

➔ Allocation

- To create a value, the value needs to be stored. It is stored inside memory.
- If we want to put some value (let it be x), the computer's task is to get the user's i/p value and store it inside the memory.
- For a value, we need to specify its size i.e., how much storage requirement the value has.
The storage specification is a global standard i.e., Bytes
- The language doesn't accept a value without mention of its allocation i.e., Declaration
Ex: -

For storing those values viz., a, b, c , we need to allocate storage corresponding to the values.

1 byte a, b, c

The statement will allocate storage in memory corresponding to the three values viz., a, b, c .

- 1 byte = 8 bits

For overcoming limitation on storage range, there are different storage specifications available

8 bits

-

1 byte (char/int)

16 bits

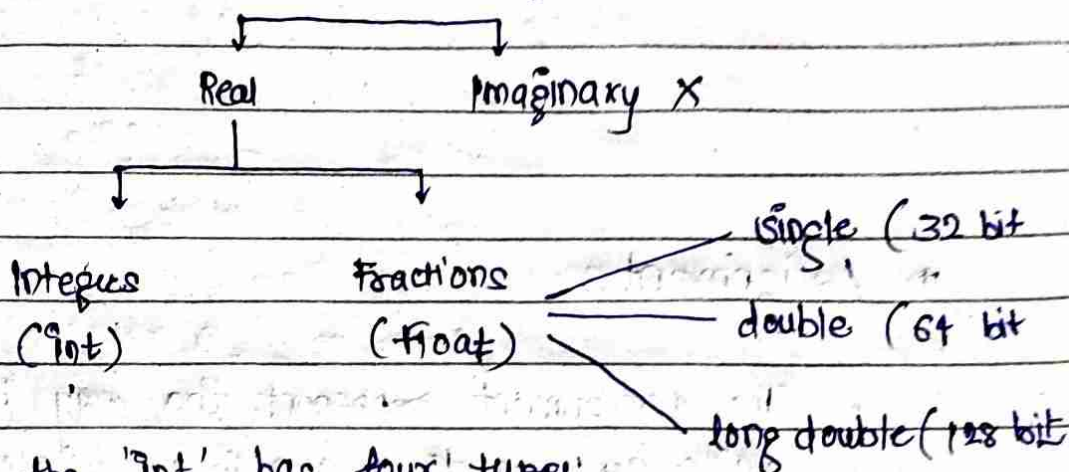
-

2 byte (short int)

32 bits - 4 byte (long int)

64 bits - 8 byte (long long int)

Types of numbers available for storage.



In C, the 'int' has four types

- 1> char (int-8)
- 2> short (int-16)
- 3> long (int-32)
- 4> long long (int-64)

Note: • 'char' is an 'int' | It's not a datatype, it's a modifier

- Normally, the char, short, long, long long int are signed (signed range of numbers)

For range to be unsigned, we need to mention the 'int' to be unsigned.

Ex: unsigned short int (16 bits/2 byte)

Signed Representation

int8_t	char
int16_t	short
int32_t	long
int64_t	long long

Unsigned Representation	uint8_t	char
	uint16_t	short
	uint32_t	long
	uint64_t	long long

➔ Assignments

- The assignment represent the way in which value assigned to a variable is stored in memory.

char 1 byte

short 2 byte

long 4 byte

long long 8 bytes

- Suppose for a short value.

Ex: short a = 25;

Binary: $(25)_{10} = (00011001)_2$

For short, \rightarrow 2 bytes ($2 \times 8 = 16$ bits)

\therefore short a = $(00000000\ 00011001)_2$

The way how the above value stored in memory is given by-

Value storage in memory

Big Endian Representation

Little Endian representation

short a = 25;

short a = 25;

byte 0	0000 0000
byte 1	0001 1001

byte 0	0001 1001
byte 1	0000 0000

fig: memory representation

fig: Memory representation

- Big value part comes first in memory

- Little value part comes first in memory

Octal Number Representation

Step 1: Take the given number in binary format.

Step 2: Divide it into group of 3 bits from RHS

Step 3: Get the equivalent decimal value corresponding to each group of 3 bits.

Step 4: Represent the number by putting '0' at the start in assignment.

Ex: For 0001000 \rightarrow 0001000 \rightarrow 010

\therefore short a = 010;

printf("%d", a);

O/p :- 8

In representation, the first digit is '0'.

DATE: ____/____/____

- The negative integers are stored using 2's complement form.

Ex:

 $a = 8; \longrightarrow 00001000$
 $b = -8; \longrightarrow 10001000$
 $c = a + b;$
 $+ 1110111$

1's complement

o/p: 0

 1111000

2's complement

This value will be stored in memory

Result:

 $a + b = 00001000$
 $+ 1111000$
 $c = 00000000$
