Floats (continued)
Intro to C programming

Lesson plan

- Rounding
- FP operations and caveats
- C programming: overview
- C programming: bitwise operators

IEEE Floating Point

Normalized Encoding:

31	30 … 23	22 … 0
s	exp = E + 127	fraction (F)

$1 <= M < 2$, $M = (1.F)_2$

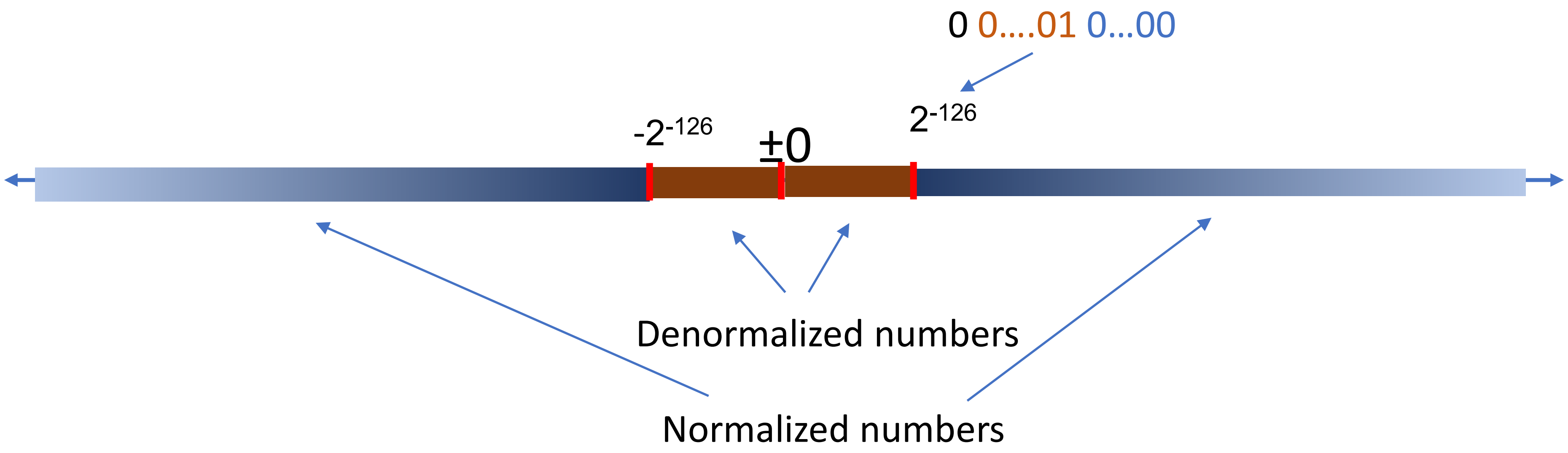
$\underline{+}M * 2^E$

Denormalized Encoding:

31	30 … 23	22 … 0
s	exp =0000 0000	fraction (F)

E = 1 – bias = -126

$0 <= M < 1$, $M = (0.F)_2$



FP: Rounding

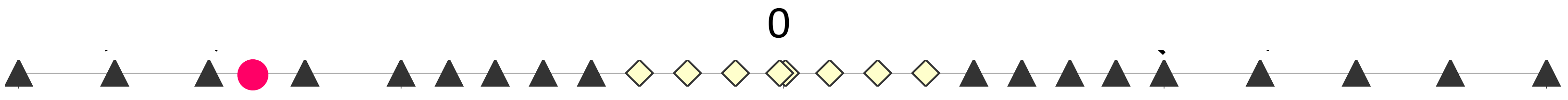
What if the result of computation is at ● ?

Rounding: Use the “closest” representable value *x’* for x.

4 modes:

- Round-down
- Round-up
- Round-toward-zero
- Round-to-nearest (Round-to-even in text book)

Round down vs. round up



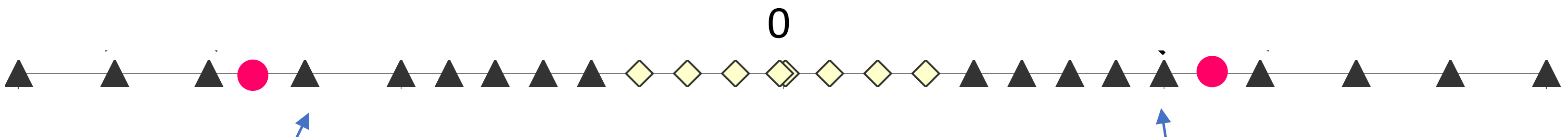
Round up rounds to the right

$\text{Round}(x) = x_+ \quad (x_+ >= x)$

Round down rounds to the left

$\text{Round}(x) = x_- \ (x_- <= x)$

Round towards zero



Rounds to the right if $x < 0$

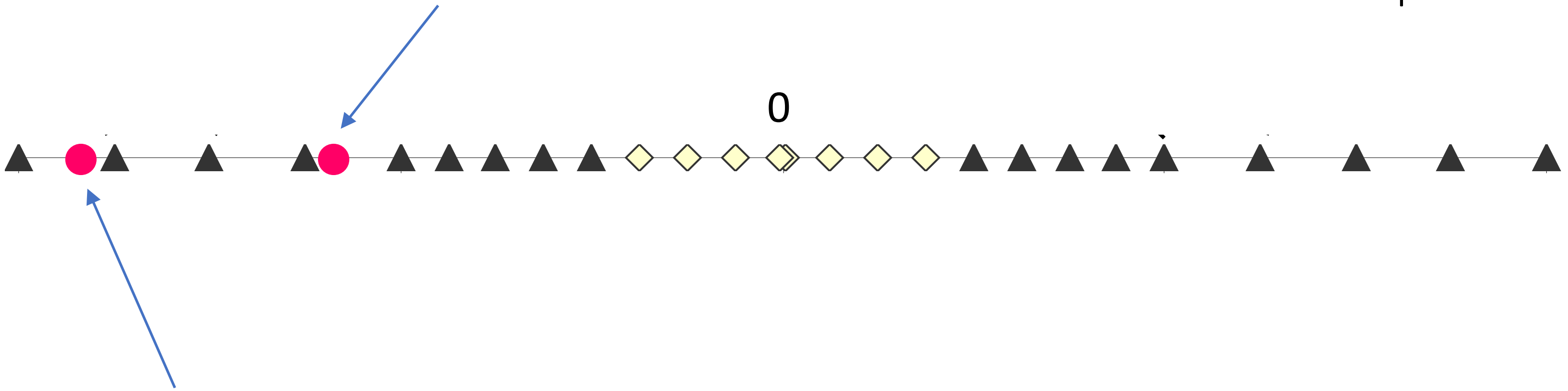
Round(x) = x_+ if $x < 0$

Rounds to the left if $x > 0$

Round(x) = x_- if $x < 0$

Round to nearest; ties to even

Round to the left if x_- is nearer to x than x_+



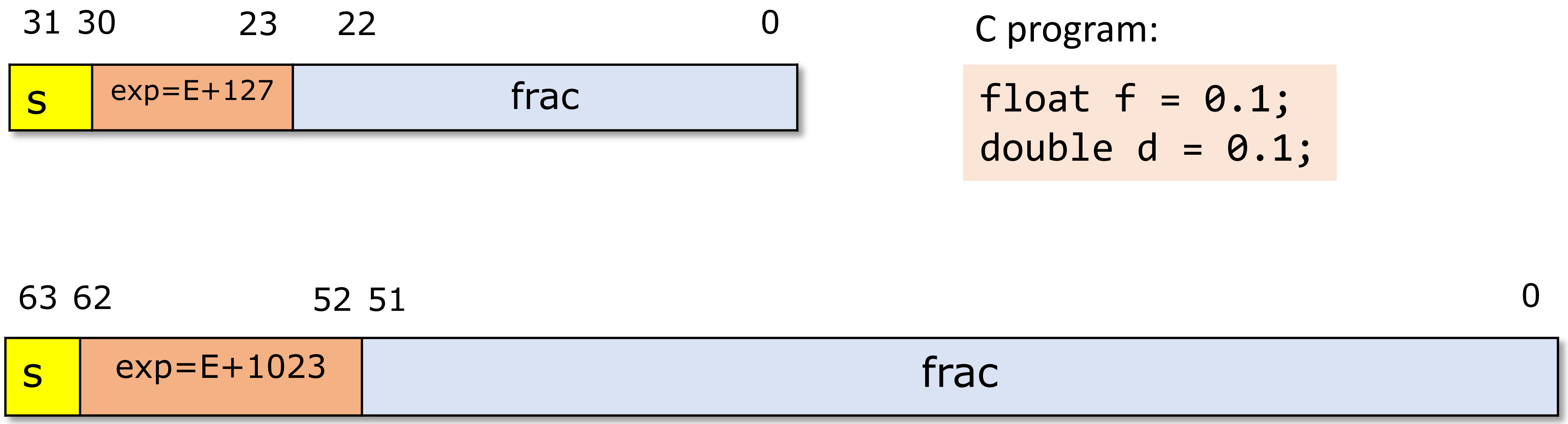
Round to the right if x_+ is nearer to x than x_-

In case of a tie, the one with its least significant bit equal to zero is chosen.

IEEE FP: single vs. double precision

single precision
(32 bits)

double precision
(64 bits)



C program:

```
float f = 0.1;
double d = 0.1;
```

- What's the highest precision? (aka intervals between two denormalized numbers?)
- What's the largest positive FP?

How does CPU know if data is FP or integer ?

4-byte data: 0x8001

Interpret as signed int:

Interpret as IEEE single-precision FP:

- CPU has separate registers for FPs and integers.
- CPU uses different instructions for FPs and integer operations.

Floating point operations

- Addition, subtraction, multiplication, division etc.
- Invalid operations (resulting in NaN):
  - 0/0
  - sqrt(-1)
  - $\infty + \infty$
- Divide by zero: $x/0 \rightarrow \infty$
- Caveats:
  - Overflow: Outside the range
  - Underflow: 0 < result < smallest denormalized value
  - Inexact: due to rounding

Floating point addition

- Commutative? x+y == y+x?
- Associative? (x+y)+z = x + (y+z)?
  - Rounding:

```
(3.14+1e10)-1e10 = 0
 3.14+(1e10-1e10) = 3.14
```

  - Overflow
- Every number has an additive inverse?
  - Yes, by flipping the sign.

Floating point multiplication

- Commutative? x* y == y*x?
- Associative? (x*y)*z = x * (y*z)?
  - Overflow:

    `(1e20*1e20)*1e-20= inf, 1e20*(1e20*1e-20) = 1e20`
  - Rounding
- Distributive? (x+y)*z = x*z + y*z?
  - `1e20*(1e20-1e20) = 0.0, 1e20*1e20 - 1e20*1e20 = NaN`

FP precision decreases as value gets larger

- Storing time in computer games as a FP?
- Precision diminishes as time gets bigger

FP value (decimal)	Time value	FP precision	Time precision
1	1 sec	1.19E-07	119 nanoseconds
100	~1.5 min	7.63E-06	7.63 microseconds
10 000	~3 hours	0.000977	.976 milliseconds
1000 000	~11 days	0.0625	62.5 milliseconds

Floating point trouble

- Comparing floats for equality is a bad idea!

```
float f = 0.1;
while (f != 1.0) {
          f += 0.1;
}
```

Floating point summary

- FP format is based on normalized exponential notation
- Floating points are tricky
  - Precision diminishes as magnitude grows
  - overflow, rounding error
- Many real world disasters due to FP trickiness
  - Patriot Missile failed to intercept due to rounding error (1991)
  - Ariane 5 explosion due to overflow in converting from double to int (1996)

Lesson plan

- Rounding
- FP operations and caveats
- C programming: overview
- C programming: bitwise operators

Python programmers

C programmers

C is an old programming language

C	Java	Python
1972	1995	2000 (2.0)
Procedure	Object oriented	Procedure & object oriented
Compiled to machine code, runs on bare machine	Compiled to bytecode, runs by another piece of software	Scripting language, interpreted by software
static type	static type	dynamic type
Manual memory management	Automatic memory management with GC	
Tiny standard library	Very Large library	Humongous library

Why learn C for CSO?

- C is a systems language
  - Language for writing OS and low-level code
  - Systems written in C:
    - Linux, Windows kernel, MacOS kernel
    - MySQL, Postgres
    - Apache webserver, NGIX
    - Java virtual machine, Python interpreter
- Why learning C for CSO?
  - simple, low-level, “close to the hardware”

The simplest C program: “Hello World”

```c
#include <stdio.h>

int main()
{
  printf("hello, world\n");
  return 0;
}
```

hello.c

Equivalent to “importing” a library package

A function “exported” by stdio.h

Compile: `gcc hello.c –o hello`

Run: `./hello`

If `–o` is not given, output executable file is a.out

C program with multiple files: naïve organization

```c
int sum(int x, int y)
{
   return x+y;
}
```

sum.c

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

Void test_sum()
{
  int r = sum(1,1);
  assert(r == 2);
}

int main()
{
   test_sum();
}
```

test.c

```c
#include <stdio.h>

int main()
{
  printf("sum=%d\n", sum(-1,1));
}
```

main.c

Compile:

```
gcc sum.c test.c –o test
```

```
gcc sum.c main.c
```

Run:

```
./test
```

```
./a.out
```

Sum.c compiled twice.
Wasteful

C program with multiple files: *.h vs *.c files

```
int sum(int x, int y)
{
   return x+y;
}
```

sum.c

```
int sum(int x, int y);
```

sum.h

Equivalent to “importing” a package

```
#include <stdio.h>
#include <assert.h>
#include “sum.h”

Void test_sum()
{
  int r = sum(1,1);
  assert(r == 2);
}

int main()
{
   test_sum();
}
```

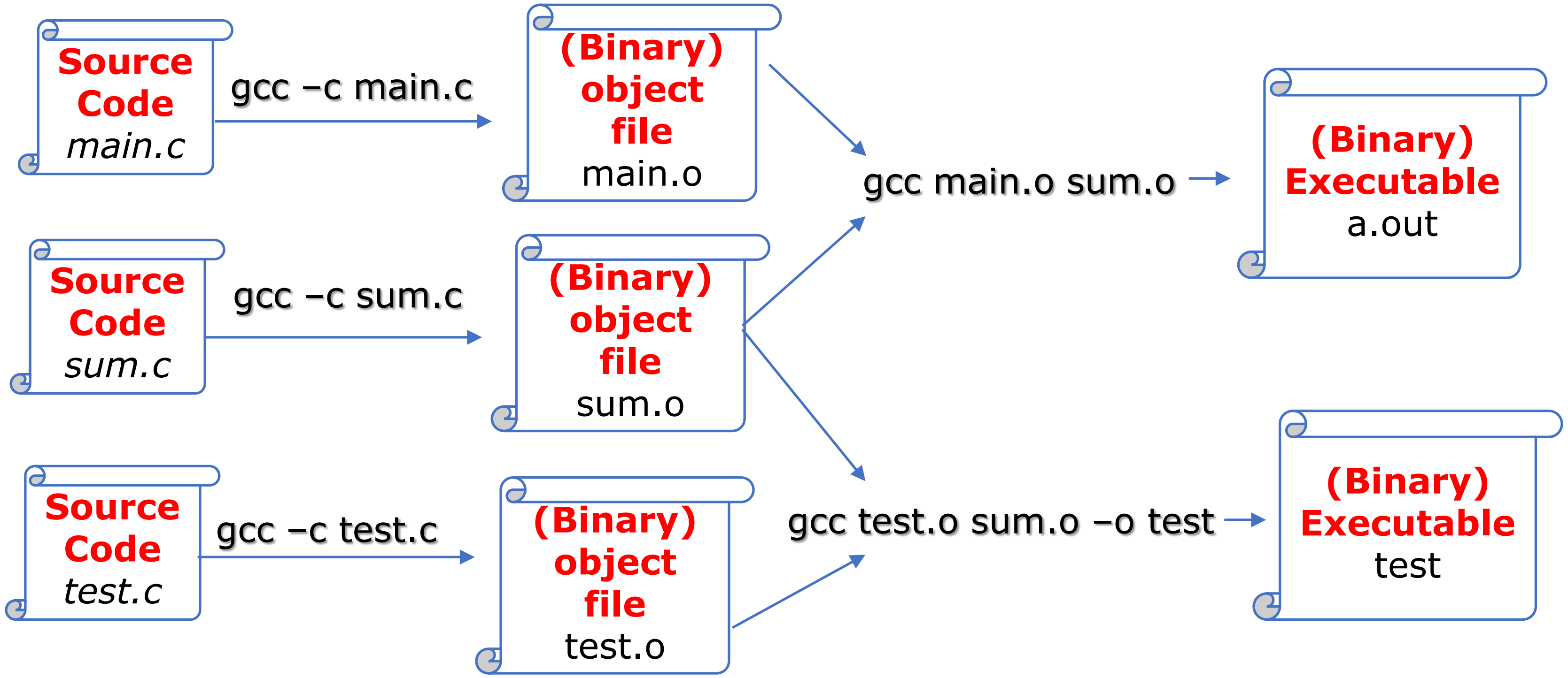
test.c

```
#include <stdio.h>
#include “sum.h”

int main()
{
  printf(“sum=%d\n”, sum(-1,1));
}
```

main.c

Compiling



C project uses the make tool to automate compiling with dependencies.

```
all: a.out test
test: test.o sum.o
    gcc $^ -o $@
a.out: main.c sum.o
    gcc $^ -o $@
%.o: %.c
    gcc –c $^
```

Basic C

- C's syntax is very similar to Java
  - Java borrowed its syntax from C

Initial value

⚠ If uninitialized, variable can have any value

Variable declaration: `int a = 1;`

Type *Name*

Primitive Types (64-bit machine)

Either a character or an intger

type	size (bytes)	example
(unsigned) char	1	char c = ‘a’
(unsigned) short	2	short s = 12
(unsigned) int	4	int i = 1
(unsigned) long	8	long l = 1
float	4	float f = 1.0
double	8	double d = 1.0
pointer	8	int *x = &i

Old C has no native boolean type. A non-zero integer represents true, a zero integer represents false

C99 has “bool” type, but one needs to include <stdbool.h>

Implicit conversion

```
int main()
{
    int a = -1;
    unsigned int b = 1;

    if (a < b) {
        printf("%d is smaller than %d\n", a, b);
    } else if (a > b) {
        printf("%d is larger than %d\n”, a, b);
    }
```

```
$gcc test.c
$./a.out
-1 is larger than 1
```

No compiler warning!

Compiler converts types to the one with the largest data type (e.g. char → unsigned char → int → unsigned int)

Implicit conversion

```
int main()
{
   int a = -1;
   unsigned int b = 1;

   if (a < b) {
        printf("%d is smaller than %d\n", a, b);
   } else if (a > b) {
        printf("%d is larger than %d\n”, a, b);
   }

   return 0;
}
```

-1 is implicitly cast to unsigned int $(4294967295)_{10}$

Explicit conversion (casting)

```
int main()
{
    int a = -1;
    unsigned int b = 1;

    if (a <  (int) b) {
        printf("%d is smaller than %d\n", a, b);
    } else if (a > (int) b) {
        printf("%d is larger than %d\n”, a, b);
    }

    return 0;
}
```

Operators

Arithmetic	+, -, *, /, %, ++, --
Relational	==, !=, >, <, >=, <=
Logical	&&, ||, !
Bitwise	&, |, ^, ~, >>, <<

Arithmetic, Relational and Logical operators are identical to java’s

Bitwise operator &

x	y	x&y
0	0	0
0	1	0
1	0	0
1	1	1

This is a truth table

Result of 0x69 & 0x55

$$\begin{array}{r} (\,0\;1\;1\;0\;1\;0\;0\;1\,)_2 \\ \&\;(\,0\;1\;0\;1\;0\;1\;0\;1\,)_2 \\ \hline (\,0\;1\;0\;0\;0\;0\;0\;1\,)_2 \end{array}$$

Example use of &

- & is often used to mask off bits
  - any bit & 0 = 0
  - any bit & 1 = unchanged

```
int clear_msb(int x) {
    return x & 0x7fffffff;
}
```

Bitwise operator |

x	y	x|y
0	0	0
0	1	1
1	0	1
1	1	1

Result of 0x69 | 0x55

$$
\begin{array}{r}
(\,0\;1\;1\;0\;1\;0\;0\;1\,)_2 \\
|\quad(\,0\;1\;0\;1\;0\;1\;0\;1\,)_2 \\
\hline
(\,0\;1\;1\;1\;1\;1\;0\;1\,)_2
\end{array}
$$

Example use of |

- | can be used to turn some bits on
  - any bit | 1 = 1
  - any bit | 0 = unchanged

```
int set_msb(int x) {
    return x | 0x80000000;
}
```

Bitwise operator ~

x	~x
0	1
1	0

result of ~0x69

$$\begin{array}{r} \sim (\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ )_2 \\ \hline (\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ )_2 \end{array}$$

Bitwise operator ^ (XOR)

x	y	x^y
0	0	0
0	1	1
1	0	1
1	1	0

result of 0x69^0x55

$$
\begin{array}{r}
(\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ )_2 \\
\wedge\ (\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ )_2 \\
\hline
(\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ )_2
\end{array}
$$

Bitwise operator <<

x << y, shift bit-vector x left by y positions

- Throw away bits shifted out on the left
- Fill in 0's on the right

	0 1 1 0 1 0 0 1
result of 0x69 << 3	0 1 0 0 1 0 0 0

Bitwise operator >>

- x >> y, shift bit-vector x right by y positions
  - Throw away bits shifted out on the right
  - (Logical shift) Fill with 0's on left

1 0 1 0 1 0 0 1

Logical result of 0xa9 >> 3 0 0 0 1 0 1 0 1

Bitwise operator >>

- x >> y, shift bit-vector x right by y positions
  - Throw away bits shifted out on the right
  - (Logical shift) Fill with 0's on the left
  - (Arithmetic shift) Replicate msb on the left

1 0 1 0 1 0 0 1

Arithmetic result of 0xa9 >> 3 1 1 1 1 0 1 0 1

Which shift is used in C ?

Arithmetic shift for signed numbers

Logical shifting on unsigned numbers

```
#include <stdio.h>
int main()
{
  int a = -1;
  unsigned int b = 1;
  printf("%d  %d\n", a>>10, b>>10);
}
```

```
Result: -1 0
```

Which shift is used?

Arithmetic shift for signed numbers

Logical shifting on unsigned numbers

```
#include <stdio.h>
int main()
{
  int a = -1;
  unsigned int b = 1;
  printf("%d  %d\n", (unsigned int)a>>10,
b>>10);
}
```

Result: 4194303 0

Example use of shift

```
         int                                        int
unsigned multiply_by_two(unsigned int x)
{
    return x << 1;
}
```

Example use of shift

```
   int                                          int
unsigned divide_by_two(unsigned int x)
{
    return x >> 1;
}
```

Example use of shift

```
// clear bit at position pos
// rightmost bit is at 0th pos

int clear_bit_at_pos(int x, int pos)
{
    unsigned int mask = 1 << pos;
    return x & (~mask);
}
```

Example use of shift

```
// set bit at position pos
// rightmost bit is at 0th pos

int set_bit_at_pos(int x, int pos)
{
    unsigned int mask = 1 << pos;
    return x | mask;
}
```