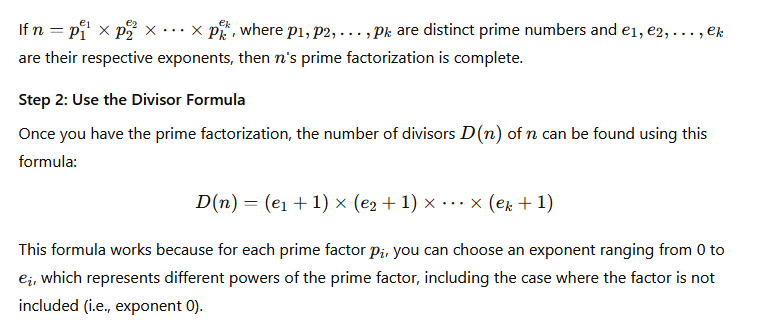
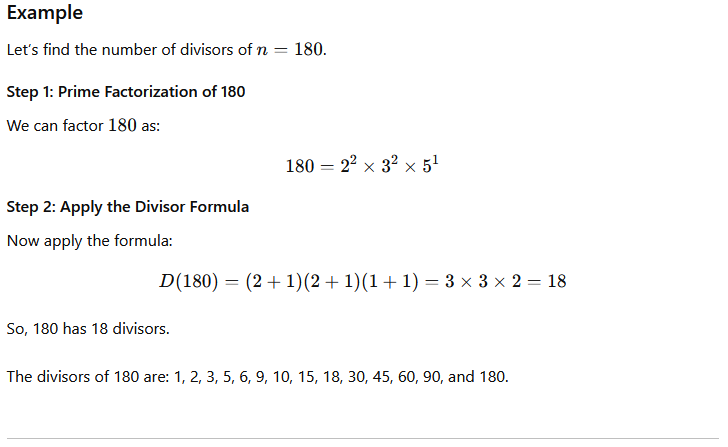
**Number of Divisors**

Let's say you have a number n, and you want to find how many divisors it has.

**Step 1: Prime Factorization**





**1. Code**

int numberOfDivisors(int n) {

int count = 1; // To store the number of divisors

int power; // To store the power of each prime factor

// Check for number of 2's that divide n

power = 0;

while (n % 2 == 0) {

n /= 2;

power++;

}

count \*= (power + 1); // Multiply the divisor count by (power + 1)

// Now check for odd numbers

for (int i = 3; i <= sqrt(n); i += 2) {

power = 0;

while (n % i == 0) {

n /= i;

power++;

}

count \*= (power + 1);

}

// If n is a prime number greater than 2

if (n > 2) {

count \*= 2;

}

return count;

}

**2. Simple way to code for calculating the number of divisors**

int numberOfDivisors(int n) {

int count = 0;

for (int i = 1; i <= sqrt(n); i++) {

if (n % i == 0) {

if (i == n / i) {

count += 1; // If divisors are the same, count only once

} else {

count += 2; // Count both divisors i and n/i

}

}

}

return count;

}

**3. Sieve-Based Approach for Multiple Queries**

If you need to find divisors for many numbers (e.g., multiple queries), a **precomputation** method like the **Sieve of Eratosthenes** can be adapted to calculate the number of divisors for all numbers up to a certain limit.

**Algorithm**:

* Use a sieve to precompute the divisor count for every number up to a limit.
* For each multiple of i, increment the divisor count.

vector<int> sieveDivisors(int maxN) {

vector<int> divisors(maxN + 1, 0);

for (int i = 1; i <= maxN; i++) {

for (int j = i; j <= maxN; j += i) {

divisors[j]++;

}

}

return divisors;

}

**Time Complexity**:

* The time complexity of this method is O(nlog⁡n)O(n \log n)O(nlogn) for precomputing divisors up to nnn.
* After precomputation, each query runs in O(1)O(1)O(1) time.

**Advantages**:

* If you need to answer multiple queries about divisors of different numbers, this is very efficient since the result for each number is computed in constant time after the sieve is built.

**Disadvantages**:

* It requires more memory and is only practical if you need results for many numbers up to a fixed upper limit.

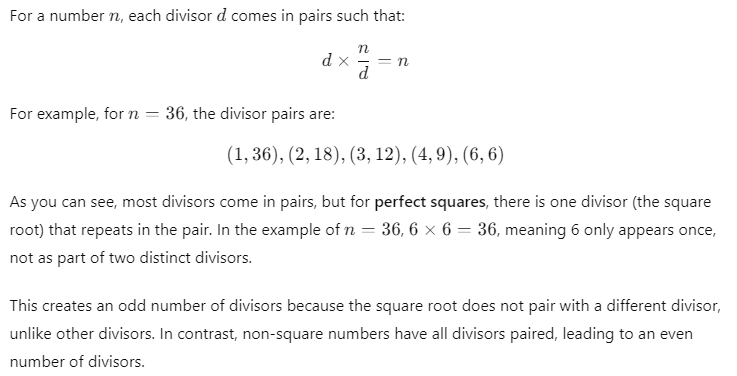
**Best Approach Based on Use Case**

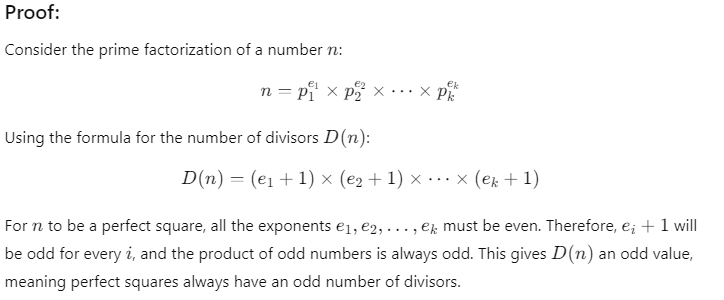
**1 For Single Queries:** The simple divisor counting approach using O(√n) without prime factorization is often the most straightforward and efficient method unless the number is very large or prime-heavy.

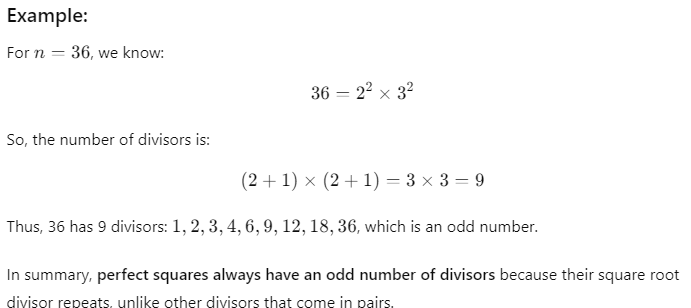
**2 For Multiple Queries:** The sieve-based approach is ideal when you need divisor counts for many numbers. It requires O(n log n) time for precomputation but answers each query in O(1).

**3 Handling Very Large Number:** If you're working with very large numbers (e.g., n≥1018n \geq 10^{18}n≥1018), algorithms like Pollard's rho algorithm or efficient prime factorization algorithms might be necessary for factoring large primes. However, these are advanced techniques not typically needed for everyday divisor counting.

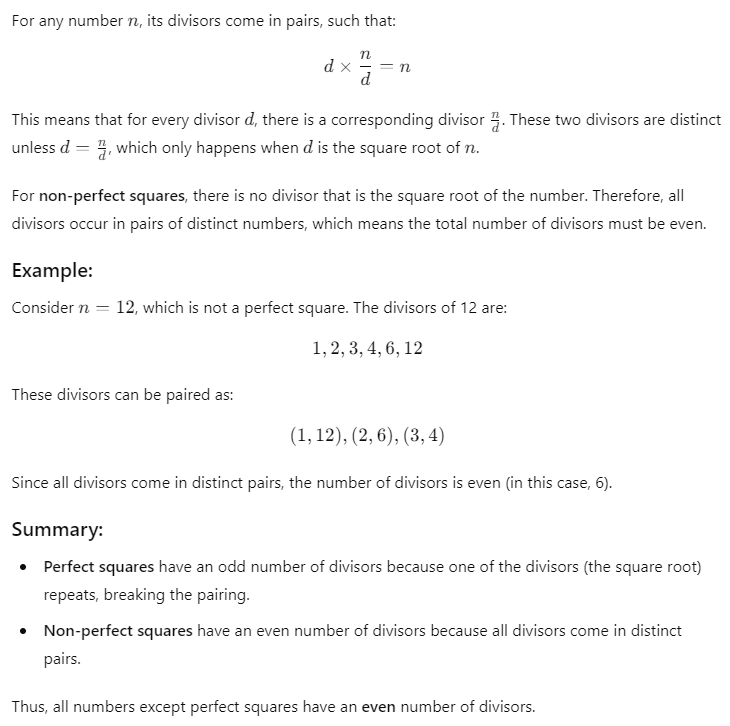
**A perfect square always has an odd number of divisors.**







**A non-perfect square always has an even number of divisors.**



# unordered\_map vs. map in C++

In C++, `unordered\_map` and `map` are two container types that allow storing key-value pairs, but they differ in terms of underlying data structures, ordering, performance, and specific use cases. Understanding these differences is crucial when deciding which container to use in various programming scenarios.

## 1. Data Structure

- \*\*`map`\*\*: Implemented as a Red-Black Tree, a self-balancing binary search tree. This structure allows it to keep keys in sorted order by default.  
- \*\*`unordered\_map`\*\*: Implemented as a hash table, where keys are stored in an arbitrary order based on their hash values. Hash tables do not maintain any sorted order.

## 2. Ordering

- \*\*`map`\*\*: Maintains keys in sorted order, which allows ordered traversal of keys. Keys are ordered based on their natural ordering (or a custom comparator if provided).  
- \*\*`unordered\_map`\*\*: Does not maintain any specific order for keys; they are stored based on their hash values. Thus, traversing an `unordered\_map` does not yield keys in any particular sequence.

## 3. Performance

- \*\*`map`\*\*:  
 - \*\*Insertion, Deletion, and Access\*\*: Since `map` uses a balanced tree, these operations have a complexity of O(log n) due to tree balancing operations.  
 - \*\*Best Use Case\*\*: `map` is ideal for scenarios where ordered traversal or range-based queries are needed.  
- \*\*`unordered\_map`\*\*:  
 - \*\*Insertion, Deletion, and Access\*\*: `unordered\_map` offers average-case O(1) time complexity for these operations due to hashing, but worst-case O(n) complexity if multiple keys hash to the same bucket.  
 - \*\*Best Use Case\*\*: `unordered\_map` is preferred when the order of elements doesn't matter, and constant-time operations are desired for efficient access and storage.

## 4. Memory Usage

- \*\*`map`\*\*: Requires more memory to store tree pointers and balancing overhead as it maintains a binary search tree structure.  
- \*\*`unordered\_map`\*\*: May use additional memory for managing hash collisions, but generally has lower overhead compared to `map`.

## 5. Use Cases

- \*\*`map`\*\*: Best suited for situations where data needs to be kept in sorted order or for scenarios involving range-based queries and lexicographical ordering.  
- \*\*`unordered\_map`\*\*: Ideal for fast access applications like frequency counting, lookup tables, and cases where order does not impact the outcome.

## 6. Summary and Example Code

In summary, `unordered\_map` is typically faster than `map` due to its O(1) average access time, but it should be used when order is not required. `map` offers O(log n) access time and is useful when data needs to be ordered.  
  
Below is an example illustrating the use of both containers:

```cpp  
#include <iostream>  
#include <map>  
#include <unordered\_map>  
  
int main() {  
 // Example with map  
 std::map<int, std::string> ordered\_map;  
 ordered\_map[1] = "one";  
 ordered\_map[3] = "three";  
 ordered\_map[2] = "two";  
  
 std::cout << "Ordered map (sorted by key):\n";  
 for (const auto& pair : ordered\_map) {  
 std::cout << pair.first << ": " << pair.second << std::endl;  
 }  
  
 // Example with unordered\_map  
 std::unordered\_map<int, std::string> unordered\_map;  
 unordered\_map[1] = "one";  
 unordered\_map[3] = "three";  
 unordered\_map[2] = "two";  
  
 std::cout << "Unordered map (arbitrary order):\n";  
 for (const auto& pair : unordered\_map) {  
 std::cout << pair.first << ": " << pair.second << std::endl;  
 }  
 return 0;  
}  
```