar()) != '\n'))
        fun();
    putchar(c1);
}
main()
{
    printf("Enter text\n");
    fun();
}
```

Soln: Enter will input as '\n'

so

```
fun()        fun()        fun()        fun()
c1 = a       c1 = b       c1 = c       c1 = '\n'
fun()        fun()        fun()        putchar(\n)
putchar(a)   putchar(b)   putchar(c)
```

o/p will be \n -> newline
c b a

* if we are using EOF then it is a good practise to use int as return type i.e int c; c = getchar(); as EOF is a very largenumber and char datatype cannot hold it

Macros:

-> All lines that start with # are processed by the

the preprocessor

```
#define MAX 120
```

char buf[MAX+1]; will become char buf[120+1]

→ #define plusone(x) x+1

i = 3 * plusone(2);

becomes

i = 3 * 2 + 1

= 7

* Just replace what the expression is as it is dont do any extra thing

wrong answer can be

i = 3 * $\underbrace{\text{plusone}(2)}_{2+1=3}$

= 3 * 3 = 9 (x) wrong

* preprocessor doesnt calculate its job is to replace

→ #include <stdio.h>

#define MULTIPLY (a,b) a*b

int main()

{

printf ("%d", MULTIPLY (2+3, 3+5));

}

Sol^n: a * b = 2+3 * 3+5

= 2 + 9 + 5

= 16

* if we write

#define MULTIPLY (a, b) (a) * (b)

then (2+3) * (3+5) = 5 * 8

= 40

Call by Value:

→ C language uses call by value only and call by reference is not suppo

now we will see a simple program for call by value

```
#include <stdio.h>
void increment (int x) {
    x = x+1;
}
main() {
    int a = 10;
    printf(" before function call %d", a);
    increment(a);
    printf(" after function call %d", a);
}
```

Here we are just passing the value

Call by Reference (NOT IN GATE SYLLABUS):

-> what is printed by the print statements in the program P_1 assumming call by reference parameter passing

```
Program P1()                    func1(x, y, z)
{                               {
    x = 10;                         y = y + 4;
    y = 3;                          z = x+y+z;
    func1(y, x, x);             }
    print x;
    print y;
}
```

A) 10, 3 B) 31, 3 C) 27, 7 D) none

Soln: Call by reference is internally supported by the programming language

Here in call by reference whatever changes made in func1() will reflect in program P1()

in C-programming language for the same operation we have todo the usage of pointers

so func1(3, 10, 10)
x y z

$y = y + 4$
$= 10 + 4 = 14$

$z = x + y + z$
$= 3 + 14 + 10 = 27$

so o/p
27, 3
is wrong

Just like pointers these updations will be Quick

So in P_1 ( ) in fun1 ( )

$Y = Y + 4$ $\quad$ $Z = X + Y + Z$

$Y = 10 + 4$ $\quad$ $= 3 + 14 + 14$

$= 14$ $\quad$ $= 31$

updation is very quick

So o/p will be 31 3

→ for static scoping and dynamic scoping :

C always uses static scoping

in static scoping :

```
int b=5;
int foo()
{
  int a = b+5;
  return a;
}
int bar()
{
  int b=2;
  return foo();
}
int main()
{
  foo();
  bar();
  return 0;
}
```

Here b is not there so in static scoping we goto global variable

so $a = 5 + 5 = 10$

foo() returns 10

in Dynamic scoping :

b is not there in foo() so control goes to the function which called foo()

Since bar() called foo() and b is present in bar() so b = 2 is taken and 7 will be in o/p

Static scoping	Dynamic scoping
foo() ⇒ 10	foo() ⇒ 10
bar() ⇒ 10	bar() ⇒ 7

when main() calls foo() and when control goes to main() in search of b and b is not found so by default global value is taken