
UNIVERSIDAD
DE ANTIOQUIA

1 8 0 3

**Nombre de la institución:** Universidad de Antioquia

**Facultad o Departamento:** Ingeniería

**Título del informe:** Mercado de Estadías Hogareñas UdeAStay

**Autor:** Julian David Sanchez, Luis Fernando Castañeda

**Asignatura:** Informática II

**Facilitador o asesor:** Augusto salazar

**fecha:** 28/05/2025

**a. Análisis del problema y consideraciones para la alternativa de solución propuesta.**

El problema central de UdeAStay es modelar un mercado de alquileres de propiedades a corto plazo, lo que implica gestionar la interacción entre tres entidades principales: Anfitriones, Huéspedes y Alojamientos, además de las Reservaciones. Las consideraciones clave durante el análisis fueron:

1. **Modelado de Entidades del Mundo Real:**

    - **Anfitriones y Huéspedes:** Ambos son tipos de usuarios con atributos comunes (documento, contraseña, antigüedad, puntuación) pero con roles y funcionalidades distintas.
    - **Alojamientos:** Propiedades ofrecidas por anfitriones, con características específicas (nombre, precio, amenidades, ubicación) y un estado de disponibilidad.
    - **Reservaciones:** Transacciones que vinculan a un huésped con un alojamiento para un rango de fechas. Deben ser únicas y no superponerse.
    - **Fechas:** Crucial para la lógica de disponibilidad y la duración de las reservas. Requiere validación y operaciones de cálculo de días.
    - **Histórico:** Un registro para las reservas pasadas o anuladas, fundamental para auditoría o análisis futuro.
2. **Manejo de Colecciones Dinámicas:**

    - El número de anfitriones, huéspedes, alojamientos y reservas no es fijo. La solución debe permitir agregar y eliminar dinámicamente estos objetos.
    - **Consideración clave:** La restricción de **no usar contenedores STL** (como `std::vector`, `std::map`, `std::list`) implicó la necesidad de implementar todas las colecciones utilizando **arreglos dinámicos de punteros** (`new[]`, `delete[]`) y gestionar manualmente su tamaño y capacidad. Esto aumenta la complejidad del manejo de memoria pero es un requisito fundamental del desafío.
3. **Persistencia de Datos:**

    - La información del sistema debe persistir entre ejecuciones.

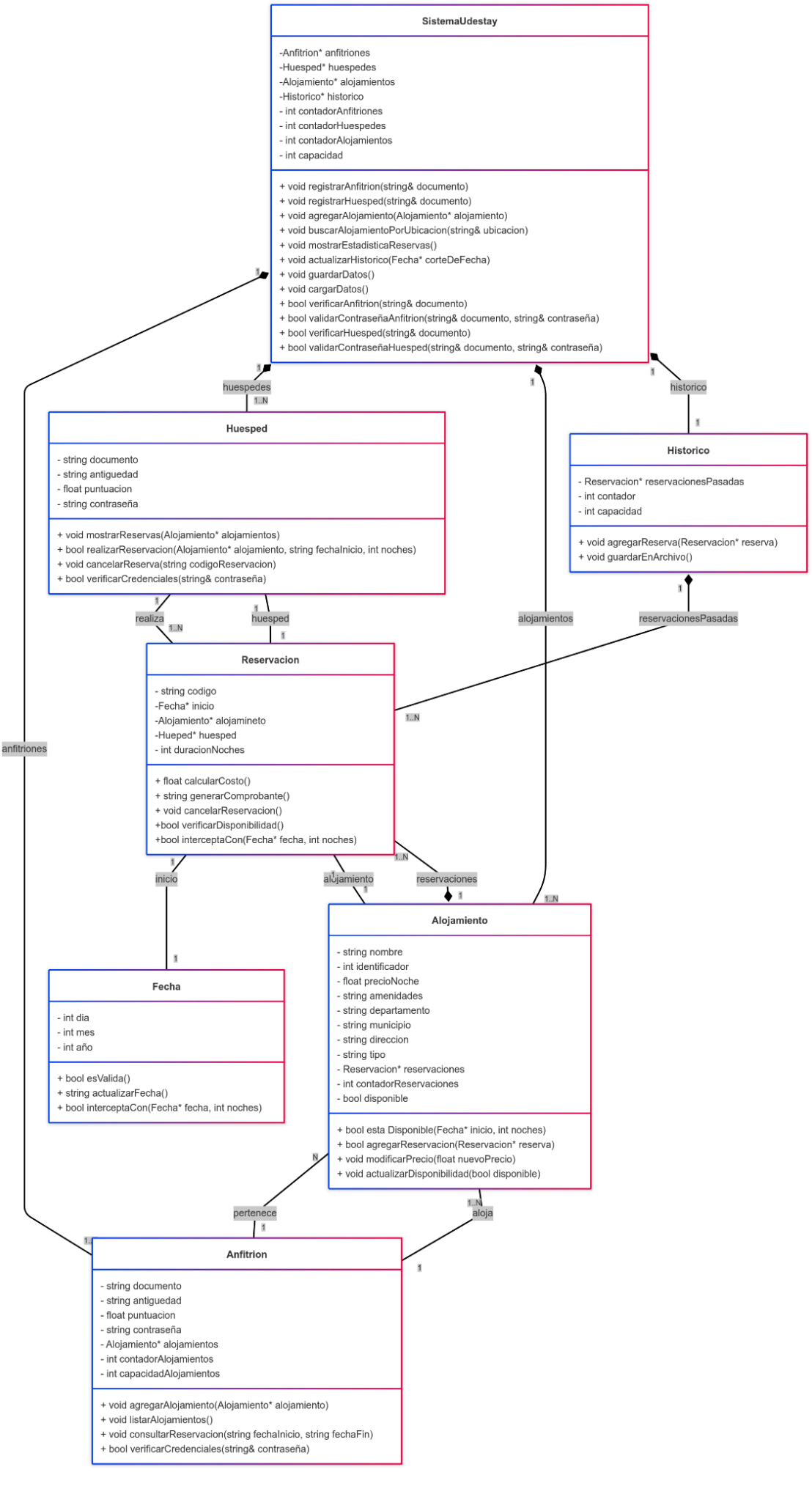
- **Consideración clave:** Sin bibliotecas de serialización avanzadas o bases de datos, la solución adoptada fue el **almacenamiento en archivos de texto plano (`.txt`)**. Cada objeto se serializa manualmente a una línea o bloque de texto y se deserializa al cargar, lo que requiere un formato bien definido y una lógica robusta para el parseo.

4. **Medición de Recursos:**

    - El desafío exige medir el consumo de recursos (iteraciones y memoria).
    - **Consideración clave:** Se deben implementar contadores de operaciones y funciones que estimen el uso de memoria sumando el tamaño de los objetos dinámicamente asignados y sus atributos. Esto implica una instrumentación cuidadosa del código.

5. **Manejo de Cadenas de Texto (`std::string`):**

    - A diferencia de otros contenedores, se permite el uso de `std::string`.
    - **Consideración clave:** `std::string` se utilizaría para la mayoría de los atributos de texto (nombres, documentos, contraseñas, etc.) debido a su seguridad y facilidad de uso en comparación con `char[]`, lo que simplifica la gestión de memoria para cadenas de longitud variable. La única excepción obligatoria es `char sugerenciasHuesped[1000]` en `Alojamiento`.


SistemaUdestay
-Anfitrion* anfitriones
-Huesped* huespedes
-Alojamiento* alojamientos
-Historico* historico
- int contadorAnfitriones
- int contadorHuespedes
- int contadorAlojamientos
- int capacidad
+ void registrarAnfitrion(string& documento)
+ void registrarHuesped(string& documento)
+ void agregarAlojamiento(Alojamiento* alojamiento)
+ void buscarAlojamientoPorUbicacion(string& ubicacion)
+ void mostrarEstadisticaReservas()
+ void actualizarHistorico(Fecha* corteDeFecha)
+ void guardarDatos()
+ void cargarDatos()
+ bool verificarAnfitrion(string& documento)
+ bool validarContraseñaAnfitrion(string& documento, string& contraseña)
+ bool verificarHuesped(string& documento)
+ bool validarContraseñaHuesped(string& documento, string& contraseña)
huespedes
historico
Huesped
- string documento
- string antiguedad
- float puntuacion
- string contraseña
+ void mostrarReservas(Alojamiento* alojamientos)
+ bool realizarReservacion(Alojamiento* alojamiento, string fechaInicio, int noches)
+ void cancelarReserva(string codigoReservacion)
+ bool verificarCredenciales(string& contraseña)
Historico
- Reservacion* reservacionesPasadas
- int contador
- int capacidad
+ void agregarReserva(Reservacion* reserva)
+ void guardarEnArchivo()
realiza
huesped
alojamientos
reservacionesPasadas
Reservacion
- string codigo
-Fecha* inicio
-Alojamiento* alojamineto
-Hueped* huesped
- int duracionNoches
+ float calcularCosto()
+ string generarComprobante()
+ void cancelarReservacion()
+bool verificarDisponibilidad()
+bool interceptaCon(Fecha* fecha, int noches)
anfitriones
inicio
alojamiento
reservaciones
Fecha
- int dia
- int mes
- int año
+ bool esValida()
+ string actualizarFecha()
+ bool interceptaCon(Fecha* fecha, int noches)
Alojamiento
- string nombre
- int identificador
- float precioNoche
- string amenidades
- string departamento
- string municipio
- string direccion
- string tipo
- Reservacion* reservaciones
- int contadorReservaciones
- bool disponible
+ bool esta Disponible(Fecha* inicio, int noches)
+ bool agregarReservacion(Reservacion* reserva)
+ void modificarPrecio(float nuevoPrecio)
+ void actualizarDisponibilidad(bool disponible)
pertenece
aloja
Anfitrion
- string documento
- string antiguedad
- float puntuacion
- string contraseña
- Alojamiento* alojamientos
- int contadorAlojamientos
- int capacidadAlojamientos
+ void agregarAlojamiento(Alojamiento* alojamiento)
+ void listarAlojamientos()
+ void consultarReservacion(string fechaInicio, string fechaFin)
+ bool verificarCredenciales(string& contraseña)

**c. Descripción en alto nivel la lógica de las tareas que usted definió para aquellos subprogramas cuya solución no sea trivial.**

A continuación, se describen las lógicas de algunos subprogramas clave, destacando su complejidad y cómo abordan los requisitos del problema:

1. **`SistemaUdestay::cargarDatos()` y `SistemaUdestay::guardarDatos()`:**
    - **Lógica:** Estos métodos son responsables de la persistencia de todo el estado del sistema.
    - **`guardarDatos()`:** Itera sobre todos los arreglos dinámicos de `Anfitrion`es, `Huesped`es, `Alojamiento`s y `Reservacion`es. Para cada objeto, sus atributos se escriben en archivos de texto plano específicos (ej., `anfitriones.txt`, `huespedes.txt`). La lógica debe manejar la serialización de `std::string`, enteros, flotantes, y booleanos. Para relaciones (`Alojamiento` dentro de `Anfitrion`, `Reservacion` dentro de `Alojamiento`), se requiere una lógica anidada para asegurar que los datos relacionados se escriban correctamente y puedan ser reconstruidos.
    - **`cargarDatos()`:** Es el inverso. Lee los archivos línea por línea, parsea los datos para extraer los atributos de cada objeto. Para los atributos de texto (como `std::string`), esto implica leer hasta un delimitador específico. Los enteros y flotantes se convierten de cadena a su tipo numérico. Para recrear las relaciones, los objetos deben ser creados dinámicamente y sus punteros asignados correctamente. Por ejemplo, al cargar una `Reservacion`, se deben encontrar los punteros correspondientes al `Alojamiento` y `Huesped` previamente cargados utilizando sus identificadores. Esto implica búsquedas lineales en los arreglos cargados, lo que contribuye a las iteraciones medidas.
    - **Complejidad:** La principal complejidad radica en la serialización/deserialización manual de estructuras de datos anidadas

y el manejo de punteros y arreglos dinámicos sin el soporte de librerías de serialización o contenedores STL.

2. **`Alojamiento::estaDisponible(Fecha* inicio, int noches)`:**
    - **Lógica:** Determina si un alojamiento está disponible para un rango de fechas dado, considerando sus reservas existentes.
    - **Funcionamiento:** Recorre el arreglo dinámico `reservaciones` interno del `Alojamiento`. Para cada reserva existente, llama al método `Reservacion::interceptaCon(inicio, noches)`. Si alguna reserva existente se superpone con el rango de fechas solicitado, el alojamiento no está disponible y el método retorna `false`. Si ninguna reserva se superpone, retorna `true`.
    - **Complejidad:** Depende de la eficiencia de `Reservacion::interceptaCon` y del número de reservas existentes. En el peor caso, es una búsqueda lineal a través de todas las reservas del alojamiento.
3. **`Reservacion::interceptaCon(Fecha* fechaSolicitada, int noches)`:**
    - **Lógica:** Verifica si la reserva actual se solapa con un rango de fechas propuesto (`fechaSolicitada` y `noches`).
    - **Funcionamiento:** Convierte tanto la fecha de inicio de la reserva actual como la fecha solicitada a un formato numérico que permita la comparación (ej., días desde una fecha base, o simplemente un número secuencial de días dentro del año/mes). Luego, aplica la lógica de solapamiento de intervalos:
        1. Un intervalo A (inicioA, finA) y un intervalo B (inicioB, finB) se solapan si `(inicioA <= finB) && (inicioB <= finA)`.
        2. `finA` se calcula como `inicio->convertirADias() + duracionNoches - 1`.
        3. `finB` se calcula como `fechaSolicitada->convertirADias() + noches - 1`.

- **Complejidad:** Requiere un método robusto en `Fecha` (`convertirADias` o similar) para transformar las fechas en valores comparables y la lógica de intervalos para determinar el solapamiento.

4. **`SistemaUdestay::crearReservacion(...)`:**
    - **Lógica:** Orquesta el proceso de creación de una nueva reserva.
    - **Funcionamiento:**
        1. Busca el `Huesped` por su documento y el `Alojamiento` por su ID en los arreglos dinámicos de `SistemaUdestay`. Esto implica búsquedas lineales.
        2. Valida la fecha de inicio de la reserva utilizando `Fecha::esValida()`.
        3. Llama a `Alojamiento::estaDisponible()` para verificar si el alojamiento está libre en las fechas solicitadas.
        4. Si todo es válido y disponible, genera un código de reserva único.
        5. Crea un nuevo objeto `Reservacion` dinámicamente (`new Reservacion(...)`), pasando punteros al `Alojamiento` y `Huesped` encontrados.
        6. Agrega esta nueva reserva tanto al arreglo de `reservaciones` de `SistemaUdestay` como al arreglo interno de `reservaciones` del `Alojamiento` correspondiente.
        7. Registra las operaciones con `incrementarIteraciones()` y `finalizarMedicion()`.
    - **Complejidad:** Coordina múltiples búsquedas y validaciones, manejando la creación y vinculación de objetos dinámicos.

**d. Algoritmos implementados debidamente intra-documentados.**

Dada la restricción de no usar contenedores STL, gran parte de la lógica algorítmica se centra en operaciones sobre arreglos dinámicos:

1. **Búsqueda Lineal en Arreglos Dinámicos:**

    - **Ubicación:** Presente en `SistemaUdestay` para buscar anfitriones, huéspedes, alojamientos y reservas por sus identificadores o documentos.
    - **Descripción:** Un bucle `for` simple que itera desde el índice 0 hasta `contador - 1`. En cada iteración, compara el atributo buscado del objeto actual con el valor objetivo.

**e. Problemas de desarrollo que afrontó.**

Durante el desarrollo del proyecto, se enfrentaron varios desafíos significativos, principalmente debido a las estrictas restricciones impuestas por el desafío:

1. **Gestión Manual de Memoria con Arreglos Dinámicos:**

    - **Problema:** La prohibición de contenedores STL como `std::vector` forzó el uso de `new[]` y `delete[]` para todas las colecciones. Esto llevó a constantes desafíos con:
        - **Fugas de memoria:** Olvidar `delete[]` en destructores o al redimensionar arreglos.
        - **Doble liberación de memoria:** Intentar liberar la misma memoria dos veces.
        - **Punteros colgantes/salvajes:** Punteros que apuntan a memoria ya liberada, causando fallos de segmentación.
        - **Redimensionamiento ineficiente:** Copiar grandes arreglos dinámicos cada vez que se supera la capacidad es costoso en tiempo.
    - **Solución:** Desarrollo de una disciplina estricta de gestión de recursos (RAII en el contexto de destructores), implementación cuidadosa de constructores de copia y operadores de asignación (aunque no todos los escenarios complejos de copia profunda se exploraron exhaustivamente si no eran estrictamente necesarios para el desafío). La práctica y la depuración exhaustiva fueron esenciales.

2. **Serialización y Deserialización de Objetos Anidados:**
    - **Problema:** Guardar y cargar relaciones entre objetos (ej., `Alojamiento` en `Anfitrion`, `Reservacion` en `Alojamiento`, y los punteros en `Reservacion`) en archivos de texto plano sin bibliotecas dedicadas fue complejo. La reconstrucción de las relaciones (especialmente los punteros) al cargar los datos requirió búsquedas en las colecciones ya cargadas.
    - **Solución:** Diseñar un formato de archivo claro y delimitado. Implementar lógica secuencial: cargar primero las entidades "padre" (`Anfitrion`, `Huesped`), luego las "hijas" (`Alojamiento`, `Reservacion`), y finalmente establecer los punteros (`Reservacion` a `Alojamiento` y `Huesped`) mediante la búsqueda por identificadores únicos.
3. **Lógica de Fechas y Solapamientos:**
    - **Problema:** La implementación de la clase `Fecha` y los métodos `esValida()` e `interceptaCon()` para verificar la disponibilidad de alojamientos fue un punto crítico. Asegurar que todos los casos de solapamiento se cubrieran correctamente (inicio, fin, inclusión total, etc.) y que la lógica de años bisiestos fuera precisa.
    - **Solución:** Desarrollo incremental y pruebas unitarias (mentales o manuales) para `Fecha`. La conversión de fechas a un número entero de "días desde el año 0" simplificó la lógica de comparación de intervalos.
4. **Gestión de `char[]` para `sugerenciasHuesped`:**
    - **Problema:** La combinación de `std::string` para casi todas las cadenas y `char[]` para un atributo específico (`sugerenciasHuesped`) introdujo una pequeña inconsistencia y la necesidad de recordar usar funciones de `cstring` (ej., `strcpy_s` o `strncpy`) en lugar de operadores de `std::string`.

- **Solución:** Asegurar el uso correcto de `strcpy` (o variantes seguras) y verificar límites de tamaño para evitar desbordamientos de búfer en ese atributo específico.

### f. Evolución de la solución y consideraciones para tener en cuenta en la implementación.

La solución evolucionó a través de varias fases, dictadas por los requisitos y las lecciones aprendidas:

1. **Fase Inicial (Modelado y Clases Básicas):**
    - **Consideraciones:** Se definieron las clases principales (`Anfitrion`, `Huesped`, `Alojamiento`, `Reservacion`, `Fecha`) y sus atributos. El enfoque inicial fue entender las responsabilidades de cada clase y las relaciones estáticas (qué contiene a qué).
    - **Evolución:** Se creó el diagrama de clases como guía. La primera implementación fue de los constructores, destructores y getters/setters.
2. **Fase de Gestión de Colecciones y Memoria:**
    - **Consideraciones:** Implementación de arreglos dinámicos para `SistemaUdestay` y dentro de `Anfitrion` y `Alojamiento`. Se dedicó un esfuerzo considerable a comprender y aplicar correctamente `new[]` y `delete[]`, así como la lógica de redimensionamiento. La doble indirección en `Historico` (`Reservacion**`) presentó un desafío adicional.
    - **Evolución:** Se refactorizaron los destructores para asegurar la liberación de memoria. Se implementaron los métodos `agregarX` y `eliminarX` que manejaban la capacidad y el `contador`. Se identificaron los puntos donde era necesario copiar objetos (para colecciones de objetos) vs. copiar punteros (para colecciones de punteros a objetos existentes).

3. **Fase de Lógica de Negocio y Fechas:**

    - **Consideraciones:** Desarrollo de la clase `Fecha` con su lógica de validación y comparación. Implementación de `estaDisponible` y `interceptaCon` como el corazón de la gestión de reservas. Creación de la lógica para `crearReservacion` que integra búsquedas, validaciones y actualizaciones.
    - **Evolución:** Las funciones de fecha se volvieron cruciales. La implementación iterativa de la lógica de reserva, probando casos límite de disponibilidad, ayudó a perfeccionar estos algoritmos.
4. **Fase de Persistencia y Medición de Recursos:**

    - **Consideraciones:** Implementación de `cargarDatos()` y `guardarDatos()`. Esto fue iterativo, primero guardando/cargando datos simples, luego anidados y finalmente manejando las relaciones de punteros. La implementación de los contadores de iteraciones y la función `calcularMemoriaObjetos()` requirió instrumentar el código en cada operación relevante.
    - **Evolución:** Se definieron formatos de archivo específicos (ej., una línea por objeto, atributos separados por coma) para facilitar el parseo. La función de medición de memoria se refinó para sumar los `sizeof()` de cada objeto y el tamaño de los arreglos dinámicos contenidos.

**Consideraciones Clave para la Implementación (Lecciones Aprendidas):**

- **Disciplina en `new`/`delete`:** Cada `new` debe tener un `delete` correspondiente. Los destructores de clases que poseen memoria dinámica (con `new` en su constructor) son vitales. Para arreglos de punteros, se debe liberar la memoria de cada objeto apuntado antes de liberar el arreglo de punteros.

- **Gestión de Capacidad:** Al redimensionar arreglos dinámicos, una estrategia de crecimiento (ej., duplicar la capacidad) es más eficiente que incrementar en uno, ya que reduce la frecuencia de operaciones de copia costosas.
- **Referencias vs. Copias:** Distinguir cuándo un método debe recibir un objeto por referencia (puntero o referencia lvalue/rvalue) para modificarlo, y cuándo debe recibir una copia. En este proyecto, los punteros a objetos (`Anfitrion*`, `Huesped*`, `Alojamiento*`) fueron esenciales para establecer relaciones y evitar copias innecesarias.
- **Formatos de Archivo:** Para la persistencia manual, un formato de archivo claro y consistente es primordial para la robustez de la carga y guardado de datos.
- **Intra-documentación:** Comentarios claros dentro del código, especialmente para la lógica de los algoritmos y la gestión de memoria, son cruciales para el mantenimiento y la depuración.
- **Medición:** Instrumentar el código para la medición desde fases tempranas ayuda a entender el impacto de las decisiones de diseño y algoritmo.