Type casting and Storing variables

conversion of data one type to another (if it is valid) is called Type casting.

Example

int a = 'a';

variable a will store the ASCII value of 'a' = 97.

Example

char ch =98;

98 will automatically get typecasted to its corresponding character.

This automatic typecasting is called implicit typecasting

```
int a = 'a';;
cout << a << endl;
char ch = 98;
cout << ch << endl;
```

→

output!

```
97
b
```

2) What if we type cast from int to char but the value is too large to be stored in char?

Solution! A warning is thrown and last byte from the orginal value is given to the character

```
Char ch =123456;
cout << ch <<endl
```

→

output/

```
1 warning generated
@
```

=> How negative numbers are stored?

Sol/ The first bit tells us whather the number is positive or negative

First bit → 0 means positive
↳ 1 means negative.

Steps to store -5 in binary format:

① Ignore the -ve sign. (5)

② write the binary representation of 5.

③ Take its 2's compliment and store it.

Example: a = -5

① -5 → 5 (Ignore the -ve sign)

② 5 → 0101 → $\underbrace{000\ldots}_{29\ \text{zeros}}$ 0101 (Binary

③ 2's compliment = 1's compliment + 1

5 → 000....0101

1's compliment → 11...1010 (Flip the bits)

$$\begin{array}{r} 11\ldots1010 \\ +\qquad\quad 1 \\ \hline \end{array}$$

2's compliment → $\underbrace{11\ldots}_{29\ \text{one}}$ 1011

⇒ Displaying Negative numbers!

① Take 2's compliment of the stored number

stored: ①11 . . . 1011
↳ this shows -ve
↳ -5 Print hoye gese!

11 . . 1011

1's → 00 . . . 0100

2's → 00 . . . 1011 = 5

@why 2's compliment?

If we stored numbers as it ~~a~~ is without using 2's compliment, then

1	0	0	. . .	0	0

and

0	0	0	. . .	0	0

will be equal and thus waste space

@store only positive Integers

The default signed representation allows us to store both positive and negative value.

To store only positive integers, we use unsigned.

Eg: unsigned int a = 10;

④ What if we store a negative number in an unsigned number?

Example:

```
unsigned int a = -112;
cout << a << endl;
```

output:

```
4294967184 ??
```

Explaination:

We tried to store -112.

-112 = 2's compliment of 112

112 → $\underbrace{000 \ldots}_{24\ \text{Zeros}}$ 0110000

1's compliment → 111 ... 1000 1111

$+$ 1

2's compliment → 11 ... 10010000

unsigned int used all 32 bits to store the value and the MSB.(=1) will make the value.

An unsigned int does not use the 2's compliment again to display the number.

Thus, 11...10010000. gets printed as it is in decimal

Therefore,

```
unsigned int a=-112;
cout << a << endl;
```

output!

```
4294967184
```

Operators

Basic Arithmetic operators!

+, -, x, /, %

Caution ⚠