✅ 1. Trigonometric Functions in C++

All trigonometric functions come from the header:

```
#include <cmath>
```

👉 Common math functions:

Function	Meaning
`sin(x)`	Sine
`cos(x)`	Cosine
`tan(x)`	Tangent
`asin(x)`	Inverse sine
`acos(x)`	Inverse cosine
`atan(x)`	Inverse tangent
`atan2(y, x)`	Angle from x, y coordinates
`sinh(x)`	Hyperbolic sine
`cosh(x)`	Hyperbolic cosine

⚠️ 2. VERY IMPORTANT: C++ uses Radians, not Degrees

So if you want to use degrees (like 30°, 45°, 60°):

Convert degrees → radians

```
double radians = degrees * M_PI / 180.0;
```

⭐ Example 1: Calculate sin, cos, tan of 30°

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

```cpp
int main() {
    double deg = 30;
    double rad = deg * M_PI / 180.0;  // convert to radians

    cout << "sin(30°) = " << sin(rad) << endl;
    cout << "cos(30°) = " << cos(rad) << endl;
    cout << "tan(30°) = " << tan(rad) << endl;

    return 0;
}
```

Expected output:

```
sin(30°) = 0.5
cos(30°) = 0.866025
tan(30°) = 0.57735
```

⭐ Example 2: Convert radian → degree

```cpp
double rad = 1.0472;
double deg = rad * 180.0 / M_PI;
```

⭐ Example 3: Find angle from sine value

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

int main() {
    double x = 0.5;

    double rad = asin(x);
    double deg = rad * 180.0 / M_PI;

    cout << "Angle = " << deg << " degrees" << endl;
}
```

⭐ Example 4: Distance between 2 points using trigonometry

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

int main() {
    double x1=0, y1=0;
    double x2=3, y2=4;

    double distance = sqrt(pow(x2-x1,2) + pow(y2-y1,2));

    cout << "Distance: " << distance;
}
```

Result:

```
Distance: 5
```

⭐ Example 5: Angle between two points

```cpp
double angle = atan2(y2 - y1, x2 - x1) * 180.0 / M_PI;
```

🎯 Summary

In C++ trigonometry:

- Include:

  ```cpp
  #include <cmath>
  ```

- Use radians.
- Convert degree ↔ radian.
- Use sin(), cos(), tan(), asin(), acos(), atan() etc.

✅ Hyperbolic Functions in C++

C++ provides all hyperbolic math functions in:

```cpp
#include <cmath>
```

👉 Available Hyperbolic Functions

Function	Meaning
`sinh(x)`	Hyperbolic sine
`cosh(x)`	Hyperbolic cosine
`tanh(x)`	Hyperbolic tangent
`asinh(x)`	Inverse hyperbolic sine
`acosh(x)`	Inverse hyperbolic cosine
`atanh(x)`	Inverse hyperbolic tangent

🧠 Basic Formulas (Important)

Hyperbolic Sine

[
$\sinh(x) = \frac{e^x - e^{-x}}{2}$
]

Hyperbolic Cosine

[
$\cosh(x) = \frac{e^x + e^{-x}}{2}$
]

Hyperbolic Tangent

[
$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$
]

⭐ Example 1: Using hyperbolic functions

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

int main() {
    double x = 1.0;

    cout << "sinh(1) = " << sinh(x) << endl;
    cout << "cosh(1) = " << cosh(x) << endl;
```

```
    cout << "tanh(1) = " << tanh(x) << endl;

    return 0;
}
```

Output:

```
sinh(1) = 1.1752
cosh(1) = 1.54308
tanh(1) = 0.761594
```

⭐ Example 2: Inverse hyperbolic functions

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

int main() {
    double x = 0.5;

    cout << "asinh(0.5) = " << asinh(x) << endl;
    cout << "acosh(1.5) = " << acosh(1.5) << endl;
    cout << "atanh(0.3) = " << atanh(0.3) << endl;

    return 0;
}
```

⭐ Example 3: Proving formulas using C++

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

int main() {
    double x = 2;

    double hyper_formula = (exp(x) - exp(-x)) / 2; // sinh formula
    double actual = sinh(x);

    cout << "Formula: " << hyper_formula << endl;
```

```
    cout << "sinh(x): " << actual << endl;
}
```

⭐ Example 4: Difference between Trigonometric vs Hyperbolic Functions

Trigonometric	Hyperbolic
sin(x)	sinh(x)
cos(x)	cosh(x)
tan(x)	tanh(x)
Uses unit circle	Uses hyperbola
Periodic	Not periodic

🎯 Summary

✔ Hyperbolic functions are available in `<cmath>`
✔ Use: `sinh()`, `cosh()`, `tanh()`
✔ Inverse: `asinh()`, `acosh()`, `atanh()`
✔ No need to convert degrees/radians — hyperbolic functions always use **real values directly**

✅ 1. Exponential Functions in C++

C++ provides exponential functions in:

```
#include <cmath>
```

👉 Common exponential functions

Function	Meaning
`exp(x)`	(e^x)
`exp2(x)`	(2^x)
`expm1(x)`	(e^x - 1) (more accurate for small x)
`pow(a, b)`	(a^b)

⭐ Example 1: Using `exp(x)`

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

int main() {
    double x = 2;

    cout << "e^2 = " << exp(x) << endl;

    return 0;
}
```

Output:

```
e^2 = 7.38906
```

⭐ Example 2: Power function `pow(a, b)`

```
cout << pow(3, 4);   // 3^4 = 81
```

✅ 2. Logarithmic Functions in C++

👉 Common logarithmic functions

Function	Meaning
`log(x)`	Natural log (base e)
`log10(x)`	Log base 10
`log2(x)`	Log base 2
`log1p(x)`	log(1 + x), accurate for small x

⭐ Example 3: Natural log

```cpp
double x = 7.389;
cout << log(x);  // approx 2
```

⭐ Example 4: Log base 10

```cpp
cout << log10(1000);  // 3
```

⭐ Example 5: Changing log base

To calculate log base *b*:

[
$$\log_b(x) = \frac{\log(x)}{\log(b)}$$
]

Example:

```cpp
double log_base_5 = log(125) / log(5);  // =3
```

🎯 3. Important Formula Connections

Exponential ↔ Logarithm inverse

[
$$e^{\log(x)} = x$$
]
[
$$\log(e^x) = x$$
]

Logarithm properties:

[
$$\log(ab) = \log(a) + \log(b)$$
]
[
$$\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$$
]
[
$$\log(a^b) = b\log(a)$$
]

⭐ Example 6: Check inverse relationship

```
double x = 5;
cout << exp(log(x));   // should print 5
```

⭐ Example 7: Compound interest (exponential growth)

```
double amount = P * exp(r * t);
```

⭐ Example 8: Logarithmic scale example

```
double intensity_ratio = log10(I2 / I1);
```

Used in:

- Sound (decibels)
- Earthquake magnitude
- pH scale

⭐ Example 9: Solve equations using log

Solve: (3^x = 81)

```
double x = log(81) / log(3);
cout << x;  // 4
```

🎯 Summary

Exponential Functions

- `exp(x)` → (e^x)
- `pow(a, b)` → (a^b)
- `exp2(x)` → (2^x)

Logarithmic Functions

- `log(x)` → natural log
- `log10(x)` → base-10 log
- `log2(x)` → base-2 log
- `log1p(x)` → log(1+x)

🔥 1. exp(x) — Exponential Function

Computes:

[
e^x
]

Example:

```
double x = exp(2.0);  // e^2
```

🔥 2. frexp(x, &exp) — Split floating-point into mantissa + exponent

Breaks number into:

[
$x = m \times 2^e$
]

Returns **mantissa**, stores **exponent**.

Example:

```
int e;
double m = frexp(16.0, &e);
// m = 0.5, e = 5 → 0.5 * 2^5 = 16
```

🔥 3. ldexp(m, e) — Build floating number from mantissa & exponent

Opposite of `frexp`.

[
$ldexp(m, e) = m \times 2^e$
]

Example:

```
double x = ldexp(0.5, 5); // = 16
```

🔥 4. log(x) — Natural Logarithm

Computes:

[
$\ln(x)$
]

Example:

```
double x = log(7.389); // approx 2
```

🔥 5. log10(x) — Common Logarithm (Base-10)

[
$\log_{10}(x)$
]

Example:

```
double x = log10(1000); // = 3
```

🔥 6. modf(x, &intpart) — Split fractional + integer part

Example:

```
double intp;
double frac = modf(12.34, &intp);
// intp = 12, frac = 0.34
```

🔥 7. exp2(x) — Compute (2^x)

[
2^x
]

Example:

```
double x = exp2(5); // 32
```

🔥 8. expm1(x) — More accurate version of `exp(x) – 1`

When x is very small, exp(x) – 1 loses precision.

Example:

```
double x = expm1(1e-7);
```

🔥 9. ilogb(x) — Integer logarithm base 2

Returns exponent `e` such that:

[
$x = m \times 2^e$
]

(Like `logb(x)` but integer only)

Example:

```
int e = ilogb(16.0); // returns 4
```

🔥 10. log1p(x) — log(1 + x) (high precision)

More accurate when x is near 0.

Example:

```
double x = log1p(1e-8);
```

🔥 11. log2(x) — Binary logarithm

[
$\log_2(x)$
]

Example:

```
double x = log2(32); // 5
```

🔥 12. logb(x) — Floating-point base logarithm

Returns exponent `e` such that:

[
$x = m \times 2^e$
]

But unlike `ilogb`, it returns **floating** value.

Example:

```
double e = logb(16.0); // 4.0
```

🔥 13. scalbn(x, n) — Multiply by 2^n efficiently

[
$x \times 2^n$
]

Example:

```
double v = scalbn(1.5, 3);
// 1.5 * 2^3 = 12
```

🔥 14. scalbln(x, n) — Same as `scalbn`, but exponent is long int

Used for very large exponents.

Example:

```
double v = scalbln(1.5, 1000);
```

🎯 Summary Table

Function	Purpose
exp	Compute (e^x)
frexp	Split into (mantissa, exponent)
ldexp	Build from (mantissa × 2^exponent)
log	Natural log
log10	Log base 10
modf	Split integer + fractional parts
exp2	Compute (2^x)
expm1	Compute (e^x - 1) precisely
ilogb	Integer log base 2
log1p	Compute log(1 + x) precisely
log2	Log base 2
logb	Floating base-2 logarithm
scalbn	Multiply by (2^n)
scalbln	Multiply by (2^n) (long exponent)

🔥 1. pow(x, y) — Raise to Power

Computes:

[
x^y
]

Example:

```
double a = pow(3, 4);  // 3^4 = 81
double b = pow(2.5, 3); // 15.625
```

🔥 2. sqrt(x) — Square Root

Computes:

[
$\sqrt{x}$
]

Example:

```
double r = sqrt(25);   // 5
```

If x is negative → result is NaN (not a number).

🔥 3. cbrt(x) — Cubic Root

Computes:

[
$\sqrt[3]{x}$
]

Works for **negative numbers** too.

Example:

```
double r = cbrt(27);   // 3
double n = cbrt(-8);   // -2
```

🔥 4. hypot(x, y) — Compute Hypotenuse

Computes:

[
$\sqrt{x^2 + y^2}$
]

Safe version of:

```
sqrt(x*x + y*y)
```

⚡ `hypot` avoids overflow/underflow → more accurate.

Example:

```
double h = hypot(3, 4); // 5 (Pythagoras)
```

Real use:

Distance between two points:

```
double d = hypot(x2 - x1, y2 - y1);
```

🎯 Summary Table

Function	Meaning	Example
pow(x, y)	(x^y)	`pow(2, 5) = 32`
sqrt(x)	(\sqrt{x})	`sqrt(49) = 7`
cbrt(x)	(\sqrt[3]{x})	`cbrt(-8) = -2`
hypot(x, y)	(\sqrt{x^2 + y^2})	`hypot(3,4) = 5`

🔥 1. ceil(x) — Round UP

Rounds **toward +∞**

Example:

```
ceil(3.2);   // 4
ceil(-3.2);  // -3
```

🔥 2. floor(x) — Round DOWN

Rounds **toward −∞**

Example:

```
floor(3.8);   // 3
floor(-3.8);  // -4
```

🔥 3. fmod(x, y) — Floating-point remainder

Computes:

[
$\text{remainder} = x - n \cdot y$
]

Where `n = trunc(x / y)`

Example:

```
fmod(7.5, 2.0); // 1.5
```

🔥 4. trunc(x) — Remove fractional part

Rounds toward **zero**.

Example:

```
trunc(5.89);  // 5
trunc(-5.89); // -5
```

🔥 5. round(x) — Round to nearest integer

Rules:

- .5 and above → round UP
- below .5 → round DOWN
- Ties → round **away from zero**

Example:

```
round(3.4);  // 3
round(3.5);  // 4
```

```
round(-3.5); // -4
```

🔥 6. lround(x) — Round then cast to long

Same rounding rules as `round()`.

Example:

```
long n = lround(4.6); // 5
```

🔥 7. llround(x) — Round then cast to long long

Example:

```
long long n = llround(4.6); // 5
```

🔥 8. rint(x) — Round to integral value

Uses current **floating-point rounding mode** (like bankers rounding in some systems).

⚠ Does **not** cast to int.

Example:

```
rint(2.5);  // result depends on system mode
```

🔥 9. lrint(x) — rint + cast to long

Example:

```
long n = lrint(3.2);
```

🔥 10. llrint(x) — rint + cast to long long

Example:

```
long long n = llrint(3.2);
```

🔥 11. nearbyint(x) — Round like rint BUT never raises floating-point exceptions

Works like `rint()` but safer.

Example:

```
nearbyint(2.7); // 3
```

🔥 12. remainder(x, y) — IEC 60559 remainder

Compute:

[
$\text{remainder} = x - n \cdot y$
]

Where `n` = nearest integer to `x/y`
(Ties → even)

More exact than fmod.

Example:

```
remainder(7.5, 2.0); // -0.5 (different from fmod)
```

🔥 13. remquo(x, y, &quo)

Returns:

- remainder (like `remainder()`)
- **lower bits** of quotient stored in `quo`

Example:

```c
int q;
double r = remquo(7.5, 2.0, &q);
// r = -0.5
// q = quotient information (implementation-dependent)
```

Useful in high-precision algorithms.

🎯 Summary Table (Fast Revision)

Function	What it does
ceil	Round up
floor	Round down
fmod	Floating remainder (uses trunc)
trunc	Remove decimal, toward zero
round	Round to nearest (away from zero on tie)
lround	round + cast to long
llround	round + cast to long long
rint	Round using system mode
lrint	rint + cast to long
llrint	rint + cast to long long
nearbyint	rint without exceptions
remainder	IEC 60559 remainder (tie-even)
remquo	remainder + quotient bits

⭐ Minimum, Maximum, Difference Functions

🔥 1. fdim(x, y) — Positive Difference

Returns:

[
$\text{fdim}(x, y) =$
\begin{cases}
x - y, & x > y \$ $0, & x \le y
\end{cases}
]

Example:

```
fdim(10, 4);    // 6
fdim(4, 10);    // 0
```

🔥 2. fmax(x, y) — Maximum Value

Returns the **larger** of two numbers.

Example:

```
fmax(3.2, 8.5); // 8.5
```

🔥 3. fmin(x, y) — Minimum Value

Returns the **smaller** of two numbers.

Example:

```
fmin(3.2, 8.5); // 3.2
```

⭐ Other Useful Functions

🔥 4. fabs(x) — Absolute Value (floating-point)

Returns:

[
$|x|$
]

Example:

```
fabs(-5.7); // 5.7
```

🔥 5. abs(x) — Absolute Value (overloaded)

Works for:

- `int`
- `long`
- `long long`
- `float`
- `double`
- etc.

Example:

```
abs(-10);      // 10
abs(-12.5);    // 12.5
```

⚠ For floating-point values `fabs()` is slightly more specific, but both work.

🔥 6. fma(x, y, z) — Fused Multiply-Add

Computes:

[
$x \times y + z$
]

in one operation, giving MUCH higher precision.

Example:

```
double v = fma(3.0, 2.0, 5.0);
// (3 * 2) + 5 = 11
```

Why fma is important?

Normal:

```
x*y + z
```

may cause **rounding twice**
BUT:

```
fma(x, y, z)
```

does **multiply + add with a single rounding**, more accurate.

🎯 Final Summary Table

Function	Purpose	Example
fdim(x, y)	Positive difference	`fdim(10,4)=6`
fmax(x,y)	Larger of two values	`fmax(3,8)=8`
fmin(x,y)	Smaller of two values	`fmin(3,8)=3`
fabs(x)	Absolute value (float)	`fabs(-5.7)=5.7`
abs(x)	Absolute value (int/float)	`abs(-10)=10`
fma(x,y,z)	Precise `(x*y)+z`	`fma(3,2,5)=11`
