

Binary Numbers: There are two types of binary numbers:

- 1) **Unsigned Numbers:** It includes only positive numbers.

+Integer

Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110

$$\begin{aligned}
 &0 \text{ to } 2^n - 1 \\
 &\qquad\qquad n = 3 \\
 &= 2^3 - 1 \\
 &= 8 - 1 \\
 &= 7
 \end{aligned}$$

We can represent 0 to 7 numbers using 3 bits

- 2) **Signed Numbers:** It includes both positive and negative both numbers. Methods to represent signed numbers are as following:

- 1) **Signed magnitude**
- 2) **1's complement**
- 3) **2's complement**

	<u>n-1</u>
<u>1</u>	0 0 0 0 1 0 0
MSB	
0 = Positive Number	
1 = Negative Number	

1's complement of binary number: Flip the bits of given binary number:

Given binary number: 10010

1's complement: **01101**

2's complement of binary number: Flip the bits of given number and add 1 to it to get 2's complement. 2's complement is

	1 0 0 1 0
1's Complement	0 1 1 0 1
Add 1	+ 1
2's Complement	0 1 1 1 0

Usage of 2's complement: **2's complement** is the most used to represent the **signed integers** because it obeys the rules of addition and subtraction. If you add 1 to 1111, you get 0000. Hence 1111 should be -1

How the negative numbers are stored in memory:

Example : `int a = -2056;`

Binary of 2056 will be calculated which is:

00000000000000000000100000001000 (32 bit representation, according of storage of int in C)

2's complement of the above binary is:

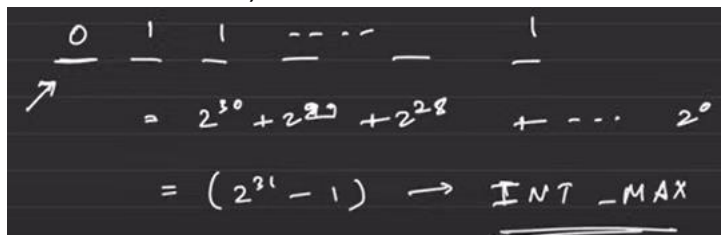
1111111111111111111101111111000.

So finally the above binary will be stored at memory allocated for variable a.

When it comes on accessing the value of variable a, the above binary will be retrieved from the memory location, then its sign bit that is the left most bit will be checked as it is 1 so the binary number is of a negative number so it will be 2's complemented and when it will be 2's complemented will be get the binary of 2056 which is:

00000000000000000000100000001000

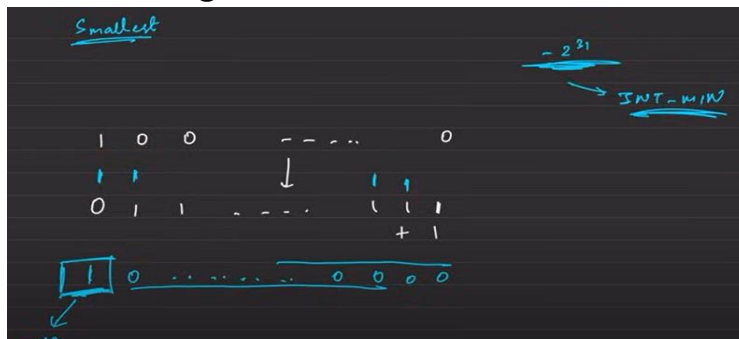
Largest integer number which will be stored: left most bit is sign bit(which will tell it is + or - number)



Handwritten derivation of the largest integer number (INT_MAX) based on a 32-bit sign bit:

$$\begin{aligned} & \begin{array}{ccccccc} 0 & 1 & 1 & \dots & 1 & & 1 \\ \hline & & & & & & \end{array} \\ & \nearrow \\ & = 2^{30} + 2^{29} + 2^{28} + \dots + 2^0 \\ & = (2^{31} - 1) \rightarrow \underline{\underline{INT_MAX}} \end{aligned}$$

Smallest integer number which will be stored:



Handwritten derivation of the smallest integer number (INT_MIN) based on a 32-bit sign bit:

Smallest

$$\begin{array}{ccccccc} 1 & 0 & 0 & \dots & 0 & & \\ \downarrow & & & & & & \\ 0 & 1 & 1 & \dots & 1 & 1 & 1 \\ & & & & & + & 1 \\ \hline \boxed{1} & 0 & \dots & 0 & 0 & 0 & 0 \end{array}$$

-2^{31}
 $\rightarrow \underline{\underline{INT_MIN}}$

Bitwise Operators:

Bitwise AND Operator (&) in C:

bit a	bit b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise OR (|) Operator

bit a	bit b	a b
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise XOR (^) Operator

bit a	bit b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

The Left Shift (<<) Operator:

The left shift operator is represented by the << symbol. It shifts each bit in its left-hand operand to the left by the number of positions indicated by the right-hand operand. Any blank spaces generated while shifting are filled up by zeroes.

Assuming that the int variable "a" has the value 60 (equivalent to 0011 1100 in binary), the "a << 2" operation results in 240, as per the bitwise left-shift of its corresponding bits illustrated below –

0011 1100 << 2 = 1111 0000

The Right Shift (>>) Operator:

The right shift operator is represented by the >> symbol. It shifts each bit in its left-hand operand to the right by the number of positions indicated by the right-hand operand. Any blank spaces generated while shifting are filled up by zeroes.

Assuming that the int variable a has the value 60 (equivalent to 0011 1100 in binary), the "a >> 2" operation results in 15, as per the bitwise right-shift of its corresponding bits illustrated below –

0011 1100 >> 2 = 0000 1111

The 1's Complement (~) Operator

The 1's complement operator (~) in C is a unary operator, needing just one operand. It has the effect of "flipping" the bits, which means 1s are replaced by 0s and vice versa.

a	~a
0	1
1	0

1. Set a bit of number:

*This can be done by left-shifting the value **1** by '**pos**' positions (**1<< pos**) and performing a bitwise OR operation with number **n**. This operation effectively turns on the bit at the specified position.*

```
// n=number
// pos=It is the position at which we want to set the bit
void set(int &n, int pos)
{
    n |= (1 << pos);
}
```

2. Unset a Bit of Number:

*This can be done by left-shifting the value **1** by **pos** positions (**1<< pos**) and then use bitwise NOT operator '~' to unset this shifted **1**, making the bit at position **pos** to **0** and then use Bitwise AND with the number **n** that will unset bit at desired position of number **n**.*

```
#include <iostream>
```

```
// Unset (clear) a bit at position pos in number n
```

```
void unset(int &n, int pos) {
    n &= ~(1 << pos);
}
```

```
int main() {
    int n = 15; // 1111 in binary
    int pos = 1;
    unset(n, pos); // Should change n to 13, which is 1101 in binary
    std::cout << n << std::endl; // Output should be 13
    return 0;
}
```

3. Flip a Bit of Number:

Use the bitwise XOR (^) operator to toggle (flip) the bit at the given position. If the bit is **0**, it becomes **1**, and if it's **1**, it becomes **0**.

```
// Flip (toggle) a bit at position pos in number n
void flip(int &n, int pos) {
    n ^= (1 << pos);
}
```

4. Checking if Bit at nth Position is Set or Unset:

This can be done by performing a bitwise AND operation with a mask having only that bit set. If the result is non-zero, the bit is set; otherwise, it's unset.

```
// Check if the bit at position pos in number n is set (1) or unset (0)
bool isBitSet(int n, int pos) {
    return ((n & (1 << pos)) != 0);
}
```

5. Check if a number is even or odd

You can determine if a number is even or odd using bitwise AND.

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

```
int main() {
    int n = 5;
    if (n & 1)
        printf("%d is odd\n", n);
    else
        printf("%d is even\n", n);
    return 0;
}
```

6. Count the number of set bits (Hamming Weight)

Count the number of 1s in the binary representation of a number.

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

```
int countSetBits(int n) {
    int count = 0;
    while (n) {
        count += n & 1;
        n >>= 1;
    }
    return count;
}
```

```

}

int main() {
    int n = 13; // Binary: 1101
    printf("Number of set bits: %d\n", countSetBits(n));
    return 0;
}

```

7. Check if a number is a power of two

A number is a power of two if it has only one set bit.

```

#include <stdio.h>

int isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}

int main() {
    int n = 16;
    if (isPowerOfTwo(n))
        printf("%d is a power of two\n", n);
    else
        printf("%d is not a power of two\n", n);
    return 0;
}

```

8. Toggle a specific bit

Toggle the *iii*-th bit of a number.

```

#include <stdio.h>

int toggleBit(int n, int i) {
    return n ^ (1 << i);
}

int main() {
    int n = 5; // Binary: 101
    int i = 1;
    printf("Number after toggling bit %d: %d\n", i, toggleBit(n, i));
    return 0;
}

```

9. Clear the iii-th bit

Set the iii-th bit to 0.

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

```
int clearBit(int n, int i) {  
    return n & ~(1 << i);  
}
```

```
int main() {  
    int n = 5; // Binary: 101  
    int i = 0;  
    printf("Number after clearing bit %d: %d\n", i, clearBit(n, i));  
    return 0;  
}
```

10. Swap two numbers without a temporary variable

Using XOR to swap values.

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

```
int main() {  
    int a = 5, b = 7;  
    printf("Before swap: a = %d, b = %d\n", a, b);  
    a = a ^ b;  
    b = a ^ b;  
    a = a ^ b;  
    printf("After swap: a = %d, b = %d\n", a, b);  
    return 0;  
}
```

11. Reverse bits of a number

Reverse the bits in a number.

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

```
unsigned int reverseBits(unsigned int n) {  
    unsigned int reversed = 0, i;  
    for (i = 0; i < 32; i++) {  
        reversed |= ((n >> i) & 1) << (31 - i);  
    }  
    return reversed;  
}
```

```

}

int main() {
    unsigned int n = 5; // Binary: 101
    printf("Reversed bits: %u\n", reverseBits(n));
    return 0;
}

```

12. Find the only non-repeating element in an array

Given an array where every element appears twice except one, find the unique element.

```

#include <stdio.h>

int findUnique(int arr[], int n) {
    int result = 0;
    for (int i = 0; i < n; i++) {
        result ^= arr[i];
    }
    return result;
}

int main() {
    int arr[] = {2, 3, 5, 4, 5, 3, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Unique element: %d\n", findUnique(arr, n));
    return 0;
}

```

13. Find the position of the rightmost set bit

Get the position (1-based index) of the rightmost 1-bit.

```

#include <stdio.h>

int rightmostSetBit(int n) {
    return (n & -n);
}

int main() {
    int n = 10; // Binary: 1010
    printf("Rightmost set bit: %d\n", rightmostSetBit(n));
}

```



```
    return 0;
}
```

14. Check if two numbers have opposite signs

Determine if two integers have opposite signs.

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

```
int haveOppositeSigns(int x, int y) {
    return (x ^ y) < 0;
}
```

```
int main() {
    int x = 5, y = -10;
    if (haveOppositeSigns(x, y))
        printf("%d and %d have opposite signs\n", x, y);
    else
        printf("%d and %d have the same signs\n", x, y);
    return 0;
}
```