1 Nuevos elementos de C++11 para manejo de memoria inteligente

C++11 introdujo los **smart pointers** (`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`) para automatizar la gestión de memoria y evitar fugas. `std::unique_ptr` gestiona propiedad exclusiva, `std::shared_ptr` permite múltiples propietarios mediante conteo de referencias, y `std::weak_ptr` evita ciclos de referencia al no incrementar dicho conteo. Estos recursos se inicializan preferiblemente con `std::make_shared` y `std::make_unique`, que optimizan la asignación de memoria y previenen errores en contextos con excepciones.

La filosofía de C++ se basa en **RAII** (Resource Acquisition Is Initialization), donde los recursos se adquieren durante la construcción y se liberan automáticamente en la destrucción, garantizando seguridad y evitando fugas.

2 Alias y simplificación de tipos

El keyword `using` permite crear alias para tipos complejos, mejorando la legibilidad. Por ejemplo, simplificar nombres de plantillas anidadas o punteros, facilitando el mantenimiento del código sin alterar su funcionalidad.

3 Constructores y operadores

Los constructores por defecto inicializan miembros con valores predeterminados, mientras que el de copia crea réplicas independientes. El constructor de movimiento transfiere recursos de objetos temporales (*rvalues*), optimizando el rendimiento. Los operadores de asignación siguen lógicas similares: el de copia emplea *copy-and-swap* para seguridad ante excepciones, y el de movimiento transfiere recursos, dejando el objeto fuente en estado válido pero indefinido.

Un *lvalue* es un objeto con ubicación de memoria persistente (variables), mientras que un *rvalue* es un temporal. `std::move` convierte *lvalues* en *rvalues*, habilitando la semántica de movimiento.

4 Inicialización y estructuras de control

La inicialización con `{}` (*uniform initialization*) previene conversiones estrechas (*narrowing conversions*) y ofrece consistencia en contenedores, clases y tipos fundamentales. En contraste, `()` se usa para llamadas a funciones o constructores explícitos. `std::for_each` con lambdas permite aplicar operaciones a rangos de elementos, mejorando la expresividad al definir comportamientos inline sin funciones externas.

5 Gestión de memoria y excepciones

Los destructores no son necesarios cuando se usan *smart pointers*, pero son cruciales para recursos no gestionados por RAII (archivos, sockets). Los *raw pointers* persisten en interoperabilidad con APIs de C o optimizaciones de bajo nivel. `noexcept` indica que una función no lanzará excepciones, permitiendo optimizaciones del compilador y garantizando seguridad en operaciones críticas como movimientos de recursos.

6 Inferencia de tipos y genericidad

`auto` deduce tipos en tiempo de compilación, `decltype` obtiene el tipo de una expresión, y `decltype(auto)` combina ambas lógicas para contextos complejos. Los *templates* permiten código genérico, aceptando parámetros de tipo, valores no tipo o plantillas. La especialización de *templates* adapta implementaciones para tipos concretos, optimizando o modificando comportamientos.

7 Arrays y constructores

Los arrays *built-in* tienen tamaño fijo y gestión manual, mientras que `std::vector` y `std::array` ofrecen tamaño dinámico y seguridad. Los constructores inicializan miembros mediante listas de inicialización, evitando estados inconsistentes. El paso de parámetros por valor copia datos (costoso para objetos grandes), por referencia (`const T&`) evita copias, y por puntero permite modificar el original o manejar opcionalidad.

8 Funciones y operadores

Las funciones `inline` sugieren al compilador insertar código directamente, reduciendo *overhead*. Las funciones `const` no modifican el estado del objeto, y la sintaxis `const T x` o `T const x` declara variables inmutables. La redefinición de operadores como `[]` y `+` requiere devolver referencias para permitir asignaciones o crear nuevos objetos combinando operandos.

9 Uniones y tipos especializados

Las `union` almacenan múltiples tipos en la misma dirección de memoria, útiles en optimizaciones o interoperabilidad, aunque requieren manejo cuidadoso para evitar inconsistencias. Su uso es común en sistemas embebidos o al interactuar con hardware.