using namespace std

En el C++ moderno, se desaconseja el uso de using namespace std por razones relacionadas con el *alcance de los nombres* y la *colisión* de nombres. Vamos a analizarlo en detalle:

1. Colisión de Nombres (Name Collisions)
Cuando utilizas using namespace std;, todos los nombres del espacio de nombres std (*que pertenece a la biblioteca estándar de C++*) se importan al ámbito global. Esto significa que las funciones, clases, y otras definiciones que están dentro de std pueden entrar en conflicto con nombres que declares en tu propio código o con nombres de otras bibliotecas.

Ejemplo:

```
#include <iostream>

using namespace std;

void cout() {
    // Esta es la función aquí va tú código...
}

int main() {
    cout << "Hello, World!" << endl; // Error: ambigüedad entre std::cout y cout
    return 0;
}
```

En este caso, hay un conflicto entre la función cout() que has definido y std::cout. El compilador no sabe a cuál te refieres, lo que genera un error de ambigüedad.

En **aplicaciones grandes**, o cuando trabajas con múltiples bibliotecas de terceros, es mucho más probable que se produzcan este tipo de colisiones de nombres, lo que puede causar errores difíciles de depurar.

2. Globalización del Espacio de Nombres
Cuando utilizas using namespace std; *en la parte superior de tu archivo*, esencialmente importas todo el espacio de nombres std a todo el archivo. Esto puede no parecer un gran problema en proyectos pequeños, pero en proyectos más grandes puede crear problemas a largo plazo al aumentar el riesgo de conflictos de nombres y dificultar la legibilidad del código.

El objetivo de los espacios de nombres (namespaces) en C++ es precisamente evitar colisiones de nombres. Si globalizas el espacio de nombres std, pierdes esa ventaja, anulando la razón por la que existen los namespaces.

3. Mejor Control con using Específico
En lugar de usar using namespace std;, es mucho más recomendable importar solo los nombres que realmente vas a utilizar, de esta manera:

```
#include <iostream>

using std::cout;
using std::endl;

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

Esto reduce significativamente el riesgo de colisión de nombres y hace que el código sea más claro, ya que es evidente que solo estás utilizando std::cout y std::endl del espacio de nombres std.

4. Mejora de la Legibilidad

Al ser explícito con el uso de std::, el lector del código tiene una referencia clara de que se están utilizando elementos de la biblioteca estándar, lo cual mejora la legibilidad y la mantenibilidad del código. Por ejemplo, en lugar de importar todo el namespace std, prefieres escribir std:: antes de los componentes de la biblioteca estándar (como std::vector, std::string, etc.).

Resumen:
using namespace std; introduce todos los nombres de std en el espacio de nombres global, lo que aumenta el riesgo de colisiones de nombres y confusión en proyectos grandes o con bibliotecas de terceros.
using std::cout; es preferible *porque limita el alcance del nombre* a los elementos específicos que estás utilizando, evitando posibles colisiones.

El C++ moderno fomenta la claridad y la seguridad en la gestión de nombres, por lo que se aconseja no utilizar using namespace std para evitar problemas futuros.

En C++, muchos elementos de la Biblioteca Estándar pertenecen al espacio de nombres std. A continuación te proporciono una lista de los elementos más comunes que se encuentran dentro del espacio de nombres std y que usualmente utilizamos con la notación std::.

1. Contenedores (Containers)
Los contenedores son clases de ***plantillas que permiten almacenar y manipular*** colecciones de datos. Algunos de los más importantes son:

std::vector – Vector dinámico.
std::array – Arreglo de tamaño fijo (C++11).
std::deque – Doble cola, permite inserciones y eliminaciones eficientes en ambos extremos.
std::list – Lista doblemente enlazada.
std::forward_list – Lista simplemente enlazada (C++11).
std::set – Conjunto de elementos únicos, ordenados.
std::unordered_set – Conjunto de elementos únicos, no ordenados (C++11).
std::map – Mapa asociativo de pares clave-valor, ordenado.
std::unordered_map – Mapa asociativo no ordenado (C++11).
std::multiset – Conjunto ordenado que permite duplicados.
std::multimap – Mapa que permite múltiples claves con el mismo valor.

2. Algoritmos
Los algoritmos son funciones que realizan operaciones comunes como buscar, ordenar y modificar contenedores.

std::sort – Ordena elementos en un rango.
std::find – Busca un elemento en un rango.
std::for_each – Aplica una función a cada elemento de un rango.
std::accumulate – Suma los elementos de un rango.
std::copy – Copia elementos de un rango a otro.
std::reverse – Invierte los elementos en un rango.
std::transform – Aplica una operación a cada elemento y almacena el resultado en otro rango.

3. Strings y Streams
Los **streams** son utilizados para la entrada y salida, mientras que las std::string son la clase para manejar cadenas de texto.

std::string – Cadenas de caracteres de longitud variable.
std::wstring – Cadenas de caracteres anchos (wide characters).
std::cout – Flujo de salida estándar (para imprimir en consola).
std::cin – Flujo de entrada estándar (para leer desde consola).
std::cerr – Flujo de error estándar.
std::ostringstream – Stream para escribir en un string.
std::istringstream – Stream para leer desde un string.

4. Funciones y Funcionales (Functional)
C++ provee utilidades para manejar funciones, punteros a funciones y funciones lambda.

std::function – Función genérica que puede contener cualquier objeto llamable (C++11).
std::bind – Vincula argumentos a una función o función miembro (C++11).
std::mem_fn – Convierte un puntero a función miembro en un objeto funcional (C++11).
std::hash – Función hash para contenedores como unordered_map.

5. Smart Pointers (Punteros Inteligentes)
C++ proporciona punteros inteligentes que ayudan a manejar la memoria de manera automática.

std::unique_ptr – Puntero único, sólo puede haber un propietario del recurso (C++11).
std::shared_ptr – Puntero compartido, varios punteros pueden compartir el mismo recurso (C++11).
std::weak_ptr – Puntero débil, no participa en la gestión de la vida del recurso (C++11).
std::make_unique – Crea un unique_ptr de manera segura (C++14).
std::make_shared – Crea un shared_ptr de manera eficiente.

6. Excepciones y Manejo de Errores
Las clases para el manejo de excepciones también pertenecen al espacio de nombres std.

std::exception – Clase base para todas las excepciones.
std::runtime_error – Excepción para errores en tiempo de ejecución.
std::logic_error – Excepción para errores lógicos en el programa.
std::out_of_range – Excepción lanzada cuando un índice está fuera del rango válido.
std::invalid_argument – Excepción lanzada cuando se pasa un argumento inválido.

7. Tuplas y Pares (Tuples and Pairs)
Estas estructuras permiten almacenar múltiples valores en un solo objeto.

std::pair – Almacena dos elementos de tipos diferentes.
std::tuple – Almacena múltiples elementos de diferentes tipos (C++11).
std::make_pair – Crea un par.
std::make_tuple – Crea una tupla.

8. Iteradores
Los iteradores permiten recorrer contenedores de una manera genérica.

std::iterator – Clase base para crear iteradores personalizados.
std::begin – Devuelve un iterador al inicio de un contenedor (C++11).
std::end – Devuelve un iterador al final de un contenedor (C++11).
std::reverse_iterator – Iterador inverso para recorrer un contenedor en orden inverso.

9. Multithreading (C++11 en adelante)
C++ introdujo soporte para multithreading en C++11.

std::thread – Clase para crear y gestionar hilos.
std::mutex – Proporciona mecanismos de bloqueo para la sincronización de hilos.
std::lock_guard – Garantiza que un mutex sea bloqueado y desbloqueado correctamente.
std::future – Representa el resultado de una operación asincrónica.
std::promise – Utilizado para proporcionar un valor a un future.

10. Utilidades Misceláneas
Algunos componentes adicionales que se encuentran en std son:

std::move – Función para indicar una transferencia de recursos (C++11).
std::forward – Mantiene la cualificación de referencia de los argumentos (C++11).
std::swap – Intercambia el valor de dos objetos.
std::chrono – Biblioteca para trabajar con tiempo y duraciones (C++11).
std::optional – Contenedor que puede o no contener un valor (C++17).
std::variant – Contenedor de tipos múltiples (C++17).
std::any – Puede contener cualquier tipo (C++17).

11. Contenedores de Pilas y Colas
Para la manipulación de datos en estilo FIFO y LIFO:

std::stack – **Pilas (LIFO)**.
std::queue – **Colas (FIFO)**.
std::priority_queue – Cola de prioridad.

12. Numerical Utilities
C++ también proporciona utilidades matemáticas y numéricas.

std::complex – Números complejos.
std::valarray – Array especializado para operaciones matemáticas rápidas.
std::numeric_limits – Proporciona límites de los tipos numéricos.

13. Random (Generación de números aleatorios)
El manejo de números aleatorios en C++ moderno:

std::random_device – Fuente de números aleatorios.
std::mt19937 – Motor basado en Mersenne Twister para la generación de números aleatorios.
std::uniform_int_distribution – Distribución de enteros uniformes.
std::normal_distribution – Distribución normal (Gaussiana).

14. Biblioteca de Entrada/Salida de Archivos
Funciones para trabajar con archivos:

std::ifstream – Para leer desde archivos.
std::ofstream – Para escribir en archivos.
std::fstream – Para lectura y escritura combinada en archivos.

Esta lista incluye los elementos más importantes que utilizan el espacio de nombres std en C++.