L7: Data Representations

2's completement

Bitwise Operators

Float memory representation

Evaluation sequence for logical operators

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Two's Complement Representation

Used to represent signed integers in C.

Positive numbers → same as binary.

Negative numbers → take 2’s complement (invert + add 1).

Two's Complement Example

8-bit system

+5:

00000101

-5:

Start with +5: 00000101

Invert: 11111010

Add 1: 11111011

So, -5 = 11111011_2

Two’s Complement Arithmetic

```c
#include <stdio.h>

int main() {
    char a = 5;   // 00000101
    char b = -5;  // 11111011

    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("a + b = %d\n", a + b); // should be 0
    return 0;
}
```

Two’s Complement – More Examples

Example: Represent -18 in 8 bits

+18 = 00010010

Invert → 11101101

Add 1 → 11101110

So, -18 = 11101110_2

Wrap-Around Example in Two’s Complement

```
#include <stdio.h>
int main() {
    char c = 127;   // 01111111
    printf("c = %d\n", c);
    c = c + 1;      // Overflow!
    printf("c after +1 = %d\n", c);
}
```

Bitwise Operators in C

- Operate **directly on the bits** of integers.
- Very useful in systems programming, embedded systems, optimization.

Operators:

- `&` (AND)
- `|` (OR)
- `^` (XOR)
- `~` (NOT / Complement)
- `<<` (Left Shift)
- `>>` (Right Shift)

Bitwise Example

```c
#include <stdio.h>

int main() {
    int a = 6, b = 3;  // a=0110, b=0011

    printf("a & b = %d\n", a & b); // 2
    printf("a | b = %d\n", a | b); // 7
    printf("a ^ b = %d\n", a ^ b); // 5
    printf("~a = %d\n", ~a);       // -7 (2's complement)
    printf("a << 1 = %d\n", a << 1); // 12
    printf("a >> 1 = %d\n", a >> 1); // 3
    return 0;
}
```

Practical Bitwise Example: Check Even/Odd

```
#include <stdio.h>
int main() {
    int n = 37;
    if(n & 1)
        printf("%d is Odd\n", n);
    else
        printf("%d is Even\n", n);
}
```

Bitwise Example: Swapping Two Numbers

```c
#include <stdio.h>
int main() {
    int x = 7, y = 12;
    printf("Before: x=%d, y=%d\n", x, y);

    x = x ^ y;
    y = x ^ y;
    x = x ^ y;

    printf("After: x=%d, y=%d\n", x, y);
}
```

Floating Point Representation (IEEE 754)

C uses IEEE 754 Standard for float (32-bit) and double (64-bit).

Sign (1 bit)	**Exponent (8 bits)**	**Mantissa (23 bits)**

$$(-1)^{\text{sign}} \times (1.\text{mantissa}) \times 2^{(\text{exponent}-127)}$$

Example

Number: 5.75

- Convert to binary:

  $5.75 = 101.11_2$

  $= 1.0111 \times 2^2$

Sign = 0 (positive)

Exponent = 127 + 2 = 129 = 10000001_2

Mantissa = 0111000...

Final 32-bit pattern:

```
0 10000001 01110000000000000000000
```

Float Representation Example – 0.15625

Convert decimal to binary fraction:

```
0.15625 × 2 = 0.3125 → 0
0.3125 × 2 = 0.625   → 0
0.625 × 2 = 1.25     → 1
0.25 × 2 = 0.5       → 0
0.5 × 2 = 1.0        → 1
```

Binary = 0.00101_2 Normalize: 1.01×2^{-3}. Sign = 0

Exponent = $127 - 3 = 124 = 01111100_2$

Mantissa = 0100000…

Final bit pattern: `0 01111100 01000000000000000000000`

Another Float Example – Negative Number

Number: -7.5

Binary: $111.1_2 = 1.111 \times 2^2$

Sign = 1

Exponent = 127 + 2 = 129 = 10000001_2

Mantissa = 1110000…

Final representation:

1 10000001 11100000000000000000000

The Problem

- Floating point numbers (`float`, `double`) are stored in **binary (IEEE 754)**.
- Many decimal values **cannot be represented exactly**.
- Equality checks (`==`) often fail due to **rounding errors**.

Example: Equality Failure

```c
#include <stdio.h>

int main() {
    float x = 0.1f;
    float y = 0.2f;
    float z = 0.3f;

    if (x + y == z) {
        printf("Equal\n");
    } else {
        printf("Not Equal\n");
    }
}
```

Why?

Internally (IEEE 754, 32-bit float):

0.1 → 0.100000001490116120

0.2 → 0.200000002980232240

0.3 → 0.300000011920928960

x + y = 0.3000000119…

z = 0.3000000119…

Tiny differences cause == to fail.

The Fix: Use an Epsilon

Instead of ==, check if the difference is within tolerance:

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

int main() {
    float x = 0.1f, y = 0.2f, z = 0.3f;
    float epsilon = 1e-6;

    if (fabs((x + y) - z) < epsilon) {
        printf("Approximately Equal\n");
    } else {
        printf("Not Equal\n");
    }
}
```

Logical Operators in C

- **AND (&&)**

  True if *both* operands are true.

- **OR (||)**

  True if *at least one* operand is true.

- **NOT (!)**

  Negates a condition.

Evaluation Order – Short-Circuit

In C, evaluation is **left-to-right** with **short-circuiting**:

- `A && B` → If `A` is false, **B is not evaluated**.
- `A || B` → If `A` is true, **B is not evaluated**.

This is called **short-circuit evaluation**.

Example: Short-Circuit AND

```c
#include <stdio.h>
int main() {
    int x = 0;
    if (x != 0 && (10 / x > 1)) {
        printf("Condition true\n");
    } else {
        printf("Condition false\n");
    }
}
```

Example: Short-Circuit OR

```c
#include <stdio.h>
int main() {
    int x = 5;
    if (x == 5 || (10 / x == 2)) {
        printf("True branch\n");
    }
}
```

Example with Side Effects

```c
#include <stdio.h>
int main() {
    int a = 0, b = 1;
    if (a++ > 0 && b++) {
        printf("Inside if\n");
    }
    printf("a = %d, b = %d\n", a, b);
}
```