DOCUMENTACIÓN TECNICA SISTEMA DE RESERVACIONES EN UN HOTEL

INDICE:

CONTENIDO:

INTRODUCCION:

Este sistema fue diseñado para facilitar el manejo de información de las reservas de un hotel, proporcionando las opciones de: crear, modificar, cancelar, buscar y mostrar todas las reservas; está construido en C y hace uso de punteros, memoria dinámica y estructuras para facilitar la manipulación de los datos de cada reserva.

FLUJO DE TRABAJO / EQUIPO:

1.1 Enfoque:

El diseño sigue un enfoque funcional y modular, usando también el inicio de los fundamentos de la programación orientada a objetos, ya que para simular una reserva se utilizó una estructura.

1.2 Flujo de pensamiento:

Ya que es un proyecto grande lo que se hizo para mantener el orden y entendimiento del programa fue primero crear una base robusta de lógica en las funciones principales, es decir, primero hicimos el menú y las funciones que iba a utilizar el mismo, después decidimos que lo mejor sería dividir el flujo de trabajo en primero terminar las funciones principales y después resolver los posibles problemas que existían en el desarrollo.

1.3 División de tareas:

Nuestro equipo decidió que cada uno haría una de las tres cosas: César el código, Dulce María la presentación y Axel la documentación del código, para que esto funcionara lo primero que se hizo fue el código, ya que el mismo fue completado, nos reunimos 2 días, 2 horas para la explicación de cómo funcionaba cada parte del código, y así todos los miembros del equipo pudieran hacer su parte del trabajo.

1.4 Funcionamiento general del código:

El código tiene 3 módulos principales: el arreglo de reservaciones, las operaciones CRUD y la verificación, cálculo de datos; para esto hay 2 variables globales que tener en cuenta para el funcionamiento de este: **capacidad actual:** es la capacidad del arreglo y **numero de reservas:** es el número actual de reservaciones.

FUNCIONES PRINCIPALES:

2.1 Operaciones CRUD:

Para la creación de las funciones principales del sistema (registrar, modificar, buscar, cancelar y mostrar) se usó la lógica de las operaciones CRUD (create, read, update y delete) ya que era todo lo que el sistema necesitaba al inicio, ninguna de las funciones que se basan en estas operaciones tienen valor de retorno.

2.2 Registrar reservas:

Esta función solo es una cadena de lectura, verificación y almacenamiento de datos, usa scanf() y fgets() dependiendo de si lo que se está almacenando es una cadena u otro tipo de dato, cada que esta función se ejecuta se incrementa + 1 la variable global num_reservas. Seudocódigo: *(material extra-2.1)*

2.3 Modificar reservas:

Esta función muestra un menú de opciones para modificar alguno de los campos de una reservación existente, si no existen reservaciones simplemente se imprime que no hay reservaciones existentes, si existen reservaciones entonces se pide que se introduzca el ID de la reservación para cambiar alguno de sus datos; hace uso de la verificación de datos a la hora de cambiar alguno de los datos; a su vez modifica el precio si se cambió la fecha o desayuno de la reservación. Seudocódigo: *(material extra-2.2)*

2.4 Cancelar reservas:

Esta función elimina una reservación en base a su ID, si no existen reservaciones entonces solo se imprime que no hay reservaciones existentes, para eliminar la reservación lo que se hace es asignarle 0 a cada uno de sus campos y a las cadenas se les asigna “” cada que se ejecuta esta función se decrementa la variable global num_reservas – 1. Seudocódigo: *(material extra-2.3)*

2.5 Buscar reserva:

Esta función tiene la misma estructura de la anterior, toma un ID, después si se encuentra una reservación con el ID proporcionado se imprimen todos sus datos. Seudocódigo: *(material extra-2.4)*

2.6 Mostrar reservas:

Esta función solo itera por las reservaciones existentes e imprime sus datos. Seudocódigo: *(material extra-2.5)*

SOLUCIÓN DE PROBLEMAS:

3.1 Primer problema | estructuras de datos:

El primer problema con el que nos encontramos fue como relacionar los diferentes tipos de datos que necesitaba cada reserva: cadenas, enteros, flotantes y booleanos en una sola estructura; para resolver eso hicimos un struct al que llamamos Reserva *(material extra-3.1)*, en el mismo almacenamos ID, nombre, fecha de entrada, fecha de salida, número de habitación, precio, desayuno y número de días de estancia, cada uno de los campos con su respectivo tipo de dato *(tabla 3.1).*

ESTRUCTURA DE DATOS: RESERVACIÓN	
Dato y su tipo	**Ejemplo**
ID *(entero)*	5
Nombre *(cadena)*	César Pérez Amador
Fecha entrada *(cadena)*	10/10/2024
Fecha salida *(cadena)*	14/10/2024
Número de habitación *(entero)*	003
Precio *(flotante)*	3200.00
Desayuno *(booleano)*	true
Número de días *(entero)*	4

Tabla 1.1

3.2 Segundo problema | límites de memoria:

El segundo problema con el que nos encontramos fue que nuestro arreglo de reservaciones no servía ya que no había forma de saber el número de reservaciones máximo, por ejemplo, imaginemos que tenemos un arreglo del struct Reserva definido así: *struct Reserva reservaciones[100]*, cuando este número se alcanza ya no se pueden incluir más reservas, lo que limita nuestro sistema, lo que hicimos para solucionar este problema fue hacer uso del heap (memoria dinámica de C), asignando y reasignando memoria dependiendo del uso que le estemos dando al sistema, esta parte del código está dividido en 3 partes principales: la declaración del arreglo, la asignación / reasignación de memoria en una función y la liberación de la memoria al final del programa, cabe aclarar que este proceso contiene error handling para evitar errores de asignación de memoria.

3.2.1 Declaración del arreglo:

Cuando usamos un arreglo estático sabemos dos cosas: 1. el nombre del arreglo es un puntero al primer elemento de este y 2. su tamaño en memoria es contiguo, es decir si tenemos un arreglo de 100 enteros y un entero en C ocupa 4 bytes en memoria entonces nuestro arreglo va a ocupar 400 bytes en memoria, estos dos principios se mantienen cuando usamos la asignación de memoria en C, cuando declaramos el arreglo lo hacemos como un puntero del tipo Reserva *(material extra-3.2)*, es decir un puntero al primer elemento del arreglo.

3.2.2 Asignación de memoria:

Ya que declaramos el arreglo, hicimos una función para asignar o reasignar memoria en base al número de reservas que existen y la capacidad actual del arreglo, después definimos el incremento que se va a hacer cada que se llame la función; pero ¿por qué? Esto se hace porque no queremos que se reasigne memoria cada que se llama a la función, esto es ineficiente y usa mucho tiempo de ejecución, entonces lo que hacemos es ir en incrementos de 10, así la función solo reasigna memoria cada 10 reservas; en la función lo primero que hacemos es que si mi capacidad actual es 0 y llamamos la función entonces le asignamos el valor de un incremento a la capacidad actual, y después le ASIGNAMOS al puntero la cantidad actual * el tamaño en memoria de una Reserva, es decir 10 * sizeof(Reserva), si el valor del puntero es NULO entonces el programa termina, ahora, si se llama la función hay dos posibles resultados en el diagrama de flujo *(figura 3.1)*: si la capacidad actual es menor o igual al número de reservas, entonces se incrementa la capacidad actual + el incremento de capacidad y se le reasigna la capacidad actual al arreglo de Reservas; si la capacidad es mayor entonces no se hace nada; al final de la ejecución liberamos la memoria del arreglo.

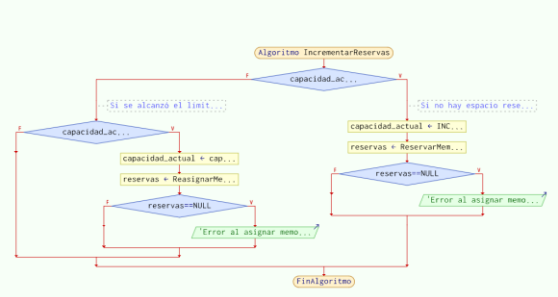

Figura 3.1

3.3 Tercer problema | lógica de fechas y verificación de datos:

Con la base de lectura y almacenamiento de los datos ya terminada el paso que tomamos después fue la verificación de datos, en específico la verificación, comparación y cálculo de fechas, para cada uno de estos se implementó una función que se encargara de cada uno de estos procesos.

3.3.1 Conversión de fechas:

En nuestro struct Reserva declaramos la fecha como una cadena, cosa que nos complicaba la manipulación de los datos ya que estos son caracteres y las operaciones con los mismos no nos iban a dejar hacer las comparaciones, por lo que lo primero que se hizo fue conversión de caracteres a enteros, lo primero fue delimitar el formato de una fecha correcta, en el sistema una fecha correcta tiene los siguientes parámetros: la fecha contiene exactamente 10 caracteres: DD/MM/AAAA, también sabemos que una cadena es un arreglo de caracteres, por lo que podemos tomar un índice y verificar carácter por carácter, en este caso sabemos que existe en cada índice, sabemos que del índice 0 al 1 existe el día, del índice 3 al 4 existe el mes y que del índice 6 al 9 existe el año, para convertir estos en enteros podemos aplicar la tabla ASCII y operaciones aritméticas, por ejemplo, para el día tomamos el carácter del índice 0 y a este le restamos el carácter ‘0’, y lo sumamos a una variable entera llamada día, después tomamos el resultado y lo multiplicamos por 10, y le sumamos el carácter del índice 1 el carácter ‘0’, para ejemplificar esto pongamos “19”, el carácter ‘1’ en la tabla ASCII vale 49, entonces cuando le restamos ‘0’ (que vale 48) el entero que nos da es 1, lo multiplicamos por 10, y le sumamos el carácter ‘9’ que vale 57, entonces cuando le restamos ‘0’ (que vale 48) el entero que nos da es 9, es decir que, después de cada operación el resultado en entero es 19; esto lo aplicamos a día, mes y año. *(figura 3.2)*

```
int dia = (fecha[0] - '0') * 10 + (fecha[1] - '0');
int mes = (fecha[3] - '0') * 10 + (fecha[4] - '0');
int anio = (fecha[6] - '0') * 1000 + (fecha[7] - '0') * 100 + (fecha[8] - '0') * 10 + (fecha[9] - '0');
```

Figura 3.2

3.3.2 Verificación de fechas:

Ahora aplicando la misma lógica acerca de las cadenas, podemos saber que si la fecha debe tener exactamente 10 caracteres y que en los índices 2 y 5 tienen que haber caracteres ‘/’ si no significa que la fecha es invalida, después de esto convertimos las fechas y las guardamos en variables enteras, ahora definimos un arreglo de enteros que contenga el número de días que hay en cada mes, después simplemente en base a diferentes condiciones, retornamos falso si la fecha es invalida, por ejemplo que los días no sean mayores a la cantidad de días en el mes introducido, también se implementaron años bisiestos, por lo que si el usuario introduce 29 de febrero en un año bisiesto esta fecha es válida. *(figura 3.3)*

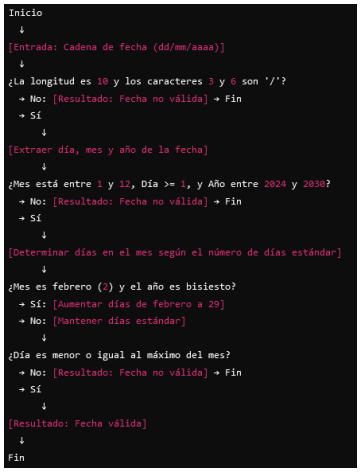

Figura 3.3

3.3.3 Comparación de fechas:

En esta función se compara cuál de dos fechas es anterior, primero se convierte la fecha con el método mencionado en el punto **3.3.1**, posteriormente la función evalúa las fechas en orden de importancia: primero el año, luego el mes y finalmente el día. Si los años son diferentes, la comparación se realiza entre ellos. Si son iguales, se procede a comparar los meses, y si estos también son iguales, se comparan los días. Por último, la función retorna true si la primera fecha es anterior a la segunda y false en caso contrario.

3.3.4 Cálculo de fechas:

En esta función, se calcula la diferencia en días entre dos fechas. Primero, se descomponen las fechas en valores enteros representando día, mes y año, utilizando el mismo método mencionado en el punto 3.3.1. Luego, se transforma cada fecha en el número total de días transcurridos desde un punto de referencia común (en este caso, el año 2024). El cálculo del total de días acumulados por cada fecha incluye: **Años completos:** Se suman los días correspondientes a los años completos entre el año base (2024) y el año de la fecha, considerando si el año es bisiesto o no (366 o 365 días). **Meses completos:** Se suman los días de los meses anteriores al mes de la fecha, según un arreglo predefinido de días por mes. **Días individuales:** Se agrega el número de días transcurridos dentro del mes actual. Finalmente, la diferencia en días entre las dos fechas se obtiene restando el total de días acumulados de la primera fecha al de la segunda. Esto da como resultado el número exacto de días que separan ambas fechas.

3.3.5 Verificación de cuarto:

Esta función verifica dos cosas, primero verifica que el número de cuarto se encuentre en el rango de 1 a 740, nuestro hotel cuenta con 8 pisos y 40 habitaciones por piso, por lo que hay un máximo de 740 habitaciones, después, usando lógica de fechas verifica que no se pueda reservar el mismo cuarto en el mismo periodo de fechas, por ejemplo, si la habitación 1 está reservada del 01 de enero al 08, esta habitación no se pueda reservar en este período de tiempo, esto se logra gracias al punto **3.3.3 Comparar fecha** y sigue esta lógica: primero verifica que exista una reservación (que no sea la que estamos haciendo) que tenga el mismo cuarto, de ser así, verificamos que las fechas no se solapen: Si la fecha de salida de la nueva reservación es menor a la fecha de entrada de la reservación a evaluar, entonces las fechas no se solapan. Si la fecha de salida de la reservación a evaluar es menor a la fecha de entrada de la nueva reservación, entonces las fechas tampoco se solapan. Ahora, si ambas condiciones son falsas, es decir, si la fecha de salida de la nueva reservación es mayor a la fecha de entrada de la reservación evaluada y la fecha de salida de la antigua reservación es mayor a la fecha de entrada de la nueva reservación, entonces las fechas se solapan. *(Material extra-4.1 a material extra-4.3)*

NIVEL DE ACCESO DE LAS FUNCIONES:

Para tener un fundamento más robusto en cuanto a seguridad y jerarquía de acceso en las funciones que tienen argumentos o parámetros se siguió el principio de menor privilegio, solo dando el nivel de acceso necesario a cada función, en este caso ninguna de las funciones necesitaba cambiar ni el valor dentro de la dirección de memoria ni la dirección de memoria a la que sea apuntaba entonces todas las funciones solo tienen permisos de lectura y están declaradas de la siguiente manera *(figura 4.1)*

```
void incrementar_reservas();
void registrar_reserva();
void modificar_reserva();
void cancelar_reserva();
void buscar_reserva();
void mostrar_reservas();
bool verificar_cuarto(const int*const, const char*const, const char*const, const int*const);
bool verificar_fecha(const char*const);
bool comparar_fechas(const char*const, const char*const);
int calcular_fecha(const char*const, const char*const);
bool isBisiesto(int);
```

Figura 4.1

CONTRIBUCIONES:

Si deseas contribuir al proyecto puedes hacer un pull request en el repositorio de GitHub, por favor comenta tu código y mantén la claridad en el código, de cualquier manera, el proyecto está bajo la licencia de MIT, es decir que puedes usarlo, modificarlo o hacer lo que quieras mientras se incluya la licencia de este.

SI LLEGASTE HASTA AQUI GRACIAS POR LEER TODO :)

MATERIAL EXTRA:

2. SEUDOCÓDIGOS (híper simplificados):

2.1 Registrar reserva:

```
función registrar_reserva
  si no hay espacio suficiente en reservas:
    incrementar tamaño de reservas
  solicitar datos del cliente
  si las fechas son inválidas:
    mostrar error y pedir fechas nuevamente
  si el número de habitación ya está reservado:
    mostrar error y pedir otro número
  si se requiere desayuno:
    asignar costo adicional de desayuno
  calcular la cantidad de días de la reserva
  calcular el costo total
  asignar un ID único a la reserva
  mostrar mensaje de éxito y detalles de la reserva
  incrementar contador de reservas
fin
```

2.2 Modificar reserva:

```
función modificar_reserva
  si no hay reservas:
    mostrar mensaje de error
  solicitar el ID de la reserva a modificar
  buscar la reserva con el ID
  si la reserva es encontrada:
    mostrar opciones para modificar (nombre, fechas, número de habitación, desayuno)
    realizar la modificación elegida
    si se modifica la fecha, recalcular el precio
    si se modifica el desayuno, ajustar el costo
    mostrar mensaje de éxito y nuevo precio
  si no se encuentra la reserva:
    mostrar mensaje de error
fin
```

2.3 Cancelar reserva:

```
función cancelar_reserva
  si no hay reservas:
    mostrar mensaje de error
  solicitar el ID de la reserva a cancelar
  buscar la reserva con el ID
  si la reserva es encontrada:
    borrar los datos de la reserva (ID, nombre, fechas, habitación, precio, etc.)
    disminuir el contador de reservas
    mostrar mensaje de éxito
  si no se encuentra la reserva:
    mostrar mensaje de error
fin
```

2.4 Buscar reserva:

```
función buscar_reserva
  si no hay reservas:
    mostrar mensaje de error
  solicitar el ID de la reserva a buscar
  buscar la reserva con el ID
  si la reserva es encontrada:
    mostrar los detalles de la reserva
  si no se encuentra la reserva:
    mostrar mensaje de error
fin
```

2.5 Mostrar reservas:

```
función mostrar_reservas
  si no hay reservas:
    mostrar mensaje de error
  para cada reserva en el arreglo de reservas:
    mostrar los detalles de la reserva
fin
```

3. CÓDIGO:

3.1 Struct Reserva:

```
struct Reserva {
  int ID;
  char fecha_entrada[20];
  char fecha_salida[20];
  char nombre[50];
  float precio;
  int numero_de_habitacion;
  bool desayuno;
  int num_dias;
};
```

3.2 Puntero al arreglo de Reserva:

```
// Puntero al primer elemento de un arreglo de Reserva
struct Reserva *reservas;
// Numero actual de reservas
int num_reservas = 0;
// Capacidad actual en el arreglo de Reserva
int capacidad_actual = 0;
```

4. DIAGRAMAS:

4.1 Ejemplo solapar fechas 1:

```
10/10/2024        12/10/2024        13/10/2024        15/10/2024
    |-----------------|                 |-----------------|
Reserva 1:       [Entrada--------------Salida]
                            Reserva 2:       [Entrada------Salida]

1. `salida1 <= entrada2` → 12/10 <= 13/10 → **Verdadero** (no se solapan)
2. `salida2 <= entrada1` → 15/10 <= 10/10 → **Falso** (no se solapan)

Resultado: **No se solapan.**
```

4.2 Ejemplo solapar fechas 2:

```
10/10/2024        12/10/2024        14/10/2024        16/10/2024
    |-----------------|-----------------|-----------------|
Reserva 1:       [Entrada-----------Salida]
                  Reserva 2:    [Entrada-----------Salida]

1. `salida1 <= entrada2` → 14/10 <= 12/10 → **Falso** (no se solapan completamente)
2. `salida2 <= entrada1` → 16/10 <= 10/10 → **Falso** (no se solapan completamente)

Resultado: **Sí se solapan parcialmente** del 12/10 al 14/10.
```

4.3 Ejemplo solapar fechas 3:

```
10/10/2024        12/10/2024        14/10/2024        15/10/2024
    |-----------------|-----------------|-----------------|
Reserva 1:       [Entrada-----------Salida]
                            Reserva 2:    [Entrada-------Salida]

1. `salida1 <= entrada2` → 15/10 <= 12/10 → **Falso** (no se solapan completamente)
2. `salida2 <= entrada1` → 14/10 <= 10/10 → **Falso** (no se solapan completamente)

Resultado: **Sí se solapan** (Reserva 2 está completamente dentro de la Reserva 1).
```