Operadores new y delete

1. Operador new

El operador ***new se utiliza para asignar memoria dinámicamente en el heap (montón)*** para variables de cualquier tipo, incluidos tipos de datos primitivos y objetos de clases. Cuando usas new, el operador asigna suficiente memoria para almacenar un objeto o una estructura y devuelve un puntero al primer byte del área asignada.

Sintaxis básica:

```
tipo* puntero = new tipo;
```

Asignación de un valor inicial:

```
int* ptr = new int(5);    // Asigna un entero con valor 5
```

Asignación de un array dinámico:

```
int* array = new int[10];  // Asigna un array de 10 enteros
```

Ejemplo:

```
#include <iostream>

int main() {
        int* ptr = new int(10);    // Asigna un entero dinámicamente con valor 10
        std::cout << "Valor: " << *ptr << std::endl;  // Accede al valor asignado
       delete ptr;  // Libera la memoria

        return 0;
}
```

2. Operador delete

El operador ***delete se utiliza para liberar la memoria asignada dinámicamente que fue reservada con new***. Es fundamental liberar la memoria para evitar fugas de memoria (memory leaks), donde la memoria ya no es accesible pero tampoco es reutilizable.

Sintaxis básica:

```
delete puntero;
```

Liberación de un array dinámico:

Si usaste new[] para asignar un array dinámico, debes usar delete[] para liberar esa memoria.

```
delete[] array;
```

Ejemplo:

```
#include <iostream>

int main() {
        int* array = new int[5];    // Asigna un array de 5 enteros

        // Inicializa el array
        for(int i = 0; i < 5; i++) {
                array[i] = i * 2;
        }

        // Imprime los valores
        for(int i = 0; i < 5; i++) {
                std::cout << "Valor en array[" << i << "]: " << array[i] << std::endl;
        }

        delete[] array;    // Libera la memoria asignada al array

        return 0;
}
```

3. Detalles Importantes y Buenas Prácticas

Verificación de new: A partir de C++11, el operador new lanza una excepción std::bad_alloc si no puede asignar la memoria solicitada. Antes de C++11, se debía verificar si el puntero retornado era nullptr para asegurar que la asignación fue exitosa.

```
try {
        int* ptr = new int[1000000000000];    // Intento de asignar mucha memoria
} catch (std::bad_alloc& e) {
        std::cerr << "Error de asignación de memoria: " << e.what() << std::endl;
}
```

Evitar fugas de memoria: Siempre debes emparejar cada new con un delete correspondiente, y cada new[] con un delete[]. No hacerlo puede resultar en fugas de memoria.

Uso de punteros inteligentes: En C++ moderno, se recomienda utilizar punteros inteligentes (std::unique_ptr, std::shared_ptr) en lugar de gestionar la memoria manualmente con new y delete. Los punteros inteligentes gestionan automáticamente la liberación de memoria, lo que ayuda a evitar errores comunes.

```
 #include <memory>

int main() {
        std::unique_ptr<int> ptr = std::make_unique<int>(10);
        std::cout << "Valor: " << *ptr << std::endl;

        return 0;  // La memoria se libera automáticamente cuando el puntero sale del alcance
}
```

4. Memoria Fragmentada y Reutilización
El uso ineficiente de new y delete puede llevar a la fragmentación de la memoria, donde la memoria libre se encuentra en bloques no contiguos. Esto puede hacer que, con el tiempo, sea difícil encontrar un bloque grande de memoria contigua para nuevas asignaciones.

Para evitar esto:

Reutiliza la memoria: Siempre que sea posible, reutiliza la memoria ya asignada en lugar de asignar y liberar memoria constantemente.
Usa técnicas de optimización de memoria: En aplicaciones que requieren un uso intensivo de memoria, puedes considerar el uso de allocators personalizados u optimizaciones específicas para reducir la fragmentación.

Resumen

new: Asigna memoria dinámica en el heap.
delete: Libera la memoria asignada por new.
new[] y delete[]: Asignan y liberan arrays dinámicos.