

C++ Fixed Precision Output

```
cout<<fixed<<setprecision(4)<<34.22224<<endl;
```

This prints the number **34.2222**.

Here's why, straight up:

- **fixed** tells C++ to use **fixed-point notation**, not scientific.
- **setprecision(4)** with **fixed** means **4 digits after the decimal point**, not total digits.
- **34.22224** rounded to 4 decimal places becomes **34.2222** (because the 5th decimal digit is 4, so no rounding up).

Output:

```
34.2222
```

If that last digit were **5** or more (e.g. **34.22225**), then it would round to **34.2223**.

how tp round in slq and round upto 3 digit and percentage also

I'll keep this practical and interview-ready.

1. Rounding numbers in SQL (up to 3 digits)

Use **ROUND()**.

```
sql
SELECT ROUND(34.22224, 3);
```

Output:

```
34.222
```

If the next digit is ≥ 5 , it rounds up.

Example:

```
sql
SELECT ROUND(34.22256, 3);
```

Output:

```
34.223
```

2. Rounding without rounding (just cut digits)

Use `TRUNCATE()` (MySQL / PostgreSQL).

```
sql
SELECT TRUNCATE(34.22256, 3);
```

Output:

```
34.222
```

👉 Interview tip: `ROUND` changes value, `TRUNCATE` does not.

3. Round and show percentage (very common in interviews)

Suppose you want $\text{percentage} = (\text{obtained} / \text{total}) * 100$

```
sql
SELECT ROUND((45.0 / 50) * 100, 3) AS percentage;
```

Output:

```
90.000
```

⚠️ Important: use `45.0` or cast to decimal, otherwise **integer division** will break your result.

4. Percentage with column values

```
sql
SELECT
    ROUND((marks * 100.0) / total_marks, 3) AS percentage
FROM students;
```

5. Round + percentage symbol (for reports only)

sql

```
SELECT
    CONCAT(ROUND((marks * 100.0) / total_marks, 3), '%') AS percentage
FROM students;
```

👉 Don't do this if the value is needed for calculations later.

6. Database-wise quick reference

DB	Round	Truncate
MySQL	ROUND(x,3)	TRUNCATE(x,3)
PostgreSQL	ROUND(x,3)	TRUNC(x,3)
SQL Server	ROUND(x,3)	ROUND(x,3,1)
Oracle	ROUND(x,3)	TRUNC(x,3)

Interview line you can say confidently:

"In SQL, **ROUND(value, n)** rounds the number to **n** decimal places, while **TRUNCATE/TRUNC** removes extra decimals without rounding. For percentages, we multiply by 100 and round, ensuring decimal division to avoid integer truncation."

If you want, I can throw **2-3 tricky interview questions** on rounding that people usually mess up.

what about ceil and floor and string in sql

Straight answer, interview-level clarity.

1. CEIL / CEILING in SQL

Purpose: Always rounds **UP** to the nearest integer.

sql

```
SELECT CEIL(4.1); -- 5
SELECT CEILING(4.0);-- 4
SELECT CEIL(-4.1); -- -4
```

👉 Key point interviewers like:

- **CEIL moves towards $+\infty$**
- For negative numbers, it becomes *less negative*

2. FLOOR in SQL

Purpose: Always rounds **DOWN** to the nearest integer.

sql

```
SELECT FLOOR(4.9);    -- 4
SELECT FLOOR(4.0);    -- 4
SELECT FLOOR(-4.1);   -- -5
```

👉 Key point:

- **FLOOR** moves **towards $-\infty$**
- For negative numbers, it becomes *more negative*

3. ROUND vs CEIL vs FLOOR (very common comparison)

Value	ROUND(·)	CEIL	FLOOR
4.2	4	5	4
4.5	5	5	4
-4.2	-4	-4	-5

💡 One-liner:

- **ROUND** → nearest value
- **CEIL** → always up
- **FLOOR** → always down

4. Using CEIL / FLOOR with percentage

Example: billing or grading logic.

sql

```
SELECT CEIL((45.0 / 50) * 100) AS percentage;
```

Output:

90

sql

```
SELECT FLOOR((44.9 / 50) * 100);
```

Output:

5. String functions in SQL (important ones)

a) Concatenate strings

```
sql  
SELECT CONCAT('Hello', ' ', 'World');
```

SQL Server alternative:

```
sql  
SELECT 'Hello' + ' ' + 'World';
```

b) Length of string

```
sql  
SELECT LENGTH('Yuvraj'); -- MySQL / PostgreSQL  
SELECT LEN('Yuvraj'); -- SQL Server
```

c) Convert string case

```
sql  
SELECT UPPER('sql'); -- SQL  
SELECT LOWER('SQL'); -- sql
```

d) Substring

```
sql  
SELECT SUBSTRING('DATABASE', 1, 4);
```

Output:

```
powershell  
DATA
```

(MySQL supports **SUBSTR** also)

e) Trim spaces

```
sql

SELECT TRIM(' SQL ');
SELECT LTRIM(' SQL');
SELECT RTRIM('SQL '');
```

f) Replace string

```
sql

SELECT REPLACE('Hello World', 'World', 'SQL');
```

Output:

```
sql

Hello SQL
```

g) String + Number (very common mistake)

✗ Wrong:

```
sql

SELECT 'Marks: ' + 90;
```

✓ Correct:

```
sql

SELECT CONCAT('Marks: ', 90);
```

6. Using numbers + strings (real example)

```
sql

SELECT
    CONCAT(
        ROUND((marks * 100.0) / total, 2),
        '%'
    ) AS percentage
FROM students;
```

7. Interview one-liner to remember

“CEIL always rounds a number up, FLOOR always rounds it down, and ROUND rounds to the nearest value. SQL provides string functions like CONCAT, SUBSTRING, UPPER, LOWER, TRIM, and REPLACE to manipulate text data.”

If you want next, I can give you **10 SQL MCQs** mixing **CEIL**, **FLOOR**, **ROUND**, and string traps—these are exactly what interviewers use.

```
# Write your MySQL query statement below
SELECT c.contest_id, ROUND(count(c.user_id)*100/(SELECT count(*) from Users), 2) as percentage
from Register c
GROUP BY c.contest_id ORDER BY percentage DESC, c.contest_id ASC
```

do this also by all possible way like subqueris,view,cte and with and all so i learn all in mYSQL

Good question. This is exactly how you *actually* learn SQL instead of memorizing syntax. I'll show **all practical ways in MySQL**, clean and correct, and I'll point out **when to use what**.

Your **goal**:

For each **contest_id**, find the **percentage of users registered**, rounded to 2 decimals, ordered by percentage desc, contest_id asc.

✓ 0. Corrected & Safe Version of Your Query (baseline)

Your logic is right, but make it **numerically safe** (avoid integer division):

```
sql

SELECT
    c.contest_id,
    ROUND(COUNT(c.user_id) * 100.0 / (SELECT COUNT(*) FROM Users), 2) AS percentage
FROM Register c
GROUP BY c.contest_id
ORDER BY percentage DESC, c.contest_id ASC;
```

Why **100.0**?

→ forces decimal math. Interviewers *will* check this.

1 Using Scalar Subquery (what you already did)

This subquery runs once (optimizer handles it well).

```
sql

SELECT
    contest_id,
    ROUND(COUNT(user_id) * 100.0 / (SELECT COUNT(*) FROM Users), 2) AS percentage
FROM Register
GROUP BY contest_id
ORDER BY percentage DESC, contest_id;
```

✓ Simple

✗ Less readable when logic grows

2 Using Derived Table (Subquery in FROM)

Cleaner and more reusable.

```
sql

SELECT
    r.contest_id,
    ROUND(COUNT(r.user_id) * 100.0 / u.total_users, 2) AS percentage
FROM Register r
CROSS JOIN (
    SELECT COUNT(*) AS total_users FROM Users
) u
GROUP BY r.contest_id
ORDER BY percentage DESC, r.contest_id;
```

👉 CROSS JOIN is intentional here (single-row table).

Interview gold line:

“I use a derived table when I need the result multiple times or for clarity.”

3 Using CTE (WITH clause) ✓ (MySQL 8+)

Most readable. Best for interviews.

```
sql

WITH total_users AS (
    SELECT COUNT(*) AS total FROM Users
)
SELECT
    r.contest_id,
    ROUND(COUNT(r.user_id) * 100.0 / t.total, 2) AS percentage
FROM Register r
JOIN total_users t
GROUP BY r.contest_id
ORDER BY percentage DESC, r.contest_id;
```

- ✓ Clean
- ✓ Reusable
- ✓ Scales well

↗ Say this in interview:

“CTEs improve readability and debugging compared to nested subqueries.”

4 Using Multiple CTEs (step-by-step thinking)

This shows **senior-level clarity**.

```
sql

WITH
total_users AS (
```

```

    SELECT COUNT(*) AS total FROM Users
),
contest_counts AS (
    SELECT contest_id, COUNT(user_id) AS registered_users
    FROM Register
    GROUP BY contest_id
)
SELECT
    c.contest_id,
    ROUND(c.registered_users * 100.0 / t.total, 2) AS percentage
FROM contest_counts c
JOIN total_users t
ORDER BY percentage DESC, c.contest_id;

```

🔥 This is *interviewer-impressive*.

5 Using a VIEW (when query is reused)

Create once, query many times.

```

sql

CREATE VIEW contest_registration_percentage AS
SELECT
    r.contest_id,
    ROUND(COUNT(r.user_id) * 100.0 / u.total_users, 2) AS percentage
FROM Register r
CROSS JOIN (
    SELECT COUNT(*) AS total_users FROM Users
) u
GROUP BY r.contest_id;

```

Then:

```

sql

SELECT *
FROM contest_registration_percentage
ORDER BY percentage DESC, contest_id;

```

- ✓ Reusable
- ✗ Slight overhead
- ✗ Not ideal for frequently changing logic

6 Using Window Function (MySQL 8+)

This is advanced and clean.

```

sql

SELECT
    contest_id,
    ROUND(
        COUNT(user_id) * 100.0 / COUNT(*) OVER (),
        2
    ) AS percentage
FROM Register

```

```
GROUP BY contest_id
ORDER BY percentage DESC, contest_id;
```

⚠ Only works if **every user appears once per contest in Register.**

Interview line:

"Window functions avoid subqueries and are efficient for analytical queries."

7 Performance & Interview Comparison

Method	Readability	Performance	Interview Value
Simple subquery	★★	★★★	★★
Derived table	★★★	★★★	★★★
CTE	★★★★	★★★	★★★★
Multiple CTE	★★★★★	★★★	★★★★★
View	★★★	★★	★★★
Window function	★★★★	★★★★	★★★★

🔑 Final interview-ready summary (say this)

"I can solve this using scalar subqueries, derived tables, CTEs, views, or window functions. For readability and maintainability, I prefer CTEs. For reuse, I use views. For analytical queries, window functions are efficient."

If you want next, I can:

- break this **line-by-line**
- give **tricky edge cases**
- convert this to **PostgreSQL / SQL Server**
- or throw **LeetCode-style follow-ups**

Your call.