Informe Desafío 2

Andrés Felipe Sepúlveda R.
71.363.392

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

El desafío 2 consiste en implementar una plataforma para reservas de alojamiento llamada UdeAStay, dicha implementación debe estar soportada en el paradigma de la POO expuesta en las clases del curso. Dentro de los aspectos clave para tener en cuenta están las características de los alojamientos, las cualidades de huéspedes y anfitriones y las condiciones para realizar las reservas.

- Alojamiento: debe poseer un nombre, un código, un anfitrión responsable, un departamento, un municipio, un tipo, una dirección, un precio por noche y unas amenidades. Dentro de las condiciones se debe conocer las fechas futuras de reserva.

- Reservación: fecha de entrada, duración, código, documento del huésped, método de pago, fecha de pago y monto.

- Anfitriones: documento, antigüedad, puntuación, alojamientos que administra.

- Huéspedes: documento, antigüedad, puntuación, reservas que posee.

Para presentar una solución posible a este desafío, se deben tener en cuenta la relación existente entre los cuatro grupos de datos presentados.

✓ Huésped – reservación: un huésped puede tener múltiples reservaciones y las reservaciones guardan el documento del huésped.

✓ Alojamiento – reservación: un alojamiento puede tener múltiples reservaciones y cada reservación contiene un código de alojamiento.

✓ Anfitrión – alojamiento: un anfitrión administra uno o varios alojamientos y cada alojamiento tiene un documento de anfitrión.

Para implementar la solución se van a definir 4 clases, las cuales coinciden con los grupos de datos mencionados anteriormente: Alojamiento, Reservación, Huésped y Anfitrión. Por otro lado, anexa a la clase Alejamiento, se define una subclase denominada Amenidades (con 5 definidas), la cual se manejará como una clase anidada dentro de la clase Alojamiento, sin embargo, para efectos prácticos es simplemente un atributo de la clase Alojamiento.

Para el manejo de los archivos se dispondrán en el formato .txt con la información listada por líneas y separada por comas, con cada uno de los componentes funcionales requeridos:

- Huésped: documento, antigüedad, puntuación.
- Anfitrión: documento, antigüedad, puntuación.
- Alojamiento: código, nombre, documento del anfitrión, Departamento, Municipio, tipo de alojamiento, dirección, precio, conjunto de amenidades (ascensor, piscina, aire acondicionado, caja fuerte, parqueadero), fechas reservadas.
- Reservas: código, código del alojamiento, documento huésped, duración de la estadía, método de pago, monto, anotaciones, estado (utilizado para marcar la reserva en caso de ser anulada).

Para la estructura general del código, de acuerdo al análisis realizado, se podrán utilizar unas funciones principales y otras auxiliares de modo que la modularidad sea el pilar del desarrollo.

Las funciones auxiliares serían:

- Buscar huésped por documento.
- Buscar anfitrión por documento.
- Buscar alojamiento por código.
- Convertir fecha.
- Alojamiento ocupado en fecha.
- Código de reserva existente.

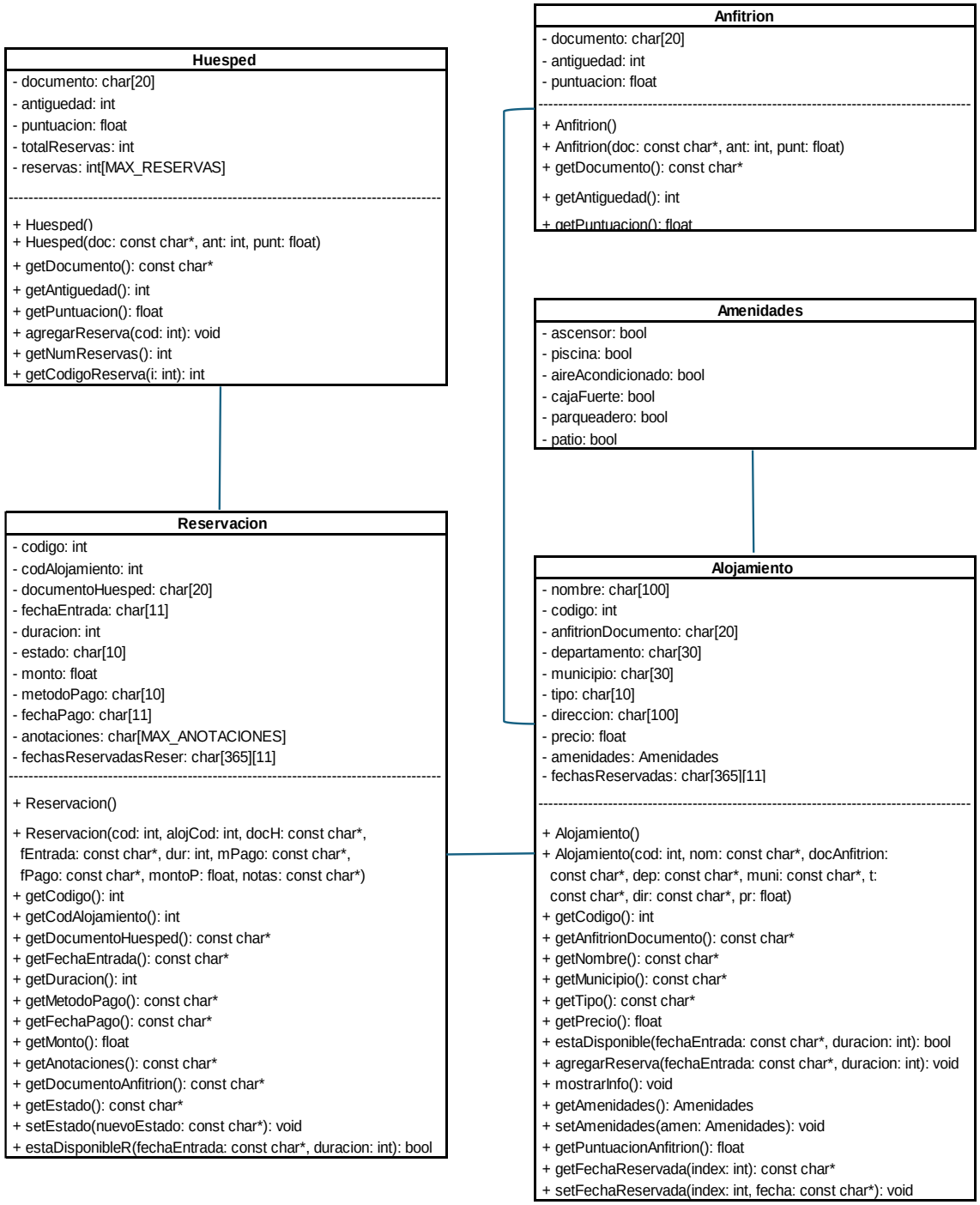
- Fecha es anterior.

Las funciones principales serían:

- Reservar alojamiento por código.
- Reservar alojamiento por filtro.
- Alojamiento ocupado en un rango.
- Reservas del anfitrión.
- Mostrar el consumo.
- Anular reservación.
- Actualizar histórico.
- Funciones para cargar y guardar los archivos.

Con la estructura y las funciones mencionadas se puede definir una solución al problema planteado cumpliendo con los requisitos mínimos de implementación, siguiendo el paradigma de la programación orientada a objetos, sin usar los contenedores de la STL y siguiendo las buenas prácticas de programación en aras de la eficiencia.

b. Diagrama de clases de la solución planteada. No debe ser un diagrama trivial que sólo incluya una o dos clases.



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

De manera inicial voy a plantear las funciones que se indican, esto estará sujeto a cambios a medida que avance con la solución del problema, teniendo en cuenta la complejidad que se vaya encontrando y las pruebas realizadas.

1. buscarHuespedPorDocumento / buscarAnfitrionPorDocumento / buscarAlojamientoPorCodigo

Propósito: Buscar un elemento (huésped, anfitrión o alojamiento) en su arreglo respectivo.

- Recorren un arreglo lineal (búsqueda secuencial).
- Comparan el valor buscado (documento o código).
- Devuelven un puntero al objeto encontrado o nullptr.

2. Reservacion::getDocumentoAnfitrion()

Propósito: Obtener el documento del anfitrión asociado a una reservación.

- A partir del código de alojamiento guardado en la reservación, busca el alojamiento en el arreglo global.
- Si lo encuentra, devuelve el documento del anfitrión asociado a ese alojamiento.

3. anularReservacion

Propósito: Permitir la anulación (eliminación) de una reserva si el usuario es el huésped o el anfitrión correspondiente.

- Recorre las reservaciones buscando la que coincide con el código dado.
- Verifica si el documento ingresado corresponde al huésped o al anfitrión del alojamiento reservado.
- Si tiene permisos, elimina la reservación desplazando los elementos restantes en el arreglo.

4. actualizarHistorico

Propósito: Mover al archivo histórico aquellas reservaciones cuya fecha de entrada ya pasó.

- Recorre el arreglo de reservaciones.
- Compara la fecha de entrada de cada reserva con la fecha actual (hoy).
- Si es anterior, escribe la información de esa reserva en el archivo historico_reservas.txt y la elimina del arreglo de reservaciones.
- Las reservas restantes se conservan en memoria.

5. mostrarConsumo

Propósito: Mostrar métricas sobre el uso de recursos del sistema.

- Imprime cuántas iteraciones se hicieron (útil para evaluar eficiencia).
- Calcula y muestra el uso de memoria basado en el tamaño de los objetos cargados.

6. alojamientoOcupadoEnRango(...)

Propósito: Determina si un alojamiento ya está reservado en un rango de fechas.

- Convierte la fecha de entrada deseada a time_t.
- Calcula el rango final sumando la duración.
- Itera sobre las reservas existentes del mismo alojamiento.
- Compara si hay solapamiento temporal con las reservas existentes.
- Devuelve true si hay cruce, false si está libre.

7. realizarReservaPorFiltro(…)

Propósito: Permite buscar alojamientos usando filtros como municipio, fecha, duración y precio máximo.

- Recibe los filtros desde el usuario.
- Llama a buscarAlojamientosPorFiltros para mostrar alojamientos que cumplen con esos criterios.
- Si el usuario elige uno, delega a reservarAlojamientoPorCodigo para completar el proceso de reserva.

8. reservarAlojamientoPorCodigo(…)

Propósito: Permite al huésped realizar una reserva directamente usando el código del alojamiento.

- Busca el alojamiento por su código.
- Verifica que esté disponible en la fecha deseada y para la duración especificada.
- Solicita los datos de la reserva (método de pago, fecha de pago, anotaciones).
- Calcula el monto total.
- Crea y almacena la nueva reserva, actualiza el alojamiento con la nueva fecha reservada.
- Asocia la reserva al huésped correspondiente.
- Guarda la nueva reserva en archivo.

d. Algoritmos implementados debidamente intra-documentados.

```
#include "funciones.h"
using namespace std;

// ---------------- Variables globales ----------------
// Arrays globales que almacenan los datos del sistema
Huesped huespedes[MAX_HUESPEDES];
Anfitrion anfitriones[MAX_ANFITRIONES];
Alojamiento alojamientos[MAX_ALOJAMIENTOS];
Reservacion reservaciones[MAX_RESERVAS];

// Variables para contar cuántos elementos hay en cada lista
int totalHuespedes = 0, totalAnfitriones = 0, totalAlojamientos = 0, totalReservas = 0;
int contadorIteraciones = 0;

// ---------------- Implementación de Clases ----------------

// --------- Clase Huesped ---------

// Constructor por defecto: inicializa datos en cero o vacío
Huesped::Huesped() : antiguedad(0), puntuacion(0.0), totalReservas(0) {
   documento[0] = '\0';
}

// Constructor con parámetros: inicializa los atributos del huésped
Huesped::Huesped(const char* doc, int ant, float punt) : antiguedad(ant),
puntuacion(punt), totalReservas(0) {
   strncpy(documento, doc, 20); // Copia segura del documento
}

// Métodos para acceder a los atributos del huésped
const char* Huesped::getDocumento() const { return documento; }
int Huesped::getAntiguedad() const { return antiguedad; }
float Huesped::getPuntuacion() const { return puntuacion; }

// Agrega una reserva al huésped
void Huesped::agregarReserva(int cod) {
   reservas[totalReservas++] = cod; // Guarda el código de la reserva
```

```
}

// Devuelve la cantidad de reservas hechas por el huésped
int Huesped::getNumReservas() const { return totalReservas; }

// Devuelve el código de reserva según el índice
int Huesped::getCodigoReserva(int i) const { return reservas[i]; }

// --------- Clase Anfitrion ---------

Anfitrion::Anfitrion() : antiguedad(0), puntuacion(0.0) {
    documento[0] = '\0';
}

Anfitrion::Anfitrion(const char* doc, int ant, float punt) : antiguedad(ant),
puntuacion(punt) {
    strncpy(documento, doc, 20); // Copia segura del documento
}

// Métodos para obtener atributos del anfitrión
float Anfitrion::getPuntuacion() const { return puntuacion; }
int Anfitrion::getAntiguedad() const { return antiguedad; }
const char* Anfitrion::getDocumento() const { return documento; }

// --------- Clase Alojamiento ---------

// Constructor por defecto: inicializa todos los atributos a cero o vacíos
Alojamiento::Alojamiento() : codigo(0), precio(0.0), amenidades({false, false, false,
false, false, false}) {
    nombre[0] = departamento[0] = municipio[0] = tipo[0] = direccion[0] = '\0';
}

// Constructor con parámetros: inicializa atributos con los valores recibidos
Alojamiento::Alojamiento(int cod, const char* nom, const char* docAnfitrion, const
char* dep, const char* muni, const char* t, const char* dir, float pr)
    : codigo(cod), precio(pr), amenidades({false, false, false, false, false, false}) {
    strncpy(nombre, nom, 100);
    strncpy(anfitrionDocumento, docAnfitrion, 20);
    strncpy(departamento, dep, 30);
```

```
    strncpy(municipio, muni, 30);
    strncpy(tipo, t, 10);
    strncpy(direccion, dir, 100);
}

// Getters para atributos clave del alojamiento
int Alojamiento::getCodigo() const { return codigo; }
const char* Alojamiento::getAnfitrionDocumento() const { return anfitrionDocumento;
}
const char* Alojamiento::getNombre() const { return nombre; }
const char* Alojamiento::getTipo() const { return tipo; }
const char* Alojamiento::getMunicipio() const { return municipio; }
float Alojamiento::getPrecio() const { return precio; }

// Verifica si el alojamiento está disponible para una fecha y duración dadas
bool Alojamiento::estaDisponible(const char* fechaEntrada, int duracion) const {
    tm entradaTm = {};
    if (!convertirFecha(fechaEntrada, entradaTm)) {
        return false;  // Fecha inválida
    }

    // Revisa cada una de las fechas reservadas para ver si hay solapamiento
    for (int i = 0; i < 365; ++i) {
        if (fechasReservadas[i][0] == '\0') continue;

        tm fechaReserva = {};
        if (convertirFecha(fechasReservadas[i], fechaReserva)) {
            // Calcula la fecha de fin de reserva y compara con entrada
            tm fechaFin = fechaReserva;
            fechaFin.tm_mday += duracion;

            if (entradaTm.tm_year == fechaReserva.tm_year && entradaTm.tm_mon ==
fechaReserva.tm_mon &&
                entradaTm.tm_mday >= fechaReserva.tm_mday && entradaTm.tm_mday <=
fechaFin.tm_mday) {
                return false; // Solapamiento encontrado
            }
        }
    }
```

```
    return true; // Si no hay conflicto, está disponible
}

// Agrega una nueva reserva al calendario del alojamiento
void Alojamiento::agregarReserva(const char* fechaEntrada, int duracion) {
    tm entradaTm = {};
    if (!convertirFecha(fechaEntrada, entradaTm)) {
        std::cerr << "Fecha inválida.\n";
        return;
    }

    // Guarda la nueva fecha de reserva
    for (int i = 0; i < 365; ++i) {
        if (fechasReservadas[i][0] == '\0') {
            snprintf(fechasReservadas[i], sizeof(fechasReservadas[i]), "%04d-%02d-%02d",
                    entradaTm.tm_year + 1900, entradaTm.tm_mon + 1, entradaTm.tm_mday);
            return;
        }
    }
    std::cerr << "No se pudo agregar la reserva, capacidad máxima alcanzada.\n";
}

// Muestra todos los detalles del alojamiento, incluidas las amenidades
void Alojamiento::mostrarInfo() const {
    cout << "\nNombre: " << nombre
         << "\nCodigo: " << codigo
         << "\nDepartamento: " << departamento
         << "\nMunicipio: " << municipio
         << "\nTipo: " << tipo
         << "\nDireccion: " << direccion
         << "\nPrecio por noche: $" << precio
         << "\nAmenidades:"
         << "\n  Ascensor: " << (amenidades.ascensor ? "Sí" : "No")
         << "\n  Piscina: " << (amenidades.piscina ? "Sí" : "No")
         << "\n  Aire acondicionado: " << (amenidades.aireAcondicionado ? "Sí" : "No")
         << "\n  Caja fuerte: " << (amenidades.cajaFuerte ? "Sí" : "No")
         << "\n  Parqueadero: " << (amenidades.parqueadero ? "Sí" : "No")
         << "\n  Patio: " << (amenidades.patio ? "Sí" : "No") << endl;
}
```

```
// Busca al anfitrión del alojamiento y retorna su puntuación
float Alojamiento::getPuntuacionAnfitrion() const {
    for (int i = 0; i < totalAnfitriones; ++i) {
        if (strcmp(anfitrionDocumento, anfitriones[i].getDocumento()) == 0) {
            return anfitriones[i].getPuntuacion();
        }
    }
    return 0.0f;  // Si no se encuentra
}

// --------- Clase Reservacion ---------

// Constructor por defecto
Reservacion::Reservacion() : codigo(0), codAlojamiento(0), duracion(0), monto(0.0) {
    documentoHuesped[0] = fechaEntrada[0] = metodoPago[0] = fechaPago[0] =
anotaciones[0] = '\0';
}

// Constructor con parámetros
Reservacion::Reservacion(int cod, int alojCod, const char* docH, const char* fEntrada,
int dur, const char* mPago, const char* fPago, float montoP, const char* notas)
    : codigo(cod), codAlojamiento(alojCod), duracion(dur), monto(montoP) {
    strncpy(documentoHuesped, docH, 20);
    strncpy(fechaEntrada, fEntrada, 11);
    strncpy(metodoPago, mPago, 10);
    strncpy(fechaPago, fPago, 11);
    strncpy(anotaciones, notas, MAX_AN



// ------------------- Implementación de las funciones auxiliares -------------------------

// Busca un huésped en el arreglo global por su documento de identidad
Huesped* buscarHuespedPorDocumento(const char* doc) {
    for (int i = 0; i < totalHuespedes; ++i) {
        contadorIteraciones++; // Contador global para análisis de eficiencia
        if (strcmp(huespedes[i].getDocumento(), doc) == 0)
            return &huespedes[i]; // Retorna puntero al huésped encontrado
    }
```

```
    return nullptr; // Si no se encuentra, retorna nulo
}

// Busca un anfitrión en el arreglo global por su documento de identidad
Anfitrion* buscarAnfitrionPorDocumento(const char* doc) {
    for (int i = 0; i < totalAnfitriones; ++i) {
        contadorIteraciones++;
        if (strcmp(anfitriones[i].getDocumento(), doc) == 0)
            return &anfitriones[i];
    }
    return nullptr;
}

// Busca un alojamiento en el arreglo global por su código
Alojamiento* buscarAlojamientoPorCodigo(int codigo) {
    for (int i = 0; i < totalAlojamientos; ++i) {
        contadorIteraciones++;
        if (alojamientos[i].getCodigo() == codigo)
            return &alojamientos[i];
    }
    return nullptr;
}

// Convierte una fecha en formato "YYYY-MM-DD" a estructura tm para manipulación
de fechas
bool convertirFecha(const char* fecha, tm& fechaTm) {
    // sscanf extrae año, mes y día de la cadena y los almacena en la estructura
    return sscanf(fecha, "%4d-%2d-%2d", &fechaTm.tm_year, &fechaTm.tm_mon,
&fechaTm.tm_mday) == 3;
}

// Verifica si un alojamiento está ocupado exactamente en una fecha específica
bool alojamientoOcupadoEnFecha(int codAlojamiento, const char* fecha) {
    for (int i = 0; i < totalReservas; ++i) {
        // Coincidencia tanto en código de alojamiento como en fecha exacta
        if (reservaciones[i].getCodAlojamiento() == codAlojamiento &&
            strcmp(reservaciones[i].getFechaEntrada(), fecha) == 0) {
            return true; // Está ocupado
        }
```

```
    }
    return false; // No se encontró reserva en esa fecha
}

// Verifica si un código de reserva ya existe en el sistema
bool codigoReservaExiste(int codigo) {
    for (int i = 0; i < totalReservas; ++i) {
        if (reservaciones[i].getCodigo() == codigo) {
            return true; // Ya existe una reserva con ese código
        }
    }
    return false; // El código es único
}

// Compara dos fechas en formato "YYYY-MM-DD" y devuelve true si f1 es anterior a
f2
bool fechaEsAnterior(const char* f1, const char* f2) {
    return strcmp(f1, f2) < 0; // strcmp compara lexicográficamente, lo cual funciona con
este formato
}


// ----------------------------- Implementación funciones principales ----------------------------

// Realiza una reserva directa si el código del alojamiento es conocido
void reservarAlojamientoPorCodigo(int codigo, const char* fechaEntrada, int duracion,
const char* documentoHuesped) {
    Alojamiento* aloj = buscarAlojamientoPorCodigo(codigo);

    // Validación de disponibilidad
    if (!aloj || !aloj->estaDisponible(fechaEntrada, duracion)) {
        std::cout << "Alojamiento no encontrado o no disponible.\n";
        return;
    }

    // Captura de información adicional para la reserva
    int codReserva;
    char metodoPago[10], fechaPago[11], anotaciones[MAX_ANOTACIONES];
```

```
    cout << endl << "Codigo unico para la reserva: ";
    cin >> codReserva;

    cout << endl << "Metodo de pago: ";
    cin >> metodoPago;

    cout << endl << "Fecha de pago (YYYY-MM-DD): ";
    cin >> fechaPago;

    cin.ignore();  // Limpia buffer antes de getline
    cout << endl << "Anotaciones: ";
    cin.getline(anotaciones, MAX_ANOTACIONES);

    float monto = aloj->getPrecio() * duracion;

    // Creación dinámica de la reserva
    Reservacion* nuevaReser = new Reservacion(
        codReserva, codigo, documentoHuesped, fechaEntrada, duracion,
        metodoPago, fechaPago, monto, anotaciones
    );

    reservaciones[totalReservas++] = *nuevaReser; // Copia la reserva en el arreglo
global
    aloj->agregarReserva(fechaEntrada, duracion);

    // Asociar la reserva al huésped
    Huesped* h = buscarHuespedPorDocumento(documentoHuesped);
    if (h) h->agregarReserva(codReserva);

    guardarReservasEnArchivo(); // Persistencia en archivo
    cout << endl << "Reserva realizada con exito. Monto total: $" << monto << "\n";

    delete nuevaReser; // Liberación de memoria
}

// Realiza una reserva paso a paso, con validaciones y captura de datos
void realizarReservaPorCodigo(const char* documentoHuesped) {
    int codigoAloj, duracion, codReserva;
```

```
    char fecha[11], metodoPago[10], fechaPago[11],
anotaciones[MAX_ANOTACIONES];

    cout << endl << "Ingrese el codigo del alojamiento a reservar: ";
    cin >> codigoAloj;

    // Validar entrada
    if (cin.fail()) {
        cin.clear(); cin.ignore(1000, '\n');
        cout << endl << "Entrada invalida. Cancelando reserva.\n";
        return;
    }

    Alojamiento* alojamiento = buscarAlojamientoPorCodigo(codigoAloj);
    if (!alojamiento) {
        cout << endl << "Codigo de alojamiento no encontrado.\n";
        return;
    }

    cout << endl << "Ingrese la fecha de entrada (YYYY-MM-DD): ";
    cin >> fecha;

    cout << endl << "Duracion de la estancia (en dias): ";
    cin >> duracion;

    // Verificación de disponibilidad
    if (!alojamiento->estaDisponible(fecha, duracion) ||
        alojamientoOcupadoEnRango(codigoAloj, fecha, duracion)) {
        cout << endl << "El alojamiento no esta disponible en esa fecha.\n";
        return;
    }

    // Validación de código único
    while (true) {
        cout << endl << "Codigo unico para la reserva: ";
        cin >> codReserva;
        if (cin.fail()) {
            cin.clear(); cin.ignore(1000, '\n');
            cout << endl << "Codigo invalido. Intente de nuevo.\n";
```

```
            continue;
        }
        if (!codigoReservaExiste(codReserva)) break;
        cout << endl << "Ese codigo ya existe. Ingrese otro.\n";
    }

    // Captura de detalles
    cout << endl << "Metodo de pago: ";
    cin >> metodoPago;

    cout << endl << "Fecha de pago (YYYY-MM-DD): ";
    cin >> fechaPago;

    cin.ignore();
    cout << endl << "Anotaciones: ";
    cin.getline(anotaciones, MAX_ANOTACIONES);

    float monto = alojamiento->getPrecio() * duracion;

    Reservacion* nuevaReserva = new Reservacion(
        codReserva, codigoAloj, documentoHuesped,
        fecha, duracion, metodoPago, fechaPago,
        monto, anotaciones
    );

    reservaciones[totalReservas++] = *nuevaReserva;
    alojamiento->agregarReserva(fecha, duracion);

    Huesped* h = buscarHuespedPorDocumento(documentoHuesped);
    if (h) h->agregarReserva(codReserva);

    guardarReservasEnArchivo();

    cout << endl << "Reserva completada. Monto total: $" << monto << "\n";
    delete nuevaReserva;
}

// Verifica si un alojamiento tiene cruce de fechas con reservas existentes
```

```
bool alojamientoOcupadoEnRango(int codAlojamiento, const char* fechaNueva, int
duracionNueva) {
   tm entradaNueva = {};
   if (!convertirFecha(fechaNueva, entradaNueva)) return true;

   time_t tEntradaNueva = mktime(&entradaNueva);
   time_t tFinNueva = tEntradaNueva + duracionNueva * 86400;

   for (int i = 0; i < totalReservas; ++i) {
      if (reservaciones[i].getCodAlojamiento() != codAlojamiento) continue;

      tm entradaExistente = {};
      if (!convertirFecha(reservaciones[i].getFechaEntrada(), entradaExistente))
continue;

      time_t tEntradaExistente = mktime(&entradaExistente);
      time_t tFinExistente = tEntradaExistente + reservaciones[i].getDuracion() * 86400;

      // Validación de cruce de rangos
      if (tEntradaNueva < tFinExistente && tFinNueva > tEntradaExistente) {
         return true;
      }
   }
   return false;
}

// Busca alojamientos disponibles con filtros aplicados
void buscarAlojamientosPorFiltros(const char* fechaEntrada, const char* municipio, int
duracion, float precioMaximo) {
   bool encontrado = false;

   for (int i = 0; i < totalAlojamientos; ++i) {
      Alojamiento& alojamiento = alojamientos[i];

      if (strcmp(alojamiento.getMunicipio(), municipio) == 0 &&
         alojamiento.estaDisponible(fechaEntrada, duracion)) {

         if ((precioMaximo < 0 || alojamiento.getPrecio() <= precioMaximo)) {
            alojamiento.mostrarInfo(); // Muestra datos del alojamiento
```

```
                encontrado = true;
            }
        }
    }

    if (!encontrado) {
        cout << endl << "No se encontraron alojamientos disponibles con esos filtros.\n";
    }
}

// Flujo completo de reserva por filtros
void realizarReservaPorFiltros(const char* documentoHuesped) {
    char fechaDeseada[11], municipio[30];
    int duracion;
    float precioMaximo = -1;

    cout << endl << "Fecha deseada (YYYY-MM-DD): ";
    cin >> fechaDeseada;

    cout << endl << "Municipio: ";
    cin >> municipio;

    cout << endl << "Duracion de la estancia (en dias): ";
    cin >> duracion;

    char deseaFiltrar;
    cout << endl << "¿Desea filtrar por precio maximo? (s/n): ";
    cin >> deseaFiltrar;

    if (deseaFiltrar == 's' || deseaFiltrar == 'S') {
        cout << endl << "Precio maximo por noche: ";
        cin >> precioMaximo;
    }

    cout << endl << "\nAlojamientos disponibles:\n";
    buscarAlojamientosPorFiltros(fechaDeseada, municipio, duracion, precioMaximo);

    int codigo;
```

```
    cout << endl << "Ingrese el codigo del alojamiento que desea reservar (0 para
cancelar): ";
    cin >> codigo;

    if (codigo != 0) {
        reservarAlojamientoPorCodigo(codigo, fechaDeseada, duracion,
documentoHuesped);
    }
}

// Muestra todas las reservas asociadas a los alojamientos de un anfitrión
void verReservasDeAnfitrion(const char* docAnfitrion) {
    bool hay = false;
    for (int i = 0; i < totalReservas; ++i) {
        Alojamiento* a =
buscarAlojamientoPorCodigo(reservaciones[i].getCodAlojamiento());
        if (a && strcmp(a->getAnfitrionDocumento(), docAnfitrion) == 0) {
            hay = true;
            cout << endl << "\n--- Reserva " << reservaciones[i].getCodigo() << " ---\n";
            cout << endl << "Huesped: " << reservaciones[i].getDocumentoHuesped() <<
"\n";
            cout << endl << "Alojamiento: " << reservaciones[i].getCodAlojamiento() <<
"\n";
            cout << endl << "Fecha entrada: " << reservaciones[i].getFechaEntrada() <<
"\n";
            cout << endl << "Duracion: " << reservaciones[i].getDuracion() << " dias\n";
            cout << endl << "Pago: $" << reservaciones[i].getMonto() << " - Metodo: " <<
reservaciones[i].getMetodoPago() << "\n";
            cout << endl << "Notas: " << reservaciones[i].getAnotaciones() << "\n";
        }
    }

    if (!hay) std::cout << "No hay reservas asociadas a sus alojamientos.\n";
}

// Muestra estadísticas de uso: iteraciones y memoria utilizada
void mostrarConsumo() {
    cout << endl << "Iteraciones: " << contadorIteraciones << "\n";
    cout << endl << "Memoria: " << sizeof(Huesped) * totalHuespedes +
```

```
                        sizeof(Anfitrion) * totalAnfitriones +
                        sizeof(Alojamiento) * totalAlojamientos +
                        sizeof(Reservacion) * totalReservas
        << " bytes\n";
}

// Anula una reserva si el usuario tiene autorización (huesped o anfitrión)
void anularReservacion(int codigo, const char* doc) {
    bool encontrada = false;
    for (int i = 0; i < totalReservas; ++i) {
        contadorIteraciones++;
        if (reservaciones[i].getCodigo() == codigo) {
            if (strcmp(reservaciones[i].getDocumentoHuesped(), doc) == 0 ||
                strcmp(reservaciones[i].getDocumentoAnfitrion(), doc) == 0) {
                reservaciones[i].setEstado("anulada");
                encontrada = true;
                break;
            }
        }
    }

    if (encontrada) {
        cout << endl << "Reservacion marcada como anulada.\n";
        guardarReservasEnArchivo();
    } else {
        cout << endl << "No se encontro la reservacion o no tiene permiso.\n";
    }
}

// Pasa reservas anteriores a un archivo histórico si su fecha ya pasó
void actualizarHistorico(const char* hoy) {
    FILE* archivo = fopen("historico_reservas.txt", "a");

    for (int i = 0; i < totalReservas;) {
        contadorIteraciones++;
        if (fechaEsAnterior(reservaciones[i].getFechaEntrada(), hoy)) {
            fprintf(archivo, "%d,%d,%s,%s,%d,%s,%s,%.2f,%s\n",
                    reservaciones[i].getCodigo(),
                    reservaciones[i].getCodAlojamiento(),
```

```
                reservaciones[i].getDocumentoHuesped(),
                reservaciones[i].getFechaEntrada(),
                reservaciones[i].getDuracion(),
                reservaciones[i].getMetodoPago(),
                reservaciones[i].getFechaPago(),
                reservaciones[i].getMonto(),
                reservaciones[i].getAnotaciones());

            for (int j = i; j < totalReservas - 1; ++j)
               reservaciones[j] = reservaciones[j + 1];

            totalReservas--; // Se reduce el total tras mover a histórico
         } else {
            i++;
         }
      }

      fclose(archivo);
      cout << endl << "Historico actualizado.\n";
   }

   // Carga los datos de los huéspedes desde el archivo huespedes.txt
   void cargarHuespedesDesdeArchivo() {
      FILE* archivo = fopen("huespedes.txt", "r");
      if (!archivo) return;

      char doc[20];
      int ant;         // cantidad de alojamientos reservados
      float punt;      // puntuación promedio

      // Lee cada línea del archivo y crea un objeto Huesped
      while (fscanf(archivo, "%19[^,],%d,%f\n", doc, &ant, &punt) == 3 &&
            totalHuespedes < MAX_HUESPEDES) {
         huespedes[totalHuespedes++] = Huesped(doc, ant, punt);
      }

      fclose(archivo);
   }
```

```
// Carga los datos de los anfitriones desde el archivo anfitriones.txt
void cargarAnfitrionesDesdeArchivo() {
    FILE* archivo = fopen("anfitriones.txt", "r");
    if (!archivo) return;

    char doc[20];
    int ant;
    float punt;

    while (fscanf(archivo, "%19[^,],%d,%f\n", doc, &ant, &punt) == 3 &&
           totalAnfitriones < MAX_ANFITRIONES) {
        anfitriones[totalAnfitriones++] = Anfitrion(doc, ant, punt);
    }

    fclose(archivo);
}

// Carga los datos de los alojamientos desde el archivo alojamientos.txt
void cargarAlojamientosDesdeArchivo() {
    FILE* archivo = fopen("alojamientos.txt", "r");
    if (!archivo) return;

    char linea[5000]; // buffer para la línea completa

    while (fgets(linea, sizeof(linea), archivo) &&
           totalAlojamientos < MAX_ALOJAMIENTOS) {
        int cod;
        char nombre[100], docAnfitrion[20], departamento[30], municipio[30];
        char tipo[10], direccion[100];
        float precio;
        int ascensor, piscina, aire, caja, parqueadero, patio;
        char fechasStr[4000];  // Fechas reservadas separadas por ";"

        int leidos = sscanf(linea,
"%d,%99[^,],%19[^,],%29[^,],%29[^,],%9[^,],%99[^,],%f,%d,%d,%d,%d,%d,%d,%399
9[^\n]",
                            &cod, nombre, docAnfitrion, departamento, municipio, tipo, direccion,
&precio,
                            &ascensor, &piscina, &aire, &caja, &parqueadero, &patio, fechasStr);
```

```
        if (leidos != 15) {
            printf("Error en el formato de la linea: %s\n", linea);
            continue;
        }

        Alojamiento& a = alojamientos[totalAlojamientos++];
        a = Alojamiento(cod, nombre, docAnfitrion, departamento, municipio, tipo,
direccion, precio);
        // TODO: Considerar método para setAmenidades si es necesario

        // Procesamiento de fechas reservadas
        int index = 0;
        char* token = strtok(fechasStr, ";");
        while (token && index < 365) {
            a.setFechaReservada(index, token);
            token = strtok(nullptr, ";");
            ++index;
        }
    }

    fclose(archivo);
}

// Carga las reservas desde el archivo reservas.txt
void cargarReservasDesdeArchivo() {
    FILE* archivo = fopen("reservas.txt", "r");
    if (!archivo) return;

    int cod, codAloj, dur;
    float monto;
    char docH[20], fEntrada[11], metodo[20], fPago[11], notas[MAX_ANOTACIONES],
estado[10];

    while (fscanf(archivo,
"%d,%d,%19[^,],%10[^,],%d,%19[^,],%10[^,],%f,%199[^,],%9[^\n]\n",
                  &cod, &codAloj, docH, fEntrada, &dur, metodo, fPago, &monto, notas,
estado) == 10 &&
           totalReservas < MAX_RESERVAS) {
```

```
        Reservacion r(cod, codAloj, docH, fEntrada, dur, metodo, fPago, monto, notas);
        r.setEstado(estado); // Estado leído desde archivo
        reservaciones[totalReservas++] = r;
    }

    fclose(archivo);
}

// Guarda todas las reservas en el archivo reservas.txt
void guardarReservasEnArchivo() {
    FILE* archivo = fopen("reservas.txt", "w");
    if (!archivo) return;

    char anotacionesLimpias[200];
    for (int i = 0; i < totalReservas; ++i) {
        // Sanitizar anotaciones reemplazando comas con punto y coma
        strncpy(anotacionesLimpias, reservaciones[i].getAnotaciones(),
sizeof(anotacionesLimpias));
        anotacionesLimpias[sizeof(anotacionesLimpias) - 1] = '\0';

        for (int j = 0; anotacionesLimpias[j] != '\0'; ++j) {
            if (anotacionesLimpias[j] == ',') {
                anotacionesLimpias[j] = ';';
            }
        }

        // Escritura formateada
        fprintf(archivo, "%d,%d,%s,%s,%d,%s,%s,%.2f,%s,%s\n",
                reservaciones[i].getCodigo(),
                reservaciones[i].getCodAlojamiento(),
                reservaciones[i].getDocumentoHuesped(),
                reservaciones[i].getFechaEntrada(),
                reservaciones[i].getDuracion(),
                reservaciones[i].getMetodoPago(),
                reservaciones[i].getFechaPago(),
                reservaciones[i].getMonto(),
                anotacionesLimpias,
                reservaciones[i].getEstado());
```

```
    }

    fclose(archivo);
}


// Menú principal del programa
void menuPrincipal() {
    char documento[20];
    int opcion;

    do {
        cout << endl << "\n---------- UdeAStay ----------\n";
        cout << "1. Ingresar como huesped\n";
        cout << "2. Ingresar como anfitrion\n";
        cout << "3. Salir\n";
        cout << "Opcion: ";
        cin >> opcion;

        switch (opcion) {
        case 1:
            cout << "\nDocumento del huesped: ";
            cin >> documento;
            if (buscarHuespedPorDocumento(documento)) {
                cout << "\nBienvenido huesped.\n";
                menuHuesped(documento);
            } else cout << "\nNo encontrado.\n";
            break;

        case 2:
            cout << "\nDocumento del anfitrion: ";
            cin >> documento;
            if (buscarAnfitrionPorDocumento(documento)) {
                cout << "\nBienvenido anfitrion.\n";
                menuAnfitrion(documento);
            } else cout << "\nNo encontrado.\n";
            break;
        }
    } while (opcion != 3);
```

```
}

// Menú de funcionalidades disponibles para anfitriones
void menuAnfitrion(const char* documento) {
    int op;
    do {
        cout << "\n--- Menu Anfitrion ---\n";
        cout << "1. Ver reservas en mis alojamientos\n";
        cout << "2. Anular una reserva\n";
        cout << "3. Consultar consumo\n";
        cout << "4. Volver al menu principal\n";
        cout << "Opcion: ";
        cin >> op;

        switch (op) {
        case 1:
            verReservasDeAnfitrion(documento);
            break;
        case 2: {
            int codigo;
            cout << "\nCodigo de la reserva a anular: ";
            cin >> codigo;
            anularReservacion(codigo, documento);
            break;
        }
        case 3:
            mostrarConsumo();
            break;
        }
    } while (op != 4);
}

// Menú de funcionalidades disponibles para huéspedes
void menuHuesped(const char* documento) {
    int op;
    do {
        cout << "\n--- Menu Huesped ---\n";
        cout << "1. Realizar una reserva por filtros\n";
        cout << "2. Realizar una reserva por codigo\n";
```

```
cout << "3. Anular una reserva\n";
cout << "4. Consultar consumo\n";
cout << "5. Volver al menu principal\n";
cout << "Opcion: ";
cin >> op;

switch (op) {
case 1:
   realizarReservaPorFiltros(documento);
   break;
case 2:
   realizarReservaPorCodigo(documento);
   break;
case 3: {
   int codigo;
   cout << "\nCodigo de la
```

e. Problemas de desarrollo que afrontó.

1. Gestión de la memoria, ya que realicé prácticamente tres implementaciones buscando usar de la mejor manera la memoria dinámica, sin embargo, cuando quise usarla para implementar el código casi que en un 50% con ello, me enfrenté a problemas al momento de leer y procesar los archivos .txt. Este problema se presentaba principalmente al intentar validar si en una fecha específica un hospedaje estaba o no disponible. Al final opté por una implementación con un uso limitado de la memoria dinámica restringiendo la cantidad de datos a procesar a través de variables fijas, a las cuales les puedo modificar el valor de ser necesario. Esto apoyándome en lo dicho por el profesor con respecto a que en la sustentación se nos proporciona un repositorio y nos indican la cantidad de información a procesar, por lo tanto, el manejo que planteo me permite modificar los rangos con facilidad.

2. La carga de los archivos .txt me presentó un reto importante, si bien, dentro de los contenidos del laboratorio habíamos visto una introducción al tema y lo habías implementado durante el laboratorio 3, los datos requeridos especialmente para alojamientos y reservas presentaron una complejidad mayor para su organización y posterior uso dentro de las demás funciones. Adicional, se debe tener especial cuidado para estructurarlos de tal forma que para la sustentación no se presenten problemas al adaptarlos con el repositorio proporcionado.

3. Para el manejo de las amenidades realicé varias implementaciones buscando el mejor manejo posible que se pudiera adaptar fácilmente a los archivos de entrada, finalmente, decidí implementar una subclase dentro de la clase alojamiento, la cual al final funciona como un método de esa clase, lo cual me permite controlar a través del archivo de alojamientos, si un alojamiento particular cuenta con determinado tipo de amenidad que define de manera estática dentro de la clase.

4. Al ser mi primera experiencia con el paradigma de la POO, me costó gran trabajo poder definir los atributos y métodos necesarios de cada clase, partiendo del hecho de que las clases, en términos generales, estaban definidas prácticamente en el enunciado del desafío. Sin embargo, en primera instancia definí unos atributos y métodos que se me fueron expandiendo a medida que iba implementando todo el código, aún cuando realicé un análisis previo a conciencia y teniendo en cuenta todo lo que consideré necesario para la solución. Para futuros problemas tendré más cuidado al momento de definir las clases para evitar los contratiempos que tuve en este para dar la solución.

5. Para la búsqueda de los alojamientos por filtro, tuve grandes inconvenientes, ya que no sabía exactamente como estructurar el código para emular de qué municipio era el alojamiento o en qué fechas estaba reservada, principalmente esta última me requirió un gasto de tiempo importante, para garantizar que se pudiera cumplir con ello, no solo por el tema de la memoria dinámica que expuse anteriormente, sino también por la lectura de los archivos dentro .txt.

6. Para cargar las reservas en el archivo reservas.txt tuve inconvenientes debido a la manera como estaba accediendo a ellos, inicialmente opté por una lectura tipo a, sin embargo, esto me presentó un problema mayor luego al tratar de cargar el histórico de reservas, al final decidí dejar la lectura con tipo r garantizando que durante la iteración el programa me guardara todas las reservas en el archivo y poder tomarlo para alimentar la carga del histórico.

7. Para la anulación de las reservas pensé desde el principio en anular la reserva directamente del archivo de reservas.txt, pero con el transcurso de la implementación pude evidenciar que era necesario que esas reservas que se anulaban quedaran en algún lugar, propiciando buenas prácticas a la hora de guardar el histórico, por lo tanto, dispuse una variable tipo char para marcar la reserva anulada y de esta manera mantenerla disponible dentro del archivo de reservas.txt y de ser necesario trasladarla luego al archivo histórico.

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

Para la elaboración de este desafío comencé realizando un análisis preliminar de las condiciones planteadas, para tener un panorama que me permitiera definir el rumbo a tomar, con este primer acercamiento pude definir claramente que las clases a utilizar para seguir el paradigma de la POO debían ser los expuestos directamente en el enunciado: huéspedes, alojamientos, anfitriones y reservas.

Con una posterior lectura más detallada del problema, pude identificar las condiciones que tendrían cada una de esas clases que debía definir, identificando la importancia del documento, la puntuación y la antigüedad para las clases huéspedes y anfitriones. Esto en concordancia con la información que debía cargar desde los archivos .txt que alimentan el sistema.

Dentro de la clase reservación es importante tener en cuenta el código de la reserva, la fecha, la duración, el código del alojamiento reservado, los cuales son relevantes para poder alimentar el archivo de reservas.txt que posteriormente me servirá para el archivo de reservas histórico. Para la clase alojamientos es importante definir el código, la ciudad, el costo y las posibles amenidades que pueda tener, importantes al momento de elegir el alojamiento por parte del huésped.

Luego de ese análisis que me permitió definir las clases completamente, venía el gran reto de definir los atributos privados y los métodos públicos que debían tener las clases definidas, para ello, comencé a implementar una posible solución basándome en los requisitos mínimos que pedía el desafío.

Lo primero fue pensar en la forma cómo iba a cargar la información proveniente de los archivos y cómo guardar la información de las reservas. Para ello me basé en la metodología proporcionada en el laboratorio del curso, junto con bibliografía recomendada, utilizando una carga en secuencia para realizar la carga con este método según recomendación de los docentes.

En segundo lugar, implementé las funciones que me permitieran realizar las reservas ya fuera por filtro o por código y alinear esas funciones con los datos necesarios para realizarlas, como son el código, ciudad y costo del alojamiento y las fechas posibles para realizar la reserva. Para implementar las funciones de reservación fueron necesarias otras funciones que denominé auxiliares, que me permitían buscar un huésped o un anfitrión por documento, buscar un alojamiento por el código, convertir la fecha en el formato que

definí YYYY-MM-DD, verificar si un alojamiento estaba ocupado en una fecha o verificar si el código de la reserva estaba ya ocupado.

Una vez definida la estructura general que quería seguir, implementé las funciones para realizar la anulación de la reserva, que en concordancia con lo que comenté en la sección de problemas afrontados, me presentó un reto importante para poder garantizar que esas reservas anuladas no desaparecieran del radar dentro de la implementación. Al final, decidí marcar las reservas anuladas para mantenerlas dentro del archivo y poder usarlas en caso de ser necesario.

Paso seguido, implementé la función para calcular el consumo, basado en los bytes utilizados por las clases y las iteraciones necesarias para realizar el proceso. De igual manera, implementé la función que me permite cargar el histórico a partir de una fecha, con su respectiva función auxiliar para verificar que fechas son anteriores.

Con esta implementación puedo lograr un funcionamiento con los requisitos mínimos establecidos y garantizando un mínimo de eficiencia acorde con los estándares de programación definidos en el curso. Sin embargo, hago la aclaración importante que el código presentado es susceptible de grandes mejoras, en cuanto a la implementación usada y en aspectos específicos como el uso de la memoria dinámica y la eficiencia.

Aún así, pude culminar con lo propuesto en el desafío, con un aprendizaje muy positivo sobre el paradigma de la POO y con mucha motivación para seguir aprendiendo los contenidos finales del curso, espero en un futuro poder complementar esta implementación para afianzar el conocimiento y establecer mejores prácticas en cuanto al uso de la memoria dinámica y la eficiencia.