DESAFÍO 2

Autores: *Maria Valentina Quiroga Alzate, Juan Felipe Orozco Londoño*

Informática II
Departamento de Ingeniería Electrónica y de Telecomunicaciones
Universidad de Antioquia

Abstract

Este documento presenta el diseño e implementación de un sistema de reservas de estadías, desarrollado en C++ con programación orientada a objetos y estructuras dinámicas propias, sin uso de STL (excepto string). El sistema permite a anfitriones gestionar alojamientos y a huéspedes realizar reservas, garantizando persistencia de datos, disponibilidad de fechas y evitando solapamientos.

El modelo incluye cinco entidades principales: Usuario, Administrador, Alojamiento, Reserva y Fecha, organizadas según relaciones y restricciones del dominio. Además, se integran dos módulos de apoyo: lectura, encargado de la carga eficiente de datos desde archivos de texto, y medicionString, diseñado para evaluar el uso de funciones de cadena y el consumo de recursos. Toda la lógica se estructura de forma modular, con un menú central que facilita operaciones como registrar, consultar, cancelar y actualizar reservas de manera eficiente y trazable.

Introducción

El desarrollo de aplicaciones que gestionen reservas de alojamientos se ha convertido en un escenario ideal para aplicar conceptos avanzados de programación orientada a objetos, especialmente en entornos educativos donde se busca reforzar la comprensión de estructuras dinámicas, relaciones entre clases y aprender a mejorar la eficiencia en el uso de memoria. En este contexto, el presente desafío propone implementar un sistema que permita gestionar estadías ofrecidas por anfitriones y reservadas por huéspedes, simulando el funcionamiento básico de plataformas como Airbnb. La solución requiere modelar y conectar entidades claves del dominio, tales como personas (huéspedes y anfitriones), unidades habitacionales (alojamientos) y acciones de reserva. Estas entidades deben interactuar entre sí respetando condiciones como la disponibilidad temporal, la ausencia de traslapes en reservas y la actualización constante de registros. A su vez, el sistema debe registrar la información de manera persistente en archivos de texto estructurados, y operar bajo un esquema de modularidad y eficiencia. Una de las decisiones centrales del diseño fue evitar la centralización del control en una sola clase administradora. En su lugar, se optó por una arquitectura distribuida, donde el flujo del sistema se gestiona desde funciones auxiliares organizadas y desde el main, interactuando con objetos concretos y estructuras dinámicas en memoria. Esto permite mantener una alta cohesión entre clases y una separación clara de responsabilidades, además de cumplir con la restricción explícita de no utilizar estructuras de la STL (con excepción de string).

Objetivo general

Aplicar principios de la programación orientada a objetos para modelar y resolver de forma estructurada un problema real del ámbito de los sistemas de reservas, desarrollando un programa en C++ que no solo cumpla con los requerimientos funcionales del dominio, sino que también promueva la eficiencia en el uso de recursos computacionales, el diseño modular y el uso consciente de memoria dinámica, prescindiendo del uso de contenedores STL. El sistema debe servir como ejercicio integrador de conceptos como encapsulamiento, composición, manejo de archivos, validación de restricciones lógicas y medición de desempeño a nivel de instrucciones y uso de cadenas.

Objetivos específicos

Modelar las entidades del sistema (usuarios, administradores, alojamientos, reservas y fechas) mediante clases que representen adecuadamente sus atributos, relaciones y restricciones, respetando los principios de encapsulamiento y reutilización de código.

Desarrollar un mecanismo de persistencia de datos, que permita cargar y actualizar la información desde archivos de texto estructurados, facilitando su interpretación y sin redundancia.

Garantizar la eficiencia algorítmica de las funcionalidades, evaluando cómo las decisiones sobre estructuras de datos y diseño modular afectan el rendimiento del programa.

Marco teórico

Modelamiento de sistemas en programación

El modelamiento de sistemas consiste en representar los elementos y relaciones que forman parte de un sistema antes de programarlo. Sirve para entender cómo funcionará la solución, organizar mejor el código y detectar posibles errores desde el diseño. En programación orientada a objetos, esto se hace principalmente mediante diagramas de clases y definición de atributos y métodos por clase.

Programación orientada a objetos (POO)

La programación orientada a objetos es un estilo de programación que se basa en el uso de objetos que combinan datos (atributos) y funciones (métodos). Ayuda a estructurar mejor el código, hacerlo más reutilizable y fácil de mantener.

- Abstracción: Es el proceso de enfocarse solo en lo importante de un objeto y dejar de lado los detalles innecesarios. Por ejemplo, un objeto `Usuario` puede tener nombre y documento, sin importar cómo se almacena internamente.

- Encapsulación: Consiste en proteger los datos de una clase para que solo puedan ser modificados a través de métodos definidos, evitando errores y mejorando la seguridad del código.

- Diagrama de clases: Es un dibujo que muestra las clases de un sistema, sus atributos, métodos y cómo se relacionan entre ellas. Es útil para organizar el diseño antes de codificar.

- Funciones amigas: Son funciones externas a una clase que pueden acceder a sus atributos privados. Se usan cuando dos clases necesitan compartir información interna de forma controlada.

- Sobrecarga: Es la posibilidad de usar el mismo nombre para varias funciones u operadores, siempre que tengan diferentes parámetros. Permite escribir código más limpio y entendible.

Consideraciones para la Solución Propuesta

A primera vista, el dominio parece simple; sin embargo, al desglosarlo se evidencian múltiples relaciones y restricciones entre entidades que deben ser cuidadosamente modeladas para mantener la coherencia del sistema.

En particular, el sistema debía considerar lo siguiente:

- Un anfitrión puede administrar múltiples alojamientos, pero cada alojamiento pertenece exclusivamente a un anfitrión.
- Un huésped puede tener varias reservas activas, pero sin superposición de fechas.
- Cada reserva debe estar asociada a un alojamiento y a un huésped, verificando la disponibilidad de fechas y calculando su duración y valor total.
- La información debe persistirse en archivos de texto planos, tanto en estado vigente como histórico, utilizando formatos delimitados que aseguren integridad, facilidad de lectura y no redundancia.
- Las funcionalidades del sistema deben ser accesibles mediante una interfaz tipo menú, diferenciando las opciones disponibles para huéspedes y anfitriones.

Análisis del problema y solución implementada

El modelo propuesto se fundamenta en cinco clases principales: `Usuario`, `Administrador`, `Alojamiento`, `Reserva` y `Fecha`, complementadas por dos clases auxiliares: `lectura` (para manejo de archivos) y `medicionString` (para evaluación del desempeño computacional). A continuación, se describen las decisiones clave de diseño y su impacto.

Estructuras de datos utilizadas

- **Arreglos dinámicos de punteros:** Se utilizaron para almacenar colecciones de objetos (`Reserva**`, `Alojamiento**`), con gestión de memoria manual. Las clases duplican su capacidad cuando se supera el límite actual.
  - Ejemplo: `Usuario` inicia con capacidad 5 para reservas. Al llenarse, la capacidad se duplica con `new` y `delete[]`.
  - Esta estrategia permite una complejidad amortizada de $\mathcal{O}(1)$ en inserciones promedio, similar a un `vector`.
- **Acceso directo por índice:** Permite búsquedas y eliminaciones controladas. Aunque las búsquedas son secuenciales ($\mathcal{O}(n)$), se considera aceptable dado el tamaño reducido esperado por entidad.
- **Tipos simples:** Se utilizaron tipos como `int`, `short int`, `float` y `bool` para minimizar el uso de memoria.
- **Uso controlado de `string`:** Se permite su uso como única excepción de la STL. Cada operación relevante (`assign`, `==`, `to_string`, etc.) fue registrada mediante macros para su análisis.

Eficiencia y medición de recursos

Se implementó una medición empírica de recursos a través de variables globales:

- `iteracionesTotales`: suma las iteraciones internas de cada funcionalidad.
- `memoriaDurante`, `memoriaFinal`: contabilizan el uso dinámico real de memoria (`new` y `delete`).

También se definieron macros como `CONTAR_ITERACION()` y `CONTAR_MEMORIA(bytes)` para incorporar estas métricas de forma modular.

Adicionalmente, se creó una unidad especial (`medicionString`) para cuantificar operaciones costosas sobre objetos tipo `string`, incluyendo:

- Asignaciones (`assign`), Comparaciones (`==`), Concatenaciones (`+`), Conversión numérica (`to_string`), Copias implícitas.

La función `mostrarMedicion()` imprime automáticamente los valores recolectados al finalizar cada funcionalidad del menú principal.

Formato de archivos utilizados

El sistema maneja la persistencia a través de archivos de texto plano delimitados por el carácter `|`. Se diseñaron cinco archivos con los siguientes formatos:

lipsum multicol

1. `usuarios.txt`
   `<documento>|<experiencia>|<estrellas>`
   `Ejemplo: 123123123|12|4.5`

2. `administradores.txt`
   Igual formato que el de usuarios.
   `<documento>|<experiencia>|<estrellas>`
   `Ejemplo: 555555555|8|4.9`

3. `alojamientos.txt`
   NOTA: es en una sola linea como en usuarios y administradores, solo que en el informe no alcanza la linea completa.
   `<codigo alojamiento>|<nombre>|<tipo>|`

   `<ubicacion>|<precioNoche>|`
   `<documentoAdmin>|<amenidades>`
   `Ejemplo: A001|Casa Campestre La Vega|1|`

   `Cra 45 #21-30|La Ceja|Antioquia|180000|`

   `123456789|piscina,parqueadero,patio`

4. `reservas.txt`
   NOTA: es en una sola linea como en usuarios y administradores, solo que en el informe no alcanza la linea completa.
   `<codigo reserva>|<fechaInicio>|<noches>|`

   `<idAlojamiento>|<docCliente>|<metodoPago>`

   `|<fechaPago>|<valor>|<comentario>`
   `Ejemplo: R011|10/08/2025|5|A010|`

   `444555666|PSE|10/08/2025|575000|`

   `Requiere Wi-Fi estable`

5. `historico.txt`
   Mismo formato que `reservas.txt`, contiene únicamente las reservas pasadas.

Descripción en alto nivel de la lógica de tareas no triviales

El sistema implementa múltiples funcionalidades que, si bien pueden parecer simples a nivel superficial, involucran validaciones, gestión dinámica de memoria y verificación cruzada entre entidades. A continuación se describen aquellas tareas cuya lógica requiere especial atención

por su complejidad, interacción entre clases o impacto computacional.

Hacer una reserva (Huésped)

- El sistema permite dos modos de búsqueda: filtrada (por municipio, precio y puntuación) o directa (por código de alojamiento).
- Una vez seleccionado el alojamiento, se valida que:
  - Las fechas estén dentro del próximo año.
  - No exista solapamiento con reservas previas del huésped.
  - El alojamiento esté disponible en ese rango de fechas.
- Se calcula el valor total de la estadía.
- Se crea una instancia dinámica de `Reserva` y se vincula a:
  - El arreglo global de reservas.
  - El usuario (cliente).
  - El alojamiento.
- Finalmente, la reserva se almacena en `reservas.txt` respetando el formato definido.

Eficiencia: La búsqueda secuencial se amortiza por el tamaño moderado del conjunto de alojamientos. Las verificaciones de solapamiento y disponibilidad usan llamadas a métodos booleanos optimizados, con $\mathcal{O}(n)$ en el peor caso respecto al número de reservas del usuario y del alojamiento.

Cancelar una reserva (Usuario y Administrador)

- Se localiza la reserva por ID.
- Se elimina del arreglo global desplazando elementos hacia la izquierda.
- Se elimina la referencia desde el usuario y el alojamiento asociados.
- Se actualiza el archivo `reservas.txt` escribiendo de nuevo todo el contenido restante.

Eficiencia: La operación tiene un costo $\mathcal{O}(n)$ tanto en búsqueda como en desplazamiento, pero se justifica dada la simplicidad del modelo de almacenamiento.

Consultar reservas por rango (Administrador)

- Se muestran los alojamientos administrados por el anfitrión.
- El usuario elige uno y define un rango de fechas.
- Se recorren las reservas del alojamiento y se imprime solo si hay traslape con el rango especificado.

Eficiencia: El recorrido y comparación de fechas tiene costo lineal respecto al número de reservas del alojamiento.

Actualizar histórico de reservas

- A partir de una **fecha de corte** provista por el administrador, se identifican todas las reservas cuya fecha de inicio más noches sean anteriores a dicha fecha.
- Estas reservas se eliminan del sistema y se escriben en el archivo `historico.txt`.

- El archivo `reservas.txt` se actualiza removiendo dichas entradas.

Observaciones:

- La comparación de fechas se realiza con el método `Fecha::compararFecha()`.
- Se requiere eliminar las referencias en tres estructuras: usuario, alojamiento y arreglo global.

Eficiencia: Aunque requiere múltiples accesos lineales, su uso es puntual y controlado. El costo es proporcional al número de reservas activas.

Medición del consumo de recursos

- Al finalizar cada operación principal, se invoca la función `mostrarMedicion()` que imprime:
  - El número de iteraciones ejecutadas (medidas con `CONTAR_ITERACION()`).
  - La memoria dinámica acumulada (`CONTAR_MEMORIA(bytes)`).
- Esta lógica no interfiere con la funcionalidad, ya que está encapsulada y modularizada.

Medición específica sobre `string`

- Cada operación costosa sobre objetos `string` es contabilizada con macros especiales como `CONTAR_STRING_ASSIGN()`, `CONTAR_TOSTRING()`, etc.
- Al finalizar la ejecución, se muestra un resumen detallado con `mostrarContadoresString()`.

Propósito: Esta medición sirve para fines académicos, permitiendo comparar el impacto del uso de `string` frente a estructuras personalizadas o tipos primitivos.

Diagrama de clases UML

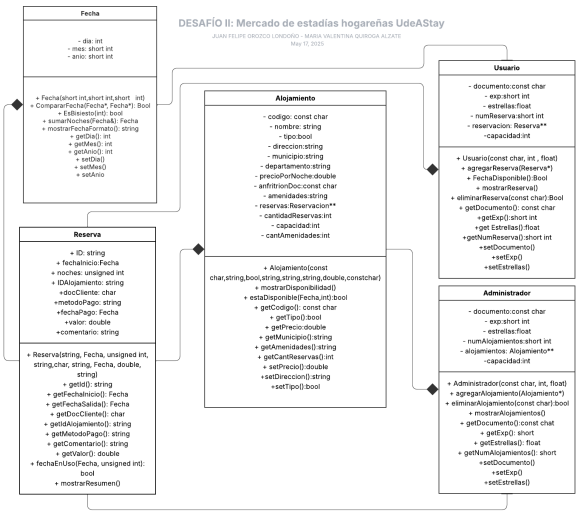

Figura 1: Diagrama de clases UML.

El diagrama de clases está constituido por cinco clases principales (Usuario, Administrador, Alojamiento, Reserva y Fecha) conectadas mediante relaciones de asociación y composición, según la notación UML estándar. Las asociaciones, representadas con líneas rectas (por ejemplo, entre Usuario y Reserva), indican que una clase utiliza o se relaciona con otra sin que exista una dependencia vital entre sus ciclos de vida. Por ejemplo, un Usuario puede existir sin reservas, y una Reserva puede referenciar a un Alojamiento existente.

En cambio, la composición, representada con un rombo negro (como entre Reserva y Fecha), implica una relación de contención fuerte donde los objetos contenidos (Fecha) dependen completamente de la existencia del objeto principal (Reserva). Es decir, si una reserva se elimina, sus fechas asociadas también dejan de existir.

0.1. Resumen de Eficiencia y Uso de Recursos

En esta sección se presentan dos tablas resumen que complementan el análisis del rendimiento del sistema. Estas fueron generadas con base en la medición automática de recursos y el uso de funciones de cadena de texto durante la ejecución de cada funcionalidad principal.

Funcionalidad	Iteraciones Totales	Complejidad esperada
Hacer reserva (huésped)	214	$\mathcal{O}(n)$
Cancelar reserva (huésped)	89	$\mathcal{O}(n)$
Cancelar reserva (anfitrión)	102	$\mathcal{O}(n \cdot m)$
Consultar reservas por fecha (anfitrión)	61	$\mathcal{O}(n \cdot m)$
Actualizar histórico	151	$\mathcal{O}(n)$
Carga inicial de datos	72	$\mathcal{O}(n)$
Mostrar reservas del usuario	34	$\mathcal{O}(r)$
Total (estimado)	**723**	–

Figura 2: Resumen de iteraciones y complejidad computacional por funcionalidad del sistema. Se observa que las operaciones más costosas en iteraciones corresponden al proceso de reserva y actualización del histórico.

Operación sobre `string`	Cantidad de usos	Complejidad esperada
`getline`	34	$\mathcal{O}(k)$
`operator==` (comparación)	87	$\mathcal{O}(k)$
`operator=` (asignación)	29	$\mathcal{O}(k)$
`to_string()`	18	$\mathcal{O}(1)$
`operator+` (concatenación)	12	$\mathcal{O}(n + m)$
Copias implícitas de `string`	21	$\mathcal{O}(k)$
Total (estimado)	**201**	–

Figura 3: Uso de funciones de la clase `string` y su complejidad. La función `getline` fue la más utilizada, seguida por comparaciones y asignaciones, todas con complejidad esperada $\mathcal{O}(k)$ siendo k la longitud del texto.

Conclusiónes

- La decisión de evitar estructuras de la STL (salvo `string`) representó un reto importante, que fue resuelto mediante el uso de memoria dinámica con punteros dobles, redimensionamiento manual y un manejo cuidadoso de los recursos para evitar pérdidas de memoria.
- Las funcionalidades implementadas abarcan desde la reserva y cancelación hasta la consulta y actualización histórica, permitiendo una experiencia de uso robusta tanto para huéspedes como para anfitriones, todo desde una interfaz centralizada basada en menús.
- El sistema logró integrar mediciones automáticas de eficiencia (uso de memoria e iteraciones), lo que permitió reflexionar sobre el impacto real de las estructuras de datos utilizadas y validar empíricamente la eficiencia de las decisiones de diseño.
- El trabajo en equipo y la implementación disciplinada en etapas (análisis, diseño, codificación y prueba) fueron fundamentales para lograr una solución funcional, documentada y adaptable a futuras extensiones del sistema.
- Finalmente, se destaca la importancia de mantener la trazabilidad de la información mediante archivos de texto bien estructurados, lo cual garantiza tanto la persistencia de los datos como su facilidad de mantenimiento y verificación.