proper address not found

Section 6

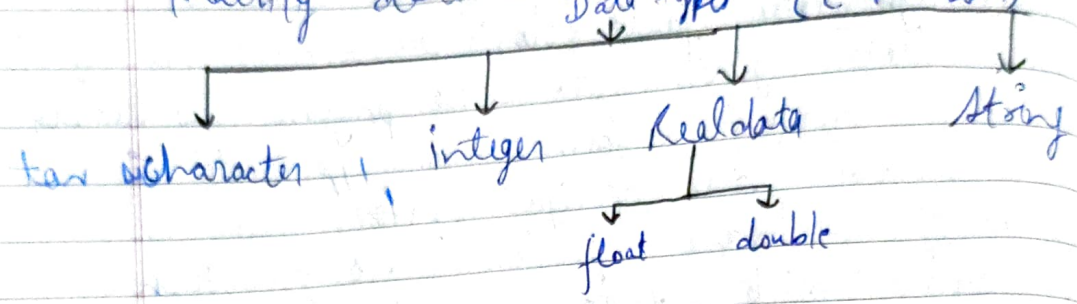
→ Constants and Variables
→ Data types → 4
① → char
② → int
③ → Real (float, double)
④ → string

→ Whenever we are using scanf to scan the data from the keyboard we need to use proper format specifier (%d, %c, %f etc) & proper address (4)

→ $ vi .vimrc
set nu Esc + shift 22 (for nos lines)

→ Data plays massive role in program.
Two values of data:-
(i) Constant (ii) Variable
↓ ↓
fixed can be changed

→ Most of the program we deal with variable type data.
Mainly data characterized in 4 types: (C & V both)

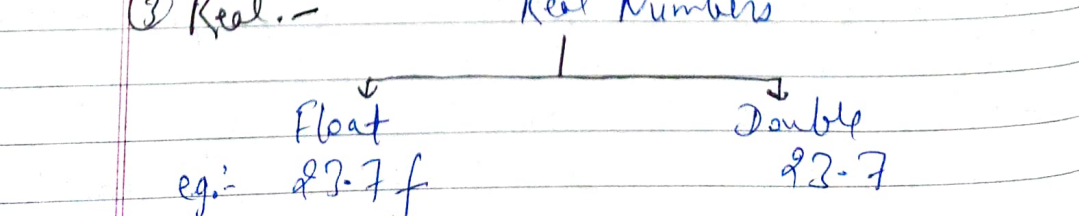


(i) Constants :-
① Constant char :- In a (program, if anything is written in a single quote, compiler will treat it as Character constant.
eg:- 'e', 'a', '8' etc.

② Integer constant :-
- Numbers are treated as integer const
eg:- 70, 150, 201 etc.

→ 10 treated as binary
→ 010 treated as octal
→ 0x10 treated as hexadecimal

③ Real :- Real Numbers



→ only when f is added at the end, it is treated as float constant.

④ String:- In C language, there is no direct data types called as string.
→ If we want to deal with strings, we need to go for char array.
— If we write anything in double quote, it is called as a string constant.
eg: a = "Hello";

* ✓ '1' → char constant
✓ 1 → int constant
✓ 1.0 → double constant
✓ 1.0f → float constant
✓ "1" → String constant

(ii) Variables:- If we want to deal with variables type of data, we need to declare a variable.
— If the variables are declared outside the main(), it is called global variable.
— If declared inside the main(), it is called as local variables.

Syntax of variable declaration

— datatype variable name;

→ Before the data type we can add the following

* Storage class type (auto, static, extern, register)
* type qualifier (constant, variable) Volatile

* size
(long, longlong)
short

* sign
(signed, unsigned)

→ These are added only if requirement of
These are advanced.

Eg:- char ch;
data type ↑ Variable name

Rules for declaring variable:-

- Variable ~~name~~ name should contain
a – z
A – Z
0 – 9 and only one special character
is allowed i.e underscore (_)
- The first letter should not be digit.
eg:- char ch; ✓
char _ch; ✓
char 1ch; ✗

- If char ch alone is written it
is called as variable declaration

→ - when char variable is declared, the
compiler allocates 1 byte (8 bits) of
memory

[char – 1 byte]

→ char ch; // variable declaration

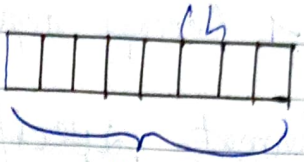
→ char ch = a; // variable initialization

→ When a variable is declared, the compiler will allocates 1 byte of memory; in that we don't know what is the value.

→ If the value is not known it, is known as Unknown value or garbage value or junk value.

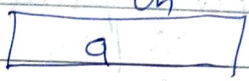
Initilization → declaring variable + storing an ~~init~~ initial value of the variable

char ch;



garbage value

char ch = 'a';



→ Initializing → Declaring + assigning

Note:- Characters are ~~stored~~ in a given memory as its ASCII values.
(American Standard Code for Information Interchange)

→ C language support ASCII codes only
→ C language display English language.

[Unicode - universal code - other language letters can be displayed]

→ For every char, there is an ASCII value
eg:- 'a' - 97, 'A' = 65
'0' - 48 single quote zero

→ These ASCII values are decimal values
eg:- decimal 97 for 'a'.
'a' - 97
'b' - 98
ASCII table

→ $ man ASCII (man is manual page/help)
type q to come out from man page

→ \n → (escape sequence) (next line)

Note:- Format specifiers informs to the printf, in which format, it needs to print the data.
eg:- %c, %d %o %x
char int octal hexa
(dec

→ There is no specific format specifier to print the binary or print in binary.

Here; char = 'a';
charvar ↘ char const is going to be stored in the char var ch.

Note:- Characters are treated as 1-byte integers constant.
eg:- void main()
{
char ch=97;
printf("%c, %d, %o, %x \n", ch, ch, ch, ch);
}
o/p:- a 97 141 61
∴ Here the integer 97 saved as character

Note:- while using scanf(), to scan the data, we need to use a proper format specifier and proper address.

'&' → address operator / bitwise AND

Here '97' is stored in its binary format.

eg:- 97

64	32	16	8	4	2	1	(96+1)
1	1	0	0	0	0	1	

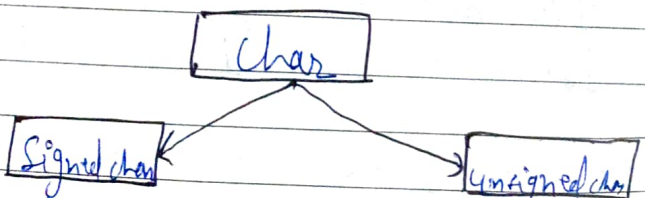
64 + 32 + 1 = 97

→
```
Void main()
{
  char ch;
  printf("Enter ch--\n");
  scanf("%c", &ch);
  printf("%c %d %o %x\n", ch, ch, ch, ch);
}
```

o/p :- Enter ch--
97 → Here only 9 is considered 7 is ignored.

Note:- To terminate the program execution in middle 'ctrl+c' should be pressed.

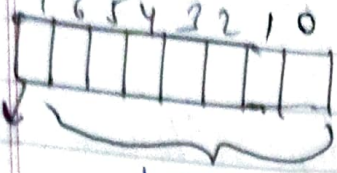
→ char ch; → Here it is treated as signed character



→ Last bit is sign bit & all rest are data bits

→ All bits are data bits

Signed char



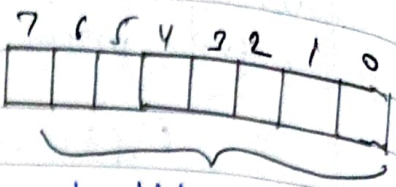
Sign bit — data bits

(1 + 7)

→ By defaul signed char

Range: -128 to 127 ✓

Unsigned char



All bits are data bits

✓ Range : 0 to 255

↓

Storage capacity

Session 7 :-

→ data types.
→ char.
→ short int, int.
→ format specifier
→ more on printf & scanf
→ more on runtime error.

→ 1 bit can store 2 values (0 and 1)
2 bits can store 4 values (2^2)
8 bits can store 256 values (2^8)

Signed char :-

- If 1st bit is '0' then it is positive
- If 1st bit is '1' then it is negative

(sign bit) flow in signed char

+ve ← (0) 0 0 0 0 0 0 0

-ve ← (1) 1 1 1 1 1 1 1

(Sign bit) ↑ indicates sign bit.

How -ve nos are going to be stored in a given memory?

* - In its 2's compliment method.
  (1's compliment + 1)

Here:-

(+ve) ① 0 0 0 0 0 0 0 → -128

⋮

① 1 1 1 1 1 1 1 → -1

→ 4 bits are called as nibble
→ 8 bits are called as 1 byte

Binary value for -10

2	10	
2	5	0↑
2	2	1↑
	1 →	0↑

0 0 0 0 | 0 | 0
1 1 1 1 | 0 | 0 1
- 1 1 1 1 | 1 1 | 0

→ 1 1 1 1 1 1 1 1
↳ If it is '-1', it is signed char
If it is '255', it is unsigned char

→ eg:- | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- If all are data bits, the value is 128 (unsigned)
- If 1st bit is sign bit -128
To know which -ve no. is stored do reverse 2's compliment

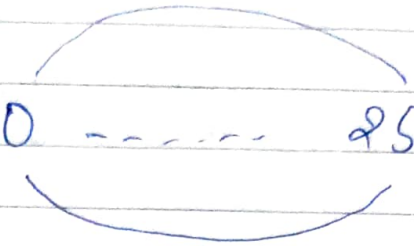
Eg:- 10000000
= -128
Signed char

10000000
= 128
Unsigned char

→ 1 1 1 1 1 1 1 1
= -1
signed char

1 1 1 1 1 1 1 1
= 255
unsigned char

→



circle

Integer:-

Integer (4 bytes)

- Short int → 2 bytes
- Long int → 4 bytes in 32 bits, 8 bytes in 64 bits
- Long Long int → 8 bytes

* → Simply 'int' is 4 bytes.
→ Ex:- ("%ld" is used)

```
Void main ()  // size of operator gives ans in long int.
{
printf ("%ld \n", sizeof (int));
printf ("%ld \n", sizeof (shortint));
printf ("%ld \n", size of (long int));
printf ("%ld \n", size of (long longint);
}
```

o/p:- Here size of operator gives the o/p in bytes

```
4
2
8 (64bit os)
8
```

File a.out → gives config details of the file.

→ long long is possible for only 8 times
→ $ vi /usr/include/limits.h // vi editor used

All the limits are defined here

$ man ASCII

Session – 8

→ short int ent
→ format specifiers
→ Little & big endians.

(i) Short integer :-
→ 2 bytes (16 bits)

Signed short int range
-32768 to 32767

Unsigned int range
0 – 65535

%hd

→ void main()
{
short int i;
printf("Enter the no...\n");
scanf("%d", &i);
%hd is used
printf("i = %d\n", i);

// A warning is thrown for
// scanf() for using wrong
// format specifier
/* As only a warning thrown
.exe file is created
by the compiler */

O/p :-
Enter the no---
10
i = 10
*** stack smashing detected ***
→ Run time error

→ Void main()
{
short int i, j;
printf("&i = %p &j = %p\n", &i, &j);
↑ address operator

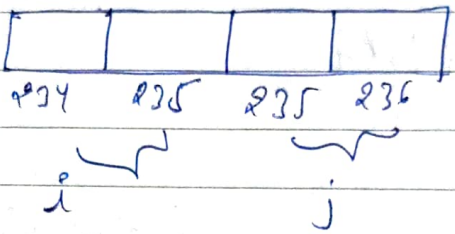
Note:- If we want to print the address of the variable, the format specifiers is %p

- Addresses are in hexadecimal notation

O/p: &i = 0276fd929d[274] &j = 0X76fd929d[230]

→ - Compiler will decide the address
→ - OS will allocate the memory in RAM.

→ Address operator will give us the starting address



→ %u can also be used for short int, but it thrown warning.
when printf is written like below

→ - printf("&i = %d &j = %u\n", &i, &j);
output:-
- &i = 393356676 &j = 393356678

→ - When two individual variables are declared, there is guarantee that they are adjacent. In that case, print the address to ensure the address

→ What happens when a wrong formal specifier is given??

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

```
short int i,j;
printf("Enter j\n");
scanf("%d",&j);
printf("Enter j = %d \n", j);

printf("Enter i\n");
scanf("%d",&i);
printf("i = %d  j = %d\n", i,j);
}
```

// %hd correct specifier

output:-

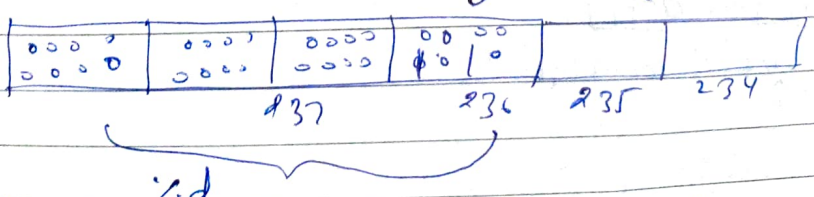
Enter j
10
j = 10
Enter i
15
i = 15 j = 0

When scanf is given with the formed specifier (%hd), j is not disturbed

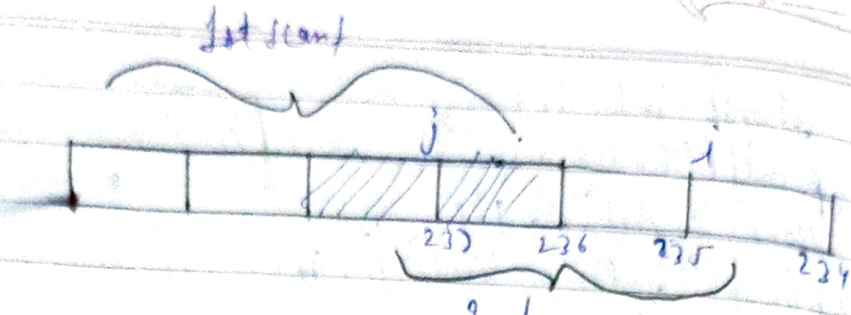
o/p:- Enter j
10
j = 10
Enter i
15
i = 15 j = 10

Stack smash error occure

When 'i' is scanned 'j' is disturbed
'j' is expected to give 10 as the answer when 'j' is scanned for %d (ind);
10 is stored as



- when i is now scanned with %d,
- it was 4 bytes due to %d which disturbes j as they are adjacent, so the remaining value with j is 0.

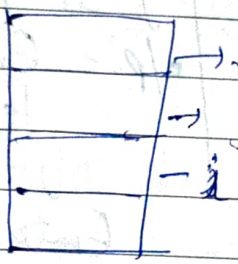


j is disturbed due to the 2nd scan, Thus a proper format specifier & address has to be provided in scanf().

It uses 4 bytes if %hd is used

Stack smash detection error

In a stack function return address are stored



As wrong format specifier is given, the stack fun. return address is disturbed or overwritten. So, when it comes back to call it is looses the fun. return address, so the stack smash error is thrown. [It doesn't know where to continue the program as the return function address is disturbed].

Endian

```
void main()
{
  int  i = 10;
  :
}
```

How 'i' will be stored in the given memory location if it is an integer.

int i = 10;

4 byte binary 10

0000 0000 | 0000 0000 | 0000 0000 | 0000 1010
MSB LSB

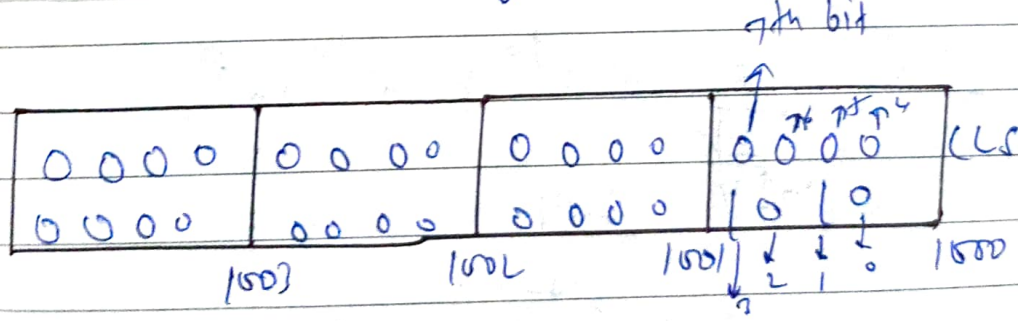
1-byte binary 10
0000 1010

In a given memory location, how a given data is going to be stored?
- It depends on processor (hardware) endianess.

→ Endianess is not a compiler dependent, OS dependent.
→ It is hardware dependent.

Endian :- Endian is a storage mechanism, in a given memory how a given data stored.

Little endian :-
In this environment, LSB is stored in given lower memory location

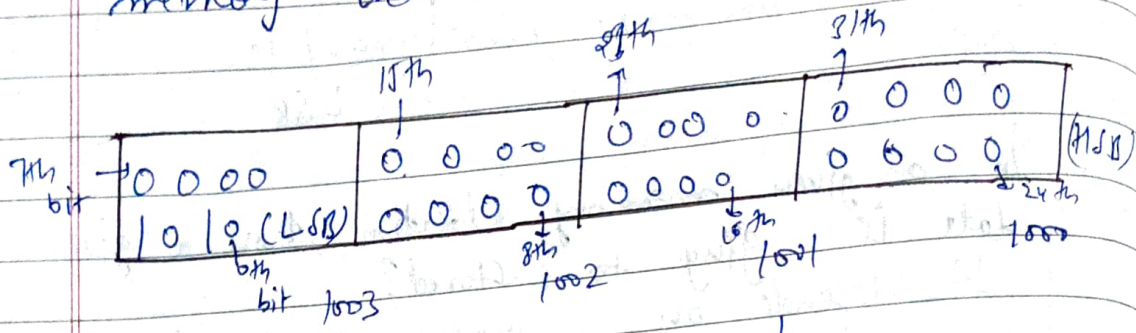


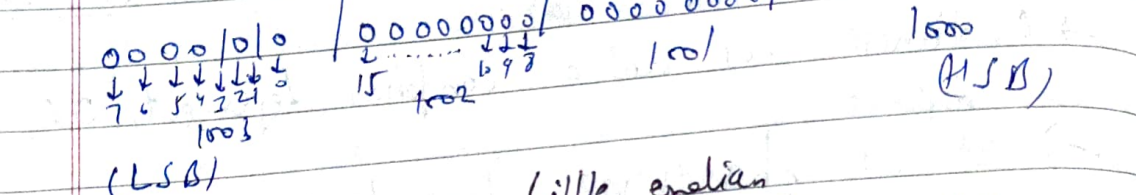
Example:
Here 1000 is the lower address

Use
→ Intel → Little endian -
→ Motorola → Big endian -

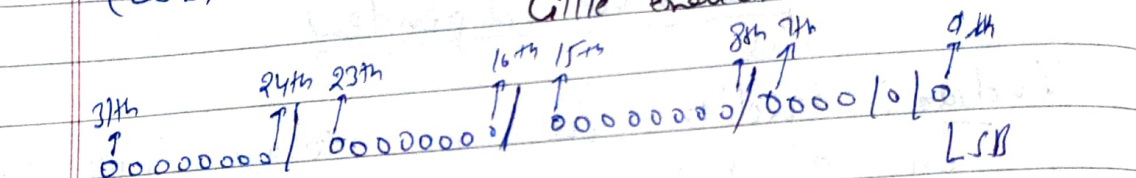
Big endian :-

Here, LSB is stored in given higher memory location.





Little endian



→ Endians is also called as byte ordering (arrangement)

→ 32 bits

$$2^{32} = 2^{10} \times 2^{10} \times 2^{10} \times 2^{2}$$

2^{10} = 1K

2^{10} = 1M

2^{2} = 4

1G × 4 = 4G

$= 2^{32} = 4G$ bits

∴ So the range of unsigned int will upto 4G bits

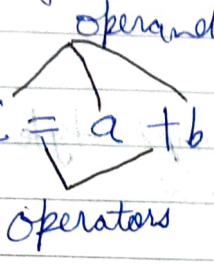
[Unsigned — 0 to 4G]

Session :- 9

Unsigned int range 0 to 4b

→ Operators
  → Arithmetic operators
  Assignment operators

→ Every program is responsible for procesing the data
Processing is just doing operations on the data.
Operators perform some operations on the data



Eg:-

- These operands can be variable or constants

→ Depending upon the operands supplied operators are divided into 3 categories.
(i) Unary (eg:- ++a, --a, (one operands))
(ii) Binary (eg:- a+b (2 operands))
(iii) Ternary (eg:- (3 operands)) 1 ? 2 : 3

- Assignment operators:- =
- Arithmetic operators:- + - * / %
- Relational operators:- > >= < <= == !=
- Logical operators:- && || !
- Bitwise operators:- & - Bitwise AND, | Bitwise OR, ^ Bitwise EXOR, ~ Bitwise compliment, << Bitwise left shift, >> Bitwise right shift.
- Conditional (Ternary operator):- _ ? _ : _
- Sizeof operator:- sizeof()

- Reference operator and dereference operator:-
  Reference operator is also called as an address operator (&) . Gives the first or starting address of the variable

  Dereference operator called as pointer operator
  '*' (* pointer)
- Dot & arrow operator:- . →
- Inc and Dec operator:- ++ --

ii- Unary operators :-

++ -- sizeof() &
* ! . ~

open command prompt type
↳ $ man operator

It gives the list of operators & its associativity or its precedence.

- Compound assignment operator:-

+= -= *= /= %= <<= >>=
&= ^= |=

① Assignment operator:- (=)

→ Symbol (=)
- It is a binary operator (needs two operands)

=]
RHS is copied to the LHS

```
a = b = 10;
    ← Associativity
```

First 10 is assigned to b, the b which is equal to 10 is assigned to a

→ If

```
a = 10 = b;   is given
```

The compile will try to copy the garbage value to the number 10. But a constant can't be changed. So an error is thrown

→ **lvalue required**

↳ It means left hand side, a variable (operand) should be present. in = operator

a = 10 ✓
a = b ✓
10 = b ✗ invalid

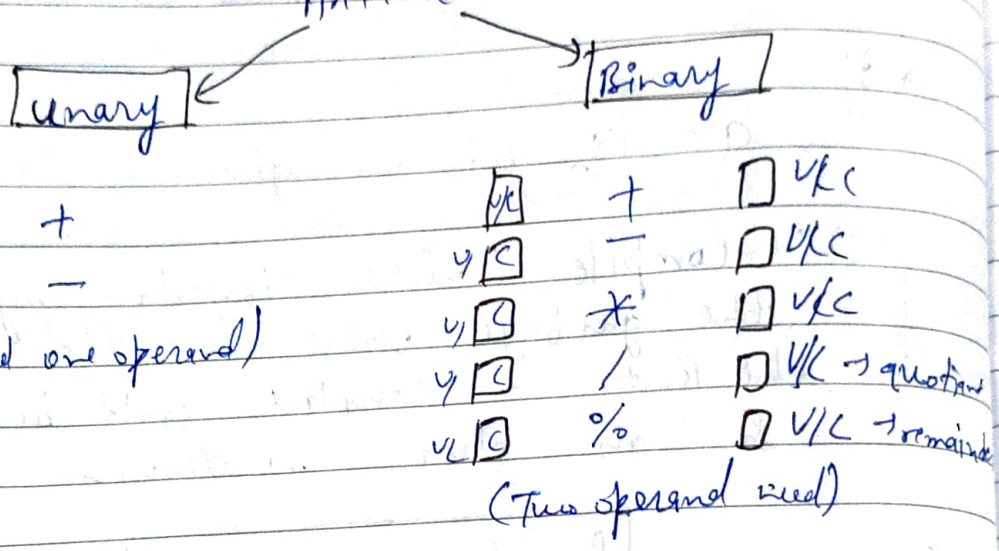
→ a = 10 → Constant is stored into a variable
a = b → Variable is stored into a variable
10 = a → Constant value can't be changed

When an assignment operator is given with a constant on the LHS, compiler throws an error (lvalue required)

* LHS must be variable
* RHS can be variable or constant

- Assignment operator will modify the left operand so LHS should be variable.

② Arithmetic operators:-



Arithmetic binary

→
```
void main()
{
  int i=11, j=2, k;
    k = i+j;
  printf("i=%d j=%d k=%d\n", i,j,k);
    }
```

k = i+j

Arithmetic operator has higher priority than the assignment operator.

∴ k = 11+2; k=13

o/p:- i=11 j=2 k=13

All arithmetic operators will not modify the operands.

c = a+b ✓ valid
c = 10+20 ✓ valid
c = a+20 ✓ valid
c = 20+a ✓ valid
a+b = c ✗ invalid

→
```
void main()
{ int i, j, k
  i = 11, j = 2;
  k = i/j
printf("%d", k);
}
```

2) 11 | 5
10
1

o/p = k=5
↳ k is 5 because k is defined as an integer (// gives quioted.

→
```
int i=11, j=2, k;
k= i%j   // i%j
printf("%d", k);
}
```

o/p:- k=1 % modulus gives the remainder value.

⇒ Note:- we can't apply '%' modulus operator to real numbers (float & double). If we applied a syntactical (translator) error is thrown.

→ Eg:-
```
void main()
{
  int a=10, b=20, c;
  c=a+b   // 1 → '+', 2 '='
  c=a+b*10 // * higher priority than +
  c=(a+b)*10; // grouping have higher priority
              // but it will do when needed.
  c=10+a; c=20+10;
```

Operators: * / → Here associativity is from left to write
+ −
→
=

```
c = a*b + 2 - 10/3;
c = R1 + 2 - 10/3;
c = R1 + 2 - R2
c = R3 - R2
c = R4
c = *a + (2-10/3;
```

Unary arthmetic operator :-

```
int i=10, j;
    j=-i;
```

unary '-'

O/p :- i=10 j=-10

Note :- Unary '+' is a default sign for all operands.

→ Write a program for swaping of two no. using 3 variables.

→ Logic

```
#include<stdio.h>
Void main()
{
  int a, b, temp;
printf("Enter a and b \n");
scanf("%d %d", &a, &b);   // a=10, b=20
printf("before swap a = %d b=%d", a, b);
  temp = a;   // a=10 so 10 goes into temp
  a = b;   // b=20, a=b means a becomes
```

b = temp; // temp is 10 so b is now 10
printf("After swap a = %d and b = %d", a, b);
}

op:- a = 10 b = 20
a = 20 b = 10

// t = a;
a = b;
b = t;

$ cp sl.c s2.c // copies content of sl.c into s2.c

→ Swap using 2 variable (without temp variable):-

a = a+b;
b = a-b;
a = a-b;

a = a+b-(b=a);

a = a*b/(b=a);

a = a*b; /* // grouping just groups the operands
b = a/b; but does not solve the expression,
a = a/b; through grouping has highest priority
grouping () is solved when it is required. */

→ Example program:-

#include <stdio.h>
void main() // unsigned range 0 to 4G
{
unsigned int i=2, j=3, k;
// k = 6 "i = %u j = %u k = %u \n" x
k = i * j; // %u because of data type unsigned int
printf("i = %u j = %u k = %u \n", i, j, k);
}

{ Now consider:-
unsigned int i = 2000000000, j = 2, k;
2G
k = i * j;
printf("i = %u j = %u k = %u", i, j, k);
}

o/p:- i= 2000000000 j=3 k=1773451773
(Truncated k value)

- K shows wrong value because k has a max value of 4b, k gives truncated value
- There's no warning because the expression is solved at run time.
- So as to increase k, we declare k as unsigned long int k, here k's capacity of k is increased from 32 bit to to 64 bit
- Even though it is increased, the o/p obtained is still truncated

```
unsigned int= 2000000000 , j=3 ;
unsigned long int k ;
 k= i*j;
printf("i=%u  j=%u  k=%u\n", i, j, k);
```

Here K= i*j;

- the multiplication result will be in some buffer of the size of the temp buffer depends on the operands.
- Here the operands are of 4b, so, the temp buffer is of the size 4b. Thus, the product of the multiplication is truncated data, even though the capacity of 'k' is increased.

long int (k) = int i * j;

So their product of is also int which is truncated at the buffer.

— To solve this, one of the operands declared as long int.

```
unsigned int i = 2000000000;
unsigned long int k, j = 3;
k = i * j;
```

// Here the required o/p; which is of 6G is obtained

Implicit type cast is happening here.

⇒ When compiler does an expression, with operands of different size, compiler converts them into one size.

k = [i] * [j]

Implicit type conversion:-

- Compiler converts all operands up to the type of the largest operand, temporarily. This is called as type promotion or implicit type conversion.

⇒ k = i * j;
int long int // compiler converts this unsigned long int.

→ In short, lower data type is converted into higher data type.