

## **Informe Desafío 2**

Jorim de Jesus Saltarin Villamizar  
Duvian Alexander Flores Munera

Informática II  
Augusto Enrique Salazar Jiménez

Universidad de Antioquia

28 de mayo de 2025

## Análisis Desafío 2

### Análisis del problema

El objetivo de este desafío es desarrollar un sistema para la gestión de una plataforma de reservas de alojamientos UdeAStay, permitiendo la interacción entre dos tipos de usuarios: huéspedes y anfitriones. De igual manera la plataforma debe permitir funcionalidades específicas para cada rol, priorizando la integridad de los datos, disponibilidad de alojamientos y trazabilidad de reservaciones históricas.

### Desafíos identificados

- **Persistencia de datos:** Se requiere desarrollar un mecanismo para cargar, actualizar y almacenar los datos de manera permanente mediante archivos .txt.
- **Control de acceso:** Se debe permitir el ingreso de usuarios según su rol, verificando credenciales desde almacenamiento permanente (usuarios.txt).
- **Gestión de reservas:** Exclusiva para huéspedes. La plataforma debe soportar búsquedas filtradas por criterios específicos (municipio, cantidad de noches, costo, puntuación del anfitrión) y validar la disponibilidad antes de reservar.
- **Historial:** Las reservaciones anteriores a una fecha (día de hoy) deben trasladarse a un archivo histórico de manera automática, según la fecha que ingrese el usuario.
- **Rendimiento:** Medir el consumo de recursos del sistema, tanto en iteraciones realizadas como en memoria consumida.

### Alternativa de Solución

- **Archivos de texto:** Se utilizarán archivos de texto, organizados con un formato estructurado.
  - usuarios.txt
  - alojamientos.txt
  - reservas.txt
  - hsitorico.txt
- **Lógica del sistema:**
  - El flujo de inicio de sesión leerá los datos del archivo usuarios.txt y permitirá el acceso solo si las credenciales coinciden.
  - El menú principal diferenciará automáticamente las funciones habilitadas para cada rol
  - **Huéspedes:**
    - Buscar alojamientos
    - Reservas alojamientos
    - Anular reservas
  - **Anfitriones:**
    - Revisar reservas a sus alojamientos registrados
    - Anular reservas relacionadas con sus alojamiento

## Informe Desafío 2

De acuerdo al objetivo del desafío para UdeAStay, el trabajo se dividió en tres fases de desarrollo:

- Login
- Menú Huésped
- Menú Anfitrión

Las cuales fueron implementadas de acuerdo a una base de datos en formato .txt CSV, utilizando 3 archivos.txt de acuerdo al análisis.

- **usuarios.txt:** almacena la información de los diferentes usuarios, e, cada línea contiene los siguientes campos, separados por **comas**:
  - **nombre:** nombre de usuario de la persona.
  - **documento:** número de identificación.
  - **rol:** tipo de usuarios “H” para huésped y “A” para anfitrión.
  - **antigüedad:** tiempo que lleva en la plataforma en meses.
  - **puntuación:** valoración promedio del usuario.
- **alojamientos.txt:** contiene una lista de alojamientos, maneja la separación por **comas** para cada uno de sus propiedades:
  - **nombre:** nombre del alojamiento.
  - **código:** código identificador del alojamiento.
  - **anfitrión:** persona responsable del alojamiento.
  - **departamento:** departamento donde se encuentra alojado.
  - **municipio:** municipio específico de ubicación.
  - **tipo:** tipo de alojamiento “casa” o “apartamento”
  - **direccion:** dirección física.
  - **precio:** precio por noches en pesos colombianos
  - **amenidades:** lista de características separadas por punto.
  - **Disponibilidad:** rango de fechas en que se encuentra reservado por algún huésped, separadas por punto “fechaInicio.fechaFin”.
- **reservas.txt:** almacena la información de las reservas realizadas para los alojamientos, continua con el mismo estilo separado por **comas**:
  - **Fecha Entrada:** fecha de inicio en que comienza la estadía.
  - **Duración:** cantidad de noches.
  - **Código Reservación:** identificador único de la reserva.
  - **Código Alojamiento:** código de la propiedad reservada.
  - **nombre Huésped:** usuario quien realiza la reserva
  - **medio Pago:** método de pago seleccionado “PSE” O “Tarjeta de crédito”
  - **fecha de pago:** fecha de pago que el usuario propone.
  - **precio Noches:** precio por noche en el alojamiento
  - **anotaciones:** comentarios adicionales y opcionales, como motivo del viaje por parte del huésped

Para la implementación de este desafío se desarrollaron 4 clases importantes, que almacenan toda la información necesaria para realizar los diferentes procesos. Utilizando la Programación Orientada a Objetos (POO)

- **Clase Usuario:** La clase Usuario modela a una persona en el sistema, y puede tener dos roles
  - “H”: Huésped
  - “A” Anfitrión
- **Atributos principales:**
  - **char\* nombreUsuario:** Nombre identificador del usuario.
  - **char\* documento:** Documento de identidad (único).
  - **char tipoRol:** Tipo de usuario ( “H” o “A”).
  - **int meseAntiguedad:** Tiempo que lleva en la plataforma en meses.
  - **float puntuacion:** valoración promedio.
- **Funcionalidades clave:**
  - **Acceso a datos:** Métodos para obtener los valores de los atributos (getNombreUsuario(), getDocumento(), getRol(), getMesesAntiguedad(), getPuntuacion()).
  - **Identificación del rol:** Métodos esHuesped() y esAnfitrión() retornan un booleano indicando el tipo de usuario.
  - **Carga de datos desde archivo:** El método cargarArchivoUsuarios permite leer los usuarios de un archivo de texto plano (usuarios.txt) y almacenarlos dinámicamente en un arreglo de punteros.
- **Aplicación de principios de POO:**
  - **Encapsulamiento:**
    - Los atributos son declarados como *private* para proteger los datos internos.
    - Solo se accede a través de métodos *get*, lo que permite mantener el control sobre cómo se accede y se modifica la información.
  - **Abstracción:**
    - La clase representa cualquier tipo de usuario de la plataforma, ocultando detalles de cómo se carga la información desde archivo o como se almacena en memoria.
  - **Modularidad:**
    - La carga de usuarios está encapsulada en una función *static*, por lo que puede usarse sin necesidad de instanciar.

- **Memoria dinámica:**
  - Se utiliza new y delete[] para manejar cadena dinámicas char\*, haciendo uso del constructor y destructor para evitar pérdidas de memoria o desbordamientos.
- **Clase Alojamiento:** Representa un lugar que puede ser reservado dentro de la plataforma. Se asocia a un usuario de tipo Anfitrión, y almacena información clave para la búsqueda, disponibilidad de cada alojamiento almacenado en el archivo de texto.
  - **Atributos principales:**
    - **char\* nombreAlojamiento.**
    - **char\* codigoAlojamiento.**
    - **Usuario\* anfitrión:** Relación con la clase Usuario.
    - **char\* departamento, char\* municipio, char\* direccion.**
    - **char tipo:** casa o apartamento.
    - **float precioPorNoche.**
    - **char\*\* amenidades:** Arreglo dinámico de cadenas con características del alojamiento.
    - **int cantidadAmenidades.**
    - **char\*\* fechasReservadas:** Arreglo de fechas de inicio y fin de reservas.
    - **int totalFechas.**
  - **Funcionalidades clave:**
    - **Getters:** Para todos los atributos principales (nombre, código, dirección, etc.).
    - **Disponibilidad:** Método estaDisponible() que verifica si el alojamiento está libre entre dos fechas.
    - **Carga desde archivo:** Método static que lee alojamientos desde un archivo y los asocia a usuarios anfitriones existentes.
  - **Aplicación de principios de POO:**
    - **Encapsulamiento:**
      - Atributos private y acceso solo por medio de métodos get
    - **Abstracción:**
      - El usuario no se tiene que preocupar por cómo se carga la información de los alojamientos.
    - **Modularidad:**
      - La lógica de carga desde archivo está separada y centralizada en un método static.

- **Relación entre objetos:**
  - El atributo Usuario\* anfitrión muestra asociación con otra clase (Usuario).
- **Memoria dinámica:**
  - Se utiliza new y delete[] para manejar cadena dinámicas char\* y char\*\*, haciendo uso del constructor y destructor para evitar pérdidas de memoria o desbordamientos.
- **Clase Reservación:** La clase Reservación representa una reserva realizada por un huésped para un alojamiento específico. Su propósito es almacenar y gestionar toda la información asociada a una reservación, como fechas, método de pago, usuario que hace la reserva, alojamiento reservado y anotaciones opcionales.
  - **Atributos principal:**
    - **char\* fechaEntrada.**
    - **int duracion.**
    - **char\* codigoReservacion.**
    - **Alojamiento\* alojamiento:** Relación con la clase Alojamiento para el código
    - **Usuario\* documentoHuesped:** Relación con la clase Usuario (usuarios que hizo la reserva)
    - **char\* metodoDePago.**
    - **char\* fechadePago.**
    - **float monto:** Monto total a pagar, según la cantidad de noches.
    - **char\* anotaciones.**
  - **Funcionalidades clave:**
    - **Getters:** Para el llamado de los atributos principales
    - **Carga desde archivo:** Método static que lee reservas desde un archivo y los asocia a Usuario y Alojamiento, para realizar la reservación correctamente.
  - **Aplicación de principios de POO:**
    - **Encapsulamiento:**
      - Atributos private y acceso solo por medio de métodos get
    - **Abstracción:**
      - El usuario no se tiene que preocupar por cómo se carga la información de las reservaciones.
    - **Modularidad:**
      - La lógica de carga desde archivo está separada y centralizada en un método static.

- **Relación entre objetos:**
  - El atributo Alojamiento\* alojamiento y Usuario\* documentoHuesped muestra asociación con otra clase (Usuario y Alojamiento).
- **Memoria dinámica:**
  - Se utiliza new y delete[] para manejar cadena dinámicas char\* y char\*\*, haciendo uso del constructor y destructor para evitar pérdidas de memoria o desbordamientos. No libera la memoria de los punteros Alojamiento\* ni Usuario\* ya que no le pertenecen a esta clase.
- **Clase Sesión:** La clase Session representa la gestión básica de la sesión de un usuario en el sistema UdeAStay. Su propósito principal es mantener el usuario activo, controlar el inicio y cierre de sesión, y permitir obtener información del usuario activo.
  - **Atributos principales:**
    - **Usuario\* usuarioActivo:** Puntero a un objeto Usuario, el cual representa el usuario que ha iniciado sesión actualmente.
  - **Funcionalidad clave:**
    - **iniciarSesion:** funcionalidad de tipo *bool* que busca un arreglo de usuarios que coincida con el nombre y documento proporcionados en el inicio de sesión del main, Si lo encuentra, asigna ese usuario a usuarioActivo y retorno *true*, de lo contrario retorna *false*.
    - **cerrarSesion:** finaliza la sesión estableciendo usuarioActivo a *nullptr*.
    - **getUsuarioActivo():** devuelve un puntero al usuario activo actual, o *nullptr* si no hay sesión activa.
  - **Aplicación de principios de POO:**
    - **Sesion():** constructor que inicializa el puntero usuarioActivo en *nullptr*.
    - **~Sesion():** destructor que simplemente pone el puntero usuarioActivo en *nullptr*.

## **ControlReservaciones.cpp**

Este archivo.cpp implementa todo el módulo de reservas de alojamientos que permite a los usuarios buscar alojamientos por filtros o por código, reservar y anular reservaciones. Se maneja fechas, disponibilidad de alojamientos y persistencia de datos en archivos de texto para evitar hacer reservaciones en días que no está disponible.

### **Tareas principales**

#### **1. Gestion de Fechas y calendario:**

- a. `esBisiesto(int anio)`
- b. `diasDelMes(int mes, int anio)`
- c. `sumarDias(const char* fecha, int dias, char* resultado)`
- d. `diaDeLaSemana(int dia, int mes, int anio)`

#### **2. Validación de disponibilidad:**

- a. **AlojamientoReservadoEnFechas(...):** Verifica si un alojamiento ya está reservado en un rango de fechas.
- b. **UsuarioConReservaEnFechas(...):** Evita que un usuario haga reservaciones en fechas donde ya tiene una reserva.
- c. **agregarRangoFechaAlojamiento(...):** Actualiza el archivo de alojamientos agregando rangos de fechas ocupadas luego de confirmar cada reservación.
- d. **eliminarRangoFechaAlojamiento(...):** Similar al anterior, en este caso remueve rangos de fechas luego de confirmar la anulación de una reserva.

#### **3. Generación de código de reservación:**

- a. **CodigoReservacion(char\* codigoGenerado):** Genera códigos únicos incrementales para cada reservación creada con formato "RSV#####".

#### **4. agregarReservacion:**

- a. **reservarAlojamiento(...):** Se divide en 4 etapas:
  - i. **Búsqueda:**
    1. **Opción 1 (Filtros):** Permite búsqueda por municipio, cantidad de noches, fechas, rango de precios y puntuación mínima.
    2. **Opción 2 (Código directo):** Búsqueda directa por código de alojamiento
  - ii. **Validaciones:**
    1. Verifica disponibilidad del alojamiento en las fechas
    2. Verifica que el usuario no tenga reservas conflictivas
    3. Valida que el alojamiento existe
  - iii. **Datos de reservación:**
    1. Solicita información de pago (PSE o Tarjeta)



2. Calcula monto total (precio por noche  $\times$  número de noches)
3. Recolecta anotaciones adicionales (opcional por el usuario)

**iv. Confirmación:**

1. Muestra comprobante detallado con fechas formateadas *nombreDía, día "de" nombreMes "del" año.*
2. Solicita confirmación final del usuario
  - a. Si se confirma:
    - i. Actualiza archivo de alojamientos con fechas ocupadas
    - ii. Guarda reserva en archivo "reservas.txt"
    - iii. Actualiza el array de reservas en memoria.

**5. anularReservacion:**

**a. anularReservas(...):** Se divide en 3 etapas:

**i. Listado de reservas:**

1. Muestra todas las reservas del usuario con formato detallado
2. Calcula fechas de salida sumando duración a fecha de entrada.

**ii. Selecccion:**

1. El usuario selecciona reserva por código
2. Valida que la reserva pertenezca al usuario

**iii. Confirmación:**

**1. Solicita confirmación:**

**a. Si se confirma:**

- i. Libera fechas en archivo de alojamientos
- ii. Elimina reserva del array en memoria
- iii. Actualiza archivo "reservas.txt" eliminando la línea correspondiente

Teniendo en cuenta el desarrollo de las clases mencionadas anteriormente, se dividió el desarrollo en tres etapas, las cuales se ven reflejadas en el main. El archivo main.cpp es el punto de entrada del sistema UdeAStay, un sistema de alojamientos basado en usuarios y reservaciones. La lógica está estructurada en torno a la carga de datos desde archivos, la gestión de sesiones de usuarios y la ejecución de un menú interactivo según el rol del usuario (huésped o anfitrión)

## Fases principales

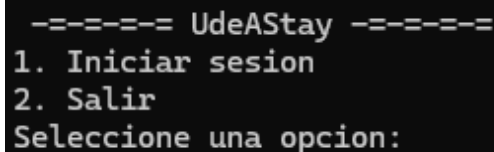
### 1. Carga de datos desde cada archivo

- Usuario::**cargarArchivoUsuarios(usuarios, totalUsuarios);
- Alojamiento::**cargarArchivoAlojamientos(alojamientos, totalAlojamientos, usuarios, totalUsuarios);
- Reservacion::**cargarReservas(reservas, totalReservas, usuarios, totalUsuarios, alojamientos, totalAlojamientos);

Estas funciones inicializan las estructuras principales en memoria dinámica para operar durante la ejecución.

### 2. Inicio de sesión:

- Se muestra un menú con dos opciones: iniciar sesión o salir. Esta es la primera cara a la hora de ejecutar.
- Si el usuario decide iniciar sesión, se solicitan el **nombre de usuario** y el **documento**.
- El sistema busca al usuario en la lista cargada. Si se encuentra, la sesión se inicia usando la clase Sesion.



```
--==--== UdeAStay --==--==
1. Iniciar sesion
2. Salir
Seleccione una opcion:
```

### 3. Menús según el rol del usuario: Luego de iniciar sesión correctamente con las credenciales, los menús se distinguen entre:

- Huésped “H”:** el cual se le presenta un menú que le permite:
  - Reservar un alojamiento.
  - Anular una reserva existente.
  - Cerrar sesión.
- Anfitrión “A”:** el cual tiene acceso a un menú que le permite:
  - Consultar reservaciones activas para todos sus alojamientos.
  - Anular una reservación.
  - Actualizar histórico.
  - Cerrar sesión.

### 4. Liberación de memoria: Antes de finalizar el programa, se libera toda la memoria dinámica asignada.

### Consumo de memoria

Al seleccionar la opción dos del menú principal “salir”, salen los dos mensajes que sirven para monitorear el rendimiento del sistema. Se implementó un módulo *mediciónmemoria.h*, el cual permite medir dos aspectos importantes:

- a. **Número de iteraciones realizadas:** para analizar la eficiencia de los algoritmos
- b. **Uso de memoria RAM en tiempo de ejecución:** se implementó con ayuda de funciones de la API de Windows *GetProcessMemoryInfo*.

Así mismo la función *mostrarConsumoDeRecursos()* y usando una variable global *iteraciones*, se incrementa linealmente en los puntos donde se requiere medir carga repetitiva, como ciclos for o while.

### Problemas de Desarrollo

Durante el desarrollo del desafío, se presentaron diversos problemas relacionados con la gestión de fechas y reservas. Entre ellos, afrontar de manera correcta la necesidad de manejar correctamente la suma y la comparación de fechas para evitar solapamientos en las reservas, así como validar la disponibilidad del alojamiento en rangos de fechas específicas. Y a la hora de obtener la fecha en “texto” y que fuera coherente con el calendario. También se identificó la importancia de verificar que un usuario no pueda realizar múltiples reservas en fechas que se crucen, todo esto, garantizando la integridad del sistema.

Para la parte del consumo de recursos, se integró una función que mide las interacciones y el uso de memoria. donde fue necesario corregir el uso de variables globales para el conteo de iteraciones y asegurar la medición adecuada del consumo de memoria, en este caso para Windows.

Además, durante el transcurso del desarrollo fue necesario replantear algunas cosas en cuanto al diagrama UML, lo que facilitó un poco más la comprensión de las clases y sus atributos, para el correcto funcionamiento y desempeño a la hora de reservar alojamientos.

