All elements of Set in C++

```
#include <iostream>
using namespace std;

void set(int x) {
    for (int i = 0; i < 32; ++i) {
        if (x & (1 << i)) {
            cout << i << endl;
        }
    }
}

int main() {
    int x = 282; // Binary representation:
100011010
    // int x = 7; // Binary representation:
111

    set(x);

    return 0;
}
```

Binary of 282

```
Decimal: 282
Binary : 00000000 00000000 00000001 00011010
Bits    :        ↑         ↑  ↑↑
Positions:       8         5   3 1  ← these are the set bits
```

🔁 Loop Table

i (bit position)	1 << i (binary)	x & (1 << i)	Is Bit Set?	Output
0	000...0001	0	✗	-
1	000...0010	2	✓	1
2	000...0100	0	✗	-
3	000...1000	8	✓	3
4	000..1_0000	0	✗	-
5	000.10_0000	32	✓	5
6	000.100_0000	0	✗	-
7	001.000_0000	0	✗	-
8	010.000_0000	256	✓	8
...	...	0	✗	-
31	100...0000 (bit 31)	0	✗	-

✓ Final Output

```
1
3
5
8
```

Output:-

```
1
3
4
8
```

Bit check in C++

```
#include <iostream>
using namespace std;

void bitChecker(int x, int k) {
    if ((x & (1 << k)) != 0) {
        cout << k << "th bit is 1" << endl;
    } else {
        cout << k << "th bit is 0" << endl;
    }
}

int main() {
    int x = 22; // Binary: 10110
    for (int k = 0; k <= 4; ++k) {
        bitChecker(x, k);
    }

    return 0;
}
```

Given:

- x = 22 → binary = 10110
- We are checking each bit from position 0 to 4

🧮 Dry Run Table:

k (Bit Position)	1 << k (Mask)	x & (1 << k)	Is Bit Set?	Output
0	00001 (1)	10110 & 00001 = 00000	No	0th bit is 0
1	00010 (2)	10110 & 00010 = 00010	Yes	1th bit is 1
2	00100 (4)	10110 & 00100 = 00100	Yes	2th bit is 1
3	01000 (8)	10110 & 01000 = 00000	No	3th bit is 0
4	10000 (16)	10110 & 10000 = 10000	Yes	4th bit is 1

✅ Output:

```
0th bit is 0
1th bit is 1
2th bit is 1
3th bit is 0
4th bit is 1
```

```
0th bit is 0
1th bit is 1
2th bit is 1
3th bit is 0
4th bit is 1
```

Div by 2k in C++

```
#include <iostream>
using namespace std;

int main() {
    int x = 24;
    int k = 3;
    int res = x >> k; // Right shift
operation to divide x by 2^k
    cout << res << endl;

    return 0;
}
```

Given:

- x = 24
- k = 3
- Operation: x >> k means shift the bits of x to the right by k positions (i.e., divide x by 2k=82^k = 82k=8).

🔢 Binary Representation

Variable	Binary	Decimal
x	0001 1000	24

Now right shift by 3 positions:

- Original: 0001 1000
- After >> 1: 0000 1100 (12)
- After >> 2: 0000 0110 (6)
- After >> 3: 0000 0011 (3)

✅ Final Result:

cout << res << endl; // prints: 3

So the output is:

3

Output:-

```
3
```

Even Odd in C++

```
#include <iostream>
using namespace std;

void fun(int x) {
    if ((x & 1) == 0) {
        cout << "even" << endl;
    } else {
        cout << "odd" << endl;
    }
}

int main() {
    int x = 27;
    fun(x);

    return 0;
}
```

Input:

- `x = 27`
- Binary of `27` = `11011`

💡 Logic:

```
if ((x & 1) == 0)
```

- `x & 1` checks the least significant bit (LSB)
- If the LSB is 1 → **odd**
- If the LSB is 0 → **even**

🧮 Dry Run:

Expression	Value	Explanation
`x`	`27`	Decimal input
`x` (binary)	`11011`	Binary representation of 27
`x & 1`	`11011 & 00001` = `00001`	LSB is 1 → odd
`== 0`	`false`	So it goes to the `else` block
Output	`odd`	✓

✓ Final Output:

```
odd
```

```
odd
```

Power of 2 in C++

```
#include <iostream>
using namespace std;

void powerOf2(int x) {
   if ((x & (x - 1)) == 0) {
      cout << x << " is Power of two" <<
endl;
   } else {
      cout << x << " is not Power of two"
<< endl;
   }
}

int main() {
   int x = 9;
   for (int i = 1; i <= 32; i++) {
      powerOf2(i);
   }

   return 0;
}
```

Key Logic:

```
if ((x & (x - 1)) == 0)
```

This works because:

- A power of two has only **one set bit** in binary.
- x & (x - 1) turns off the lowest set bit, so:
  - If result is 0 → x was a power of 2.
  - Otherwise → it's not.

🔢 Dry Run Table (for x from 1 to 16 for brevity):

x	Binary of x	x-1	Binary of x-1	x & (x-1)	Is Power of 2?
1	00000001	0	00000000	00000000	✓ Yes
2	00000010	1	00000001	00000000	✓ Yes
3	00000011	2	00000010	00000010	✗ No
4	00000100	3	00000011	00000000	✓ Yes
5	00000101	4	00000100	00000100	✗ No
6	00000110	5	00000101	00000100	✗ No
7	00000111	6	00000110	00000110	✗ No
8	00001000	7	00000111	00000000	✓ Yes
9	00001001	8	00001000	00001000	✗ No
10	00001010	9	00001001	00001000	✗ No
11	00001011	10	00001010	00001010	✗ No
12	00001100	11	00001011	00001000	✗ No
13	00001101	12	00001100	00001100	✗ No
14	00001110	13	00001101	00001100	✗ No
15	00001111	14	00001110	00001110	✗ No
16	00010000	15	00001111	00000000	✓ Yes

✓ Output for i = 1 to 32:

The function will print:

```
1 is Power of two
2 is Power of two
3 is not Power of two
4 is Power of two
5 is not Power of two
...
32 is Power of two
```

Output:-

```
1 is Power of two
```

```
2 is Power of two
3 is not Power of two
4 is Power of two
5 is not Power of two
6 is not Power of two
7 is not Power of two
8 is Power of two
9 is not Power of two
10 is not Power of two
11 is not Power of two
12 is not Power of two
13 is not Power of two
14 is not Power of two
15 is not Power of two
16 is Power of two
17 is not Power of two
18 is not Power of two
19 is not Power of two
20 is not Power of two
21 is not Power of two
22 is not Power of two
23 is not Power of two
24 is not Power of two
25 is not Power of two
26 is not Power of two
27 is not Power of two
28 is not Power of two
29 is not Power of two
30 is not Power of two
31 is not Power of two
32 is Power of two
```

Subsets in C++

```
#include <iostream>
using namespace std;

int main() {
   int n = 4;
   for (int b = 0; b < (1 << n); b++) {
      cout << b << endl;
   }

   return 0;
}
```

You're generating all numbers from `0` to `2^n - 1` using bit manipulation!

🔍 Breakdown:

- `n = 4` → total combinations = `2^4 = 16`
- `(1 << n)` means `1` shifted left `n` times → equals `2^n`
- Loop runs from `0` to `15`, printing each value

🧮 Dry Run Table:

`b`	Binary of `b`
0	`0000`
1	`0001`
2	`0010`
3	`0011`
4	`0100`
5	`0101`
6	`0110`
7	`0111`
8	`1000`
9	`1001`
10	`1010`
11	`1011`
12	`1100`
13	`1101`
14	`1110`
15	`1111`

Output:-

```
0
```