


PORTADA

Nombre Alumno / DNI	Pablo Nicolás Soto Irago / 02583827F
Título del Programa	Bachelor Degree in Computer Science & Artificial Intelligence
Nº Unidad y Título	UNIT 20 - APPLIED PROGRAMMING & DESIGN PRINCIPLES
Año académico	2024/2025
Profesor de la unidad	Ángel Bravo
Título del Assignment	Assignment Brief
Día de emisión	10/10/2025
Día de entrega	21/01/2025
Nombre IV y fecha	
Declaración del estudiante	<p><b>Certifico que la presentación del assignment es completamente mi propio trabajo y entiendo completamente las consecuencias del plagio. Entiendo que hacer una declaración falsa es una forma de mala práctica.</b></p> <p><b>Fecha: 21/01/2025</b></p> <p><b>Firma del alumno:</b></p> 

**Plagio**

*El plagio es una forma particular de hacer trampa. El plagio debe evitarse a toda costa y los alumnos que infrinjan las reglas, aunque sea inocentemente, pueden ser sancionados. Es su responsabilidad asegurarse de comprender las prácticas de referencia correctas. Como alumno de nivel universitario, se espera que utilice las referencias adecuadas en todo momento y mantenga notas cuidadosamente detalladas de todas sus fuentes de materiales para el material que ha utilizado en su trabajo, incluido cualquier material descargado de Internet. Consulte al profesor de la unidad correspondiente o al tutor del curso si necesita más consejos.*



---

# UNIT 20 - APPLIED PROGRAMMING & DESIGN PRINCIPLES

---

ENTREGA FINAL



21 DE ENERO DE 2025  
PABLO NICOLÁS SOTO IRAGO  
Computer Science & AI

## Contenido

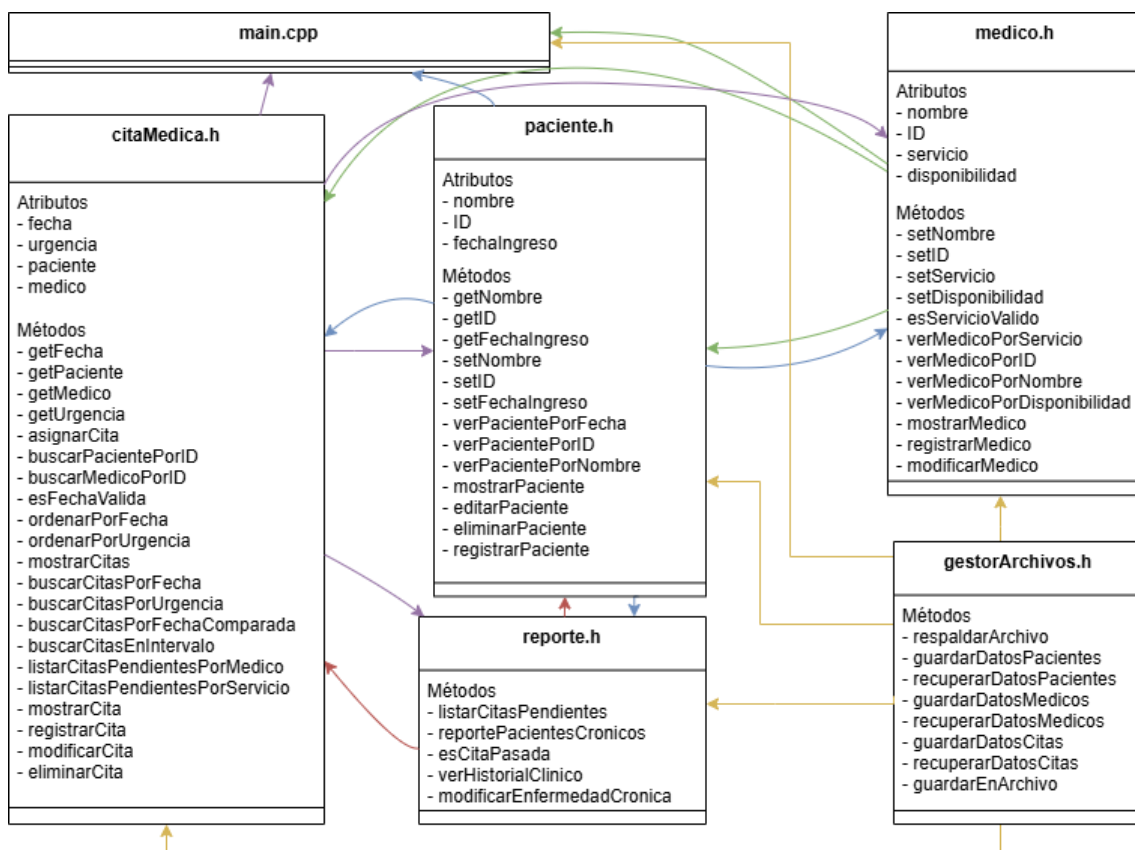
Marco General del Proyecto .....	3
Diagrama general del proyecto .....	3
Código .....	3
#include.....	3
citaMedica.h .....	4
Clase CitaMedica .....	4
Listado de funciones en la clase CitaMedica .....	5
Funciones y métodos llamados desde otros archivos .....	10
Explicación detallada de las decisiones de diseño.....	10
gestorArchivos.h.....	11
Listado de funciones en la clase GestorArchivos .....	12
Funciones y métodos llamados desde otros archivos .....	14
Explicación detallada de las decisiones de diseño.....	14
medico.h.....	15
Clase Medico .....	15
Listado de funciones en la clase Medico .....	16
Funciones y métodos llamados desde otros archivos .....	19
Explicación detallada de las decisiones de diseño.....	19
paciente.h .....	20
Clase Paciente .....	20
Listado de funciones en la clase Paciente .....	21
Explicación detallada de las decisiones de diseño.....	23
reporte.h .....	24
Listado de funciones en reporte.h.....	24
Funciones privadas .....	25
Funciones y métodos llamados desde otros archivos .....	26
Explicación detallada de las decisiones de diseño.....	26
main.cpp.....	27
Listado de funciones en main.cpp .....	27
Funciones y métodos llamados desde otros archivos .....	28
Explicación detallada de las decisiones de diseño.....	30

Archivos.txt .....	30
pacientes.txt .....	30
medicos.txt.....	31
citas.txt .....	31
pacientes_cronicos.txt .....	31
servicios.txt .....	31
carpeta backup .....	31
Algoritmos de Búsqueda y Ordenación .....	32
Búsqueda lineal.....	32
Búsqueda de pacientes por ID en main.cpp: .....	32
Búsqueda de médicos por ID en main.cpp: .....	32
Búsqueda de citas médicas por fecha o urgencia en CitaMedica: .....	32
Posibles optimizaciones .....	32
Bubble Short .....	32
Ordenación de citas por fecha en CitaMedica: .....	33
Ordenación de citas por urgencia en CitaMedica: .....	33
Ordenación de citas en el historial clínico (verHistorialClinico):.....	33
Posibles optimizaciones .....	33
Conclusiones .....	33
Bibliografía .....	34
Anexos .....	34

# Marco General del Proyecto

El proyecto consiste en desarrollar un software de gestión hospitalaria, permitiendo la administración de información relacionada con pacientes, médicos, citas y otros servicios. Se utilizarán conceptos de programación orientada a objetos (POO) y manejo eficiente de datos a través de algoritmos de búsqueda y ordenación, así como archivos para el almacenamiento persistente.

## Diagrama general del proyecto



## Código

A continuación, se presenta la descripción del código completo del AB.

### #include

**<iostream>**: Para operaciones de entrada y salida, como `std::cin` y `std::cout`.

**<vector>**: Permite el uso de contenedores dinámicos, como `std::vector`, para manejar colecciones de objetos (Pacientes, Médicos, Citas, etc.).

**<string>**: Proporciona la clase `std::string` para manipular cadenas de texto.

**<fstream>**: Para operaciones de lectura y escritura de archivos (GestorArchivos).

**<ctime>**: Utilizado para obtener y manipular fechas y horas actuales (por ejemplo, en Reporte para manejar fechas de citas).

**<limits>**: Ofrece valores límite para tipos de datos, usado para validar entradas numéricas y manejar errores de entrada.

**<set>**: Proporciona un contenedor para almacenar elementos únicos, usado para conjuntos ordenados y eliminar duplicados.

**<algorithm>**: Incluye funciones estándar para manipulación de datos, como ordenamiento, búsqueda y transformaciones en contenedores.

**"citaMedica.h"**: Incluye la definición de la clase CitaMedica, que gestiona la creación, modificación y eliminación de citas médicas.

**"paciente.h"**: Contiene la definición de la clase Paciente, utilizada para manejar información de los pacientes, como nombre, ID y enfermedades.

**"medico.h"**: Define la clase Medico, que gestiona información sobre los médicos, como servicios ofrecidos, ID y disponibilidad.

**"gestorArchivos.h"**: Proporciona funcionalidades para la lectura y escritura de datos en archivos, como pacientes, médicos y citas médicas.

**"reporte.h"**: Incluye la definición de la clase Reporte, que se encarga de generar informes, como historial clínico y pacientes crónicos.

## citaMedica.h

El archivo citaMedica.h se centra en definir la clase CitaMedica, que maneja la información de las citas médicas. La clase proporciona métodos para acceder a los datos de las citas y así realizar operaciones relacionadas con las ellas.

## Clase CitaMedica

### Atributos

- Fecha (string): La fecha de la cita médica.
- Urgencia (int): Nivel de urgencia de la cita médica.
- Paciente (puntero a Paciente): Puntero al objeto Paciente asociado.
- Medico (puntero a Medico): Puntero al objeto Medico asociado.

### Getters

#### getFecha()

Tipo de retorno: std::string

Descripción: Retorna la fecha de la cita médica almacenada en el atributo fecha.

*getPaciente()*

Tipo de retorno: Paciente\*

Descripción: Retorna un puntero al objeto Paciente asociado a la cita médica.

*getMedico()*

Tipo de retorno: Medico\*

Descripción: Retorna un puntero al objeto Medico asociado a la cita médica.

*getUrgencia()*

Tipo de retorno: int

Descripción: Retorna el nivel de urgencia de la cita médica, representado por un valor entero.

## Listado de funciones en la clase CitaMedica

A continuación, se enumeran las funciones definidas en citaMedica.h, específicamente dentro de la clase CitaMedica{}

*asignarCita*

**void asignarCita()**

Propósito: Guarda una cita médica en el archivo citas.txt con todos sus detalles, como la fecha, el paciente, el médico y la urgencia.

Descripción:

Abre el archivo citas.txt en modo de adición (std::ios::app) para que las nuevas citas se añadan al final del archivo sin modificar los datos existentes.

Escribe la información de la cita: la fecha, el nombre del paciente y médico, el ID del paciente y médico, y el nivel de urgencia.

*buscarPacientePorID*

**static Paciente\* buscarPacientePorID**

**(const std::vector<Paciente\*>& pacientes, int id)**

Propósito: Buscar un paciente en el sistema mediante su ID.

Descripción:

Recorre el vector de punteros a pacientes (std::vector<Paciente\*>) y compara cada ID con el ID proporcionado. Si encuentra un paciente cuyo ID coincide con el proporcionado, devuelve un puntero a ese paciente.

### *buscarMedicoPorID*

```
static Medico* buscarMedicoPorID  
(const std::vector<Medico*>& medicos, int id)
```

Propósito: Buscar un médico en el sistema mediante su ID.

Descripción:

Recorre el vector de punteros a médicos (std::vector<Medico\*>) y compara cada ID con el ID proporcionado. Si encuentra un médico cuyo ID coincide con el proporcionado, devuelve un puntero a ese médico.

### *esFechaValida*

```
static bool esFechaValida(const std::string& fecha)
```

Propósito: Validar que una fecha esté en el formato pedido (DD-MM-AAAA) y que sea una válida (con respecto al sistema).

Descripción:

Comprueba que la fecha tenga el formato DD-MM-AAAA verificando los caracteres en las posiciones correctas.

### *ordenarPorFecha*

```
static void ordenarPorFecha(  
std::vector<CitaMedica*>& citas)
```

Propósito: Ordenar las citas médicas por fecha.

Descripción:

Esta función ordena el vector de citas (std::vector<CitaMedica\*>) en orden ascendente según la fecha. La ordenación permite que las citas se presenten en el orden cronológico. Utiliza una función de comparación que verifica las fechas de las citas para establecer el orden.

### *ordenarPorUrgencia*

```
static void ordenarPorUrgencia  
(std::vector<CitaMedica*>& citas)
```

Propósito: Ordenar las citas médicas por urgencia.

Descripción:



Esta función organiza las citas en el vector (`std::vector<CitaMedica*>`) según el nivel de urgencia, de menor a mayor. Utiliza `std::sort` junto con una función de comparación basada en el valor de urgencia de cada cita.

#### *mostrarCitas*

```
static void mostrarCitas  
(const std::vector<CitaMedica*>& citas)
```

Propósito: Mostrar todas las citas médicas.

Descripción:

Imprime por terminal una lista de todas las citas contenidas en el vector `citas`. Esto proporciona una visión general de todas las citas en el sistema, facilitando la inspección o verificación manual de las citas.

#### *buscarCitasPorFecha*

```
static void buscarCitasPorFecha(const  
std::vector<CitaMedica*>& citas, const std::string& fecha)
```

Propósito: Buscar citas médicas por una fecha específica.

Descripción:

Filtra y muestra todas las citas que ocurren en una fecha específica dada por el parámetro `fecha`. Compara la fecha de cada cita con la fecha introducida, y muestra las coincidencias. Este método permite a los usuarios consultar las citas que se han programado para un día específico.

#### *buscarCitasPorUrgencia*

```
static void buscarCitasPorUrgencia(const  
std::vector<CitaMedica*>& citas, int urgencia)
```

Propósito: Buscar citas médicas según su nivel de urgencia.

Descripción:

Filtra las citas del sistema y muestra aquellas que tienen el nivel de urgencia especificado. El parámetro `urgencia` debe ser un número entero que representa el nivel de urgencia, y la función mostrará solo aquellas citas que coincidan con ese valor.

### *buscarCitasPorFechaComparada*

```
static void buscarCitasPorFechaComparada (const  
std::vector<CitaMedica*>& citas, const std::string& fecha)
```

Propósito: Buscar citas pasadas o futuras comparadas con una fecha dada.

Descripción:

Permite filtrar y mostrar las citas que ocurren antes o después de una fecha. El parámetro fecha es comparado con la fecha de cada cita y se muestran aquellas que ocurren antes o después de esa fecha.

### *buscarCitasEnIntervalo*

```
static std::vector<CitaMedica*>  
buscarCitasEnIntervalo (const std::vector<CitaMedica*>&  
citas, const std::string& fechaInicio, const std::string&  
fechaFin)
```

Propósito: Buscar citas médicas dentro de un intervalo de fechas.

Descripción:

Filtra las citas en el sistema que caen dentro de un rango de fechas especificado por fechaInicio y fechaFin. La función compara las fechas de las citas con el intervalo proporcionado y devuelve un nuevo vector con las citas que ocurren dentro de ese intervalo. Esto facilita la consulta de citas programadas dentro de un período determinado.

### *listarCitasPendientesPorMedico*

```
static void listarCitasPendientesPorMedico (const  
std::vector<CitaMedica*>& citas, Medico* medico)
```

Propósito: Lista las citas pendientes para un médico específico.

Descripción:

Filtra y muestra todas las citas que están asignadas a un médico determinado y que aún no han sido atendidas. El parámetro medico es un puntero a un objeto de la clase Medico, y la función muestra las citas que están pendientes para ese médico.

### *listarCitasPendientesPorServicio*

```
static void listarCitasPendientesPorServicio (const  
std::vector<CitaMedica*>& citas, const std::string&  
servicio)
```

Propósito: Lista las citas pendientes por un servicio específico.

Descripción:

Muestra todas las citas asignadas a un servicio específico que aún no han sido atendidas.

El parámetro servicio es una cadena de texto que representa el nombre del servicio (por ejemplo, "Cardiología"), y la función filtra y muestra las citas pendientes para ese servicio en particular.

#### *mostrarCita*

```
void mostrarCita() const
```

Propósito: Mostrar los detalles de una cita médica.

Descripción:

Imprime en pantalla la información completa de la cita médica en un formato legible, incluyendo el servicio del médico, la fecha de la cita, el nombre e ID del paciente, el nombre e ID del médico, y el nivel de urgencia.

La función no modifica el estado del objeto, solo muestra sus atributos, y se marca como const para garantizar que no altere el objeto.

#### *registrarCita*

```
static void registrarCita(std::vector<CitaMedica*>& citas,  
    Paciente* paciente, Medico* medico, const std::string&  
        fecha, int urgencia)
```

Propósito: Registra una nueva cita médica en el sistema.

Descripción:

Crea y agrega una nueva cita médica al sistema, utilizando los detalles proporcionados sobre el paciente, médico, fecha y nivel de urgencia. La función agrega la nueva cita al vector citas, permitiendo que se mantenga un registro de todas las citas en el sistema. Después de registrar la cita, se puede acceder a ella mediante las funciones de búsqueda, visualización y modificación.

#### *modificarCita*

```
static void modificarCita(std::vector<Paciente*>& pacientes,  
    std::vector<Medico*>& medicos, std::vector<CitaMedica*>& citas)
```

Propósito: Permite modificar los detalles de una cita médica existente.

Descripción:

Solicita al usuario la fecha de la cita que desea modificar. Después de validar la fecha proporcionada, la función busca la cita correspondiente. Permite al usuario modificar varios aspectos de la cita, como el paciente asignado, el médico, la urgencia y la fecha. Una vez que se validan todos los cambios, la cita es actualizada con los nuevos datos.

#### *eliminarCita*

```
static void eliminarCita(std::vector<CitaMedica*>& citas)
```

Propósito: Elimina una cita médica del sistema.

Descripción:

Solicita al usuario la fecha de la cita que desea eliminar. Muestra todas las citas correspondientes a esa fecha. Permite seleccionar una cita para eliminar basándose en el ID del paciente. Solicita confirmación antes de proceder con la eliminación de la cita seleccionada. Una vez confirmada la eliminación, la cita es eliminada del sistema.

### Funciones y métodos llamados desde otros archivos

A continuación, se enumeran las funciones y métodos llamados desde otros archivos en citaMedica.h, organizados según su origen. Todos se encuentran dentro de la clase CitaMedica{}

#### *Paciente*

**Paciente::verPacientePorID:** Muestra la información de un paciente según su ID.

**Paciente::eliminarPaciente:** Elimina un paciente del sistema.

#### *Medico*

**Medico::verMedicoPorID:** Muestra la información de un médico según su ID.

**Medico::eliminarMedico:** Elimina un médico del sistema.

### Explicación detallada de las decisiones de diseño

#### *Modularidad*

El código está diseñado de manera modular, donde cada función realiza una tarea específica. Esto hace que el código sea fácil de mantener y extender, permitiendo agregar nuevas funcionalidades de forma sencilla.

### *Funciones Estáticas*

Se ha optado por definir las funciones de `citaMedica.h` como funciones estáticas para facilitar su uso sin necesidad de crear instancias de la clase `CitaMedica`. Esto simplifica el acceso a las funciones desde cualquier parte del programa, evitando la creación de objetos innecesarios.

### *Estructura de Datos*

Las citas se almacenan en un vector de punteros (`std::vector<CitaMedica*>`), lo que permite una gestión flexible y eficiente de las citas médicas, permitiendo realizar búsquedas y ordenaciones rápidas. Este enfoque también facilita la eliminación y modificación de citas sin afectar a la estructura de datos subyacente.

### *Interacciones entre clases*

Las funciones de `CitaMedica` interactúan de forma estrecha con las clases `Paciente` y `Medico` para asociar citas a pacientes y médicos. Este diseño permite una gestión centralizada de las citas médicas, manteniendo la lógica de negocio en el archivo `citaMedica.h` mientras delega las responsabilidades específicas a las clases `Paciente` y `Medico`. El uso de punteros permite modificar directamente las instancias de `Paciente` y `Medico` cuando se realizan cambios en las citas.

### *Validación de entradas*

La validación de fechas y otros parámetros importantes se realiza en las funciones de `citaMedica.h` para asegurar que los datos ingresados sean correctos y consistentes.

### *Interacción con Archivos*

Las funciones de la clase `GestorArchivos` se usan para cargar y guardar las citas médicas desde y hacia archivos. Esta funcionalidad permite que las citas médicas se persistan entre ejecuciones del programa, evitando la pérdida de datos.

Este enfoque modular y el uso de punteros, funciones estáticas y validación robusta permiten un control eficiente y flexible sobre las citas médicas, así como una integración fluida con las demás partes del sistema, como los pacientes, médicos y archivos.

## **gestorArchivos.h**

El archivo `gestorArchivos.h` define la clase `GestorArchivos`, encargada de gestionar la lectura y escritura de datos relacionados con pacientes, médicos y citas médicas. Permite cargar y guardar información desde y hacia archivos, facilitando la persistencia de los datos en el sistema.

## Listado de funciones en la clase GestorArchivos

### *respaldarArchivo*

```
void respaldarArchivo(const std::string& archivoOriginal)
```

Propósito: Realiza un respaldo de un archivo específico.

Descripción:

La función crea una copia del archivo original en un directorio de respaldo (backup/), generando un nombre de archivo único con la fecha y hora actuales. Si el directorio de respaldo no existe, lo crea. Se utiliza la librería `std::filesystem` para la manipulación de archivos y directorios.

### *guardarDatosPacientes*

```
void guardarDatosPacientes  
(const std::vector<Paciente*>& pacientes)
```

Propósito: Guarda los datos de los pacientes en un archivo de texto (pacientes.txt).

Descripción:

La función guarda la información de cada paciente (nombre, ID, fecha de ingreso) en un archivo de texto. Además, realiza un respaldo del archivo antes de escribir los nuevos datos.

### *recuperarDatosPacientes*

```
void recuperarDatosPacientes(std::vector<Paciente*>&  
pacientes)
```

Propósito: Recupera los datos de los pacientes desde un archivo de texto (pacientes.txt).

Descripción:

Lee el archivo `pacientes.txt` y extrae los datos de los pacientes (nombre, ID, fecha de ingreso). Los pacientes recuperados se almacenan en el vector proporcionado.

### *guardarDatosMedicos*

```
void guardarDatosMedicos(const std::vector<Medico*>&  
medicos)
```

Propósito: Guarda los datos de los médicos en un archivo de texto (medicos.txt).

Descripción:

Similar a la función `guardarDatosPacientes`, pero para médicos. Guarda el nombre, ID, servicio y disponibilidad de cada médico en el archivo `medicos.txt`.

### *recuperarDatosMedicos*

```
void recuperarDatosMedicos (std::vector<Medico*>& medicos)
```

Propósito: Recupera los datos de los médicos desde un archivo de texto (medicos.txt).

Descripción:

Lee el archivo medicos.txt y extrae los datos de los médicos (nombre, ID, servicio, disponibilidad). Los médicos recuperados se almacenan en el vector proporcionado.

### *guardarDatosCitas*

```
void guardarDatosCitas (const std::vector<CitaMedica*>&  
                        citas)
```

Propósito: Guarda los datos de las citas médicas en un archivo de texto (citas.txt).

Descripción:

Guarda la fecha, paciente, médico y urgencia de cada cita médica en el archivo citas.txt.

### *recuperarDatosCitas*

```
void recuperarDatosCitas (std::vector<CitaMedica*>& citas,  
                        const std::vector<Paciente*>& pacientes, const  
                        std::vector<Medico*>& medicos)
```

Propósito: Recupera los datos de las citas médicas desde un archivo de texto (citas.txt).

Descripción:

Lee el archivo citas.txt, extrae la información de cada cita (fecha, ID del paciente, ID del médico y urgencia), y busca los pacientes y médicos correspondientes a través de sus ID. Luego, crea las citas médicas y las agrega al vector proporcionado.

### *guardarEnArchivo*

```
void guardarEnArchivo (const std::string& nombrePaciente,  
                      bool enfermedadCronica)
```

Propósito: Guardar la información de un paciente con enfermedad crónica en un archivo.

Descripción:

Esta función primero respalda el archivo de pacientes crónicos, luego abre un archivo para guardar el nombre del paciente y su estado de enfermedad crónica (sí o no). Si se puede abrir, escribe la información del paciente en el archivo y lo cierra.

## Funciones y métodos llamados desde otros archivos

A continuación, se enumeran las Funciones y métodos llamados desde otros archivos en el archivo gestorArchivos.h, organizados según su origen.

### *Paciente.h*

**Paciente::buscarPacientePorID:** Para buscar un paciente en el sistema mediante su ID.

### *medico.h*

**Medico::buscarMedicoPorID:** Para buscar un médico en el sistema mediante su ID.

## Explicación detallada de las decisiones de diseño

### *Modularidad*

El archivo gestorArchivos.h es responsable de las operaciones de entrada y salida (E/S) de archivos. Estas funciones son modulares y se centran en una tarea específica: guardar y recuperar datos de pacientes, médicos y citas médicas. La modularidad permite una fácil extensión en caso de que se necesiten más tipos de datos para guardar o cargar.

### *Uso de std::filesystem*

Se utiliza std::filesystem para manejar la creación del directorio de respaldo y la copia de archivos. Esta biblioteca moderna de C++ facilita las operaciones de entrada y salida de archivos, especialmente para gestionar directorios y realizar copias de seguridad sin tener que preocuparse por las plataformas de bajo nivel.

### *RespalDOS automáticos*

Antes de guardar cualquier dato de pacientes, médicos o citas, se realiza un respaldo automático del archivo correspondiente para evitar la pérdida de información. Esto mejora la fiabilidad y seguridad del sistema.

### *Funciones de entrada/salida*

Las funciones de guardado y recuperación de datos (guardarDatosPacientes, recuperarDatosPacientes, etc.) utilizan flujos de archivos estándar en modo binario o de texto. Esto permite que los datos se guarden de manera eficiente y luego se puedan leer para su posterior uso.

### *Interacciones entre clases*

El GestorArchivos interactúa principalmente con las clases Paciente, Medico y CitaMedica. Se utilizan punteros a estas clases para guardar o recuperar la información de los archivos de manera eficiente.



### *Validación de entrada*

Aunque no se realiza una validación compleja en estas funciones de archivos, se asume que los datos en los archivos son correctos y que el formato de los archivos es consistente. Sin embargo, se incluyen mensajes de error si no se pueden abrir los archivos, lo que proporciona retroalimentación al usuario.

Este diseño modular, que delega las responsabilidades de manipulación de archivos a un gestor especializado, permite una fácil expansión y mantenimiento del código sin sobrecargar otras partes del programa.

## **medico.h**

El archivo `medico.h` define la clase `Medico`, que gestiona la información relacionada con los médicos en el sistema. Proporciona métodos para registrar, editar, eliminar y consultar los datos de los médicos, como su nombre, ID, servicio y disponibilidad. Esta clase facilita la organización y manejo de los médicos en el sistema, permitiendo asociar a cada médico un servicio o especialidad, y verificar su disponibilidad para atender a los pacientes.

## **Clase Medico**

### *Atributos*

- `nombre (std::string)`: El nombre del médico.
- `ID (int)`: El ID único del médico.
- `servicio (std::string)`: El servicio al que pertenece el médico.
- `disponibilidad (bool)`: Indica si el médico está disponible.

### *Getters*

#### `getNombre()`

Tipo de retorno: `std::string`

Descripción: Retorna el nombre del médico almacenado en el atributo `nombre`.

#### `getID()`

Tipo de retorno: `int`

Descripción: Retorna el identificador único del médico almacenado en el atributo `ID`.

#### `getServicio()`

Tipo de retorno: `std::string`

Descripción: Retorna el servicio o especialidad del médico almacenado en el atributo `servicio`.

*getDisponibilidad()*

Tipo de retorno: bool

Descripción: Retorna el estado de disponibilidad del médico almacenado en el atributo disponibilidad.

## Listado de funciones en la clase Medico

### *Constructor Medico*

```
Medico(std::string& nombre, int ID, const std::string&
servicio, bool disponibilidad)
```

Propósito: Inicializa un objeto Medico con los valores proporcionados.

Descripción:

Este constructor permite crear un objeto Medico a partir de un nombre, un ID, un servicio y la disponibilidad del médico. Utiliza estos parámetros para configurar las propiedades del médico al momento de su creación.

### *setNombre*

```
void setNombre(const std::string& nombre)
```

Propósito: Establece el nombre del médico.

Descripción:

Este método establece el valor del atributo nombre del médico.

### *setID*

```
void setID(const int& ID)
```

Propósito: Establece el ID del médico.

Descripción:

Este método establece el valor del atributo ID del médico.

### *setServicio*

```
void setServicio(const std::string& servicio)
```

Propósito: Establece el servicio del médico si es válido.

Descripción:

Este método asigna un servicio al médico, pero antes verifica que el servicio esté presente en el archivo servicios.txt. Si el servicio no es válido, se muestra un mensaje de error.

### *setDisponibilidad*

```
void setDisponibilidad(const bool& disponibilidad)
```

Propósito: Establece la disponibilidad del médico.

Descripción:

Este método asigna un valor booleano al atributo disponibilidad, indicando si el médico está disponible o no.

*esServicioValido*

```
bool esServicioValido(const std::string& servicio)
```

Propósito: Verifica si un servicio es válido.

Descripción:

Esta función abre el archivo servicios.txt y verifica si el servicio proporcionado se encuentra en el archivo, ignorando las diferencias entre mayúsculas y minúsculas.

*verMedicoPorServicio*

```
static void verMedicoPorServicio  
(const std::vector<Medico*>& medicos)
```

Propósito: Muestra los médicos disponibles en un servicio específico.

Descripción:

La función solicita al usuario que ingrese un servicio, valida que sea correcto y luego muestra todos los médicos que pertenecen a ese servicio.

*verMedicoPorID*

```
static void verMedicoPorID  
(const std::vector<Medico*>& medicos)
```

Propósito: Muestra los detalles de un médico basado en su ID.

Descripción:

Solicita al usuario un ID, valida que sea un número y busca el médico correspondiente. Si se encuentra, muestra su información.

*verMedicoPorNombre*

```
static void verMedicoPorNombre  
(const std::vector<Medico*>& medicos)
```

Propósito: Muestra los detalles de un médico basado en su nombre.

Descripción:

Permite al usuario buscar médicos por nombre, y muestra sus detalles si se encuentra alguno que coincida.

#### *verMedicoPorDisponibilidad*

```
static void verMedicoPorDisponibilidad  
(const std::vector<Medico*>& medicos)
```

Propósito: Muestra los médicos según su disponibilidad.

Descripción:

Solicita al usuario la disponibilidad del médico (1 para disponible, 0 para no disponible), y muestra los médicos que coinciden con esa disponibilidad.

#### *mostrarMedico*

```
void mostrarMedico() const
```

Propósito: Muestra los detalles del médico.

Descripción:

Imprime la información completa del médico, incluyendo su nombre, ID, servicio y disponibilidad, formateada (con setw y setfill) para su correcta presentación en la consola.

#### *registrarMedico*

```
void registrarMedico()
```

Propósito: Permite registrar un nuevo médico en el sistema.

Descripción:

Solicita al usuario los datos del médico, realiza validaciones y luego lo agrega al sistema. Este método incluye la verificación de la validez del servicio y la disponibilidad.

#### *modificarMedico*

```
void modificarMedico(std::vector<Medico*>& medicos)
```

Propósito: Permite modificar los datos de un médico existente.

Descripción:

Solicita el ID del médico, busca el médico en la lista y permite modificar sus datos, incluyendo el nombre, el ID, el servicio y la disponibilidad.

#### *eliminarMedico*

```
static void eliminarMedico(std::vector<Medico*>& medicos)
```

Propósito: Elimina un médico del sistema según su ID.

Descripción:

Solicita el ID del médico a eliminar, verifica si existe y lo elimina de la lista de médicos. Además, pide confirmación antes de realizar la eliminación.

## Funciones y métodos llamados desde otros archivos

A continuación, se enumeran las Funciones y métodos llamados desde otros archivos en el archivo `medico.h`, organizados según su origen.

### *Paciente.h*

**Paciente::buscarPacientePorID**

Utilizado para buscar un paciente en el sistema mediante su ID.

### *citaMedica.h*

**CitaMedica::buscarCitaPorID**

Usado para asociar un médico a una cita médica en el sistema.

## Explicación detallada de las decisiones de diseño

### *Modularidad*

La clase `Medico` sigue un enfoque modular, donde cada función tiene una responsabilidad específica, lo que facilita la extensión y el mantenimiento del código. Al estar diseñada de manera independiente, se puede agregar fácilmente nueva información relacionada con médicos sin afectar otras áreas del programa.

### *Funciones Estáticas*

En la clase `Medico`, se ha optado por utilizar funciones estáticas para simplificar el acceso a las operaciones más comunes de la clase sin necesidad de crear instancias. Esto permite realizar operaciones, como la búsqueda de médicos por ID o la validación de la disponibilidad, directamente desde cualquier parte del código, sin crear objetos adicionales innecesarios.

### *Estructura de Datos*

Los médicos se almacenan en un vector de punteros (`std::vector<Medico*>`), lo que permite una gestión eficiente de los datos. Este enfoque facilita la búsqueda de médicos por ID, y también permite realizar modificaciones o eliminaciones sin afectar a la estructura subyacente. Además, se aprovechan las ventajas de la gestión dinámica de memoria al trabajar con punteros.

### *Interacciones entre clases*

Las funciones de la clase `Medico` interactúan estrechamente con otras clases del sistema, como `Paciente` y `CitaMedica`. La interacción con `Paciente` se realiza a través de la gestión de citas médicas, donde un médico puede estar asociado a múltiples citas. Mediante punteros, se mantiene una referencia eficiente a las

instancias de Paciente y CitaMedica, lo que facilita la actualización de los datos cuando sea necesario.

### *Validación de entradas*

En la clase Medico, se validan las entradas de datos relacionadas con el ID, la disponibilidad y el servicio del médico. La validación asegura que los datos sean correctos y consistentes antes de proceder con su almacenamiento o modificación, lo que mejora la robustez del sistema.

### *Interacción con Archivos*

Las funciones de GestorArchivos interactúan con la clase Medico para guardar y recuperar los datos de los médicos desde archivos, asegurando que la información se persista entre ejecuciones del programa. Esto garantiza que los cambios en la disponibilidad o los datos de los médicos sean reflejados en los archivos de forma confiable.

## **paciente.h**

El archivo paciente.h define la clase Paciente, que gestiona la información de los pacientes en el sistema. Proporciona métodos para registrar, editar, eliminar y consultar los datos de los pacientes, como su nombre e ID. Además, permite manejar aspectos relacionados con las citas médicas y enfermedades crónicas asociadas a cada paciente.

## **Clase Paciente**

### *Atributos*

- nombre (std::string): El nombre completo del paciente.
- ID (int): Identificador único del paciente.
- fechaIngreso (std::string): Fecha de ingreso del paciente al sistema en formato dd-MM-AAAA.

### *Getters*

#### **getNombre()**

Tipo de retorno: std::string

Descripción: Retorna el nombre del paciente almacenado en el atributo nombre.

#### **getID()**

Tipo de retorno: int

Descripción: Retorna el identificador único del paciente almacenado en el atributo ID.

*getFechaIngreso()*

Tipo de retorno: `std::string`

Descripción: Retorna la fecha de ingreso del paciente almacenada en el atributo `fechaIngreso`.

## Listado de funciones en la clase Paciente

*esFechaValida*

```
static bool esFechaValida(const std::string& fecha)
```

Propósito: Validar que una fecha esté en el formato correcto DD-MM-AAAA y que sea una fecha válida.

Descripción:

Comprueba que la fecha tenga el formato DD-MM-AAAA verificando los caracteres en las posiciones correctas.

*Constructor Paciente*

```
Paciente(const std::string& nombre, int ID, const  
std::string& fechaIngreso)
```

Propósito: Inicializa un objeto Paciente con los valores proporcionados.

Descripción:

Este constructor asigna un nombre, un ID y una fecha de ingreso al paciente al momento de su creación.

*setNombre*

```
void setNombre(const std::string& nombre)
```

Propósito: Establece el nombre del paciente.

Descripción:

Este método asigna un valor al atributo Nombre del paciente.

*setID*

```
void setID(const int& ID)
```

Propósito: Establece el ID del paciente.

Descripción:

Este método asigna un valor al atributo ID del paciente, asegurando que sea único.

### *setFechaIngreso*

```
void setFechaIngreso(const std::string& fechaIngreso)
```

Propósito: Establece la fecha de ingreso del paciente si es válida.

Descripción:

Este método valida y asigna una fecha al atributo FechaIngreso, asegurándose de que siga el formato dd-MM-AAAA.

### *verPacientePorFecha*

```
static void verPacientePorFecha(const  
std::vector<Paciente*>& pacientes, const std::string&  
fechaIngreso)
```

Propósito: Muestra pacientes ingresados en una fecha específica.

Descripción:

Solicita una fecha y lista los pacientes cuyo atributo FechaIngreso coincida con la fecha proporcionada.

### *verPacientePorID*

```
static void verPacientePorID  
(const std::vector<Paciente*>& pacientes)
```

Propósito: Busca y muestra los detalles de un paciente según su ID.

Descripción:

Solicita un ID, valida la entrada y muestra la información del paciente si se encuentra en la lista.

### *verPacientePorNombre*

```
static void verPacientePorNombre(const std::vector<Paciente*>&  
pacientes)
```

Propósito: Muestra pacientes basándose en su nombre.

Descripción:

Solicita un nombre al usuario, busca coincidencias y muestra los detalles de los pacientes encontrados.

### *mostrarPaciente*

```
void mostrarPaciente() const
```

Propósito: Muestra los detalles completos del paciente.

Descripción:



Imprime la información del paciente en la terminal, incluyendo nombre, ID y fecha de ingreso.

#### *editarPaciente*

```
void editarPaciente(std::vector<Paciente*>& pacientes)
```

Propósito: Permite modificar los datos de un paciente existente.

Descripción:

Solicita el ID de un paciente, busca el registro en el sistema y permite editar su información.

#### *eliminarPaciente*

```
static void eliminarPaciente(std::vector<Paciente*>&  
                             pacientes)
```

Propósito: Elimina un paciente del sistema.

Descripción:

Solicita el ID del paciente, verifica su existencia y elimina el registro correspondiente.

#### *registrarPaciente*

```
void registrarPaciente()
```

Propósito: Permite registrar un nuevo paciente en el sistema.

Descripción:

Solicita al usuario los datos del paciente, valida las entradas y los agrega al sistema.

## Explicación detallada de las decisiones de diseño

### *Modularidad*

La clase Paciente está diseñada con un enfoque modular que distribuye las responsabilidades en funciones específicas. Esto permite mantener un código organizado y fácilmente extensible. Cada funcionalidad, como registrar, editar, o buscar pacientes, se implementa como un método independiente, lo que facilita la localización y el mantenimiento del código.

### *Funciones estáticas*

Las funciones estáticas se utilizan para realizar operaciones comunes sin necesidad de instanciar objetos de la clase Paciente. Este diseño reduce la complejidad y mejora la reutilización de las funciones desde cualquier parte del programa.

### *Estructura de Datos*

Los pacientes se gestionan mediante un `std::vector` de punteros (`std::vector<Paciente*>`). Este enfoque permite almacenar dinámicamente los objetos de la clase `Paciente`, facilitando la adición, eliminación y búsqueda eficiente de pacientes. Además, esta estructura se integra de manera fluida con otras funcionalidades del sistema.

### *Interacciones entre clases*

La clase `Paciente` interactúa directamente con `CitaMedica` para asociar y gestionar las citas de cada paciente. Estas interacciones están diseñadas mediante punteros, lo que permite mantener referencias a las citas asociadas sin duplicar datos, asegurando la coherencia y la eficiencia en el manejo de las relaciones entre clases.

### *Validación de Entradas*

La clase implementa métodos de validación para garantizar que los datos ingresados, como fechas, IDs y nombres, sean correctos y consistentes antes de procesarlos. Por ejemplo, el método `esFechaValida` verifica el formato y la validez de las fechas ingresadas, reduciendo la posibilidad de errores durante la ejecución.

### *Interacción con Archivos*

Las operaciones de lectura y escritura de pacientes desde y hacia archivos se delegan a funciones externas, integrándose de manera transparente con el sistema. Esto permite persistir la información de los pacientes entre ejecuciones del programa y asegura que los cambios realizados en el sistema se reflejen en los datos almacenados.

## **reporte.h**

El archivo `reporte.h` define la clase `Reporte`, que centraliza funciones relacionadas con la generación de reportes, consulta de historial clínico y manejo de enfermedades crónicas.

### Listado de funciones en `reporte.h`

#### *listarCitasPendientes*

```
void listarCitasPendientes(const std::vector<CitaMedica*>&  
                           citas, const std::string& criterio)
```

Propósito: Mostrar las citas pendientes agrupadas por médico o servicio.

Descripción:

Recorre un vector de citas médicas y muestra información de las citas pendientes según el criterio especificado ("medico" o "servicio").

### *reportePacientesCronicos*

```
void reportePacientesCronicos  
(const std::vector<Paciente*>& pacientes)
```

Propósito: Generar un reporte de pacientes con enfermedades crónicas.

Descripción:

Recorre el vector de pacientes para identificar aquellos con enfermedades crónicas y muestra sus nombres e identificadores.

### *esCitaPasada*

```
bool esCitaPasada(const std::string& fechaCita, int year,  
                  int month, int day)
```

Propósito: Verificar si una cita pertenece al pasado.

Descripción:

Compara la fecha de una cita con la fecha actual y devuelve true si la cita ya ocurrió.

### *verHistorialClinico*

```
void verHistorialClinico(std::vector<CitaMedica*>& citas)
```

Propósito: Consultar el historial clínico de un paciente o de todos los pacientes.

Descripción:

Ordena las citas por fecha y muestra aquellas pasadas, filtrando por el identificador del paciente o mostrando todas si no se especifica un ID.

### *modificarEnfermedadCronica*

```
void modificarEnfermedadCronica  
(const std::vector<Paciente*>& pacientes)
```

Propósito: Registrar o modificar el estado de enfermedad crónica de un paciente.

Descripción:

Permite buscar un paciente por nombre y actualizar su estado de enfermedad crónica, mostrando un mensaje de confirmación.

## Funciones privadas

### *tieneEnfermedadCronica*

```
bool tieneEnfermedadCronica(Paciente* paciente)
```

Propósito: Determinar si un paciente tiene una enfermedad crónica.

Descripción:

Devuelve un valor booleano indicando si el paciente posee enfermedades crónicas. Esta implementada como una función auxiliar dentro de la clase.

## Funciones y métodos llamados desde otros archivos

A continuación, se enumeran las Funciones y métodos llamados desde otros archivos en el archivo `reporte.h`, organizados según su origen.

### *Archivo `citaMedica.h`*

**CitaMedica::getFecha:** Obtiene la fecha de una cita médica.

**CitaMedica::getPaciente:**

Devuelve un puntero al paciente asociado a la cita.

**CitaMedica::getMedico:** Devuelve un puntero al médico asociado a la cita.

### *Archivo `paciente.h`*

**Paciente::getNombre:** Obtiene el nombre del paciente.

**Paciente::getID:** Devuelve el identificador único del paciente.

### *Archivo `medico.h`*

**Medico::getNombre:** Devuelve el nombre del médico.

**Medico::getServicio:** Obtiene el servicio asociado al médico.

## Explicación detallada de las decisiones de diseño

### *Modularidad*

Se encapsularon las funciones relacionadas con reportes y consultas en la clase `Reporte` para mantener una estructura modular y facilitar el mantenimiento del código.

### *Uso de vectores de punteros*

Se usan vectores de punteros para facilitar el manejo dinámico de objetos, permitiendo modificaciones, búsquedas y almacenamiento eficiente de datos relacionados con pacientes, médicos y citas.

### *Validación centralizada*

La función auxiliar `esCitaPasada` centraliza la lógica de validación de fechas, evitando redundancia en otras funciones.

### *Interactividad*

Funciones como `modificarEnfermedadCronica` y `verHistorialClinico` permiten la interacción directa con el usuario, aumentando la flexibilidad en el manejo de datos.

## main.cpp

El archivo main.cpp actúa como el controlador principal del programa, donde se gestionan las interacciones del usuario y la ejecución de las funcionalidades del sistema. Contiene el flujo principal de ejecución, incluyendo la carga de datos, la visualización de menús, y la captura de entradas del usuario. Además, delega las tareas específicas a otras clases como CitaMedica, Paciente, Medico y GestorArchivos, manteniendo el código organizado y modular.

### Listado de funciones en main.cpp

A continuación, se enumeran las funciones definidas en main.cpp.

#### *leerEntero*

```
int leerEntero (const std::string& mensaje)
```

Propósito: Solicitar al usuario que ingrese un número entero, validando la entrada.

Descripción:

Muestra un mensaje al usuario. Valida que la entrada sea un número entero y solicita reintentos en caso de entrada inválida.

#### *buscarPacientePorID*

```
Paciente* buscarPacientePorID (  
const std::vector<Paciente*>& pacientes, int id)
```

Propósito: Buscar un paciente en el vector de pacientes a partir de su ID.

Descripción:

Recorre el vector de punteros a pacientes (std::vector<Paciente\*>) y compara cada ID con el ID proporcionado. Si encuentra un paciente cuyo ID coincide con el proporcionado, devuelve un puntero a ese paciente.

#### *buscarMedicoPorID*

```
Medico* buscarMedicoPorID  
(const std::vector<Medico*>& medicos, int id)
```

Propósito: Buscar un médico en el vector de médicos a partir de su ID.

Descripción:

Recorre el vector de punteros a médicos (std::vector<Medico\*>) y compara cada ID con el ID proporcionado. Si encuentra un médico cuyo ID coincide con el proporcionado, devuelve un puntero a ese médico.

## Funciones y métodos llamados desde otros archivos

A continuación, se enumeran las Funciones y métodos llamados desde otros archivos en el archivo main.cpp, organizados según su origen.

### *citaMedica.h*

**CitaMedica::buscarCitasEnIntervalo:**

Busca citas médicas en un intervalo de fechas.

**CitaMedica::modificarCita:** Permite editar una cita médica.

**CitaMedica::eliminarCita:** Elimina una cita médica del sistema.

**CitaMedica::ordenarPorFecha:** Ordena las citas por fecha.

**CitaMedica::ordenarPorUrgencia:** Ordena las citas por urgencia.

**CitaMedica::mostrarCitas:** Muestra una lista de citas médicas.

**CitaMedica::listarCitasPendientesPorMedico:** Lista citas pendientes para un médico específico.

**CitaMedica::listarCitasPendientesPorServicio:** Lista citas pendientes por servicio.

**CitaMedica::buscarCitasPorFecha:** Busca citas médicas por fecha específica.

**CitaMedica::buscarCitasPorUrgencia:** Busca citas médicas según su urgencia.

**CitaMedica::buscarCitasPorFechaComparada:** Busca citas pasadas o futuras según una fecha.

**CitaMedica::registrarCita:** Registra una nueva cita médica.

### *Archivo gestorArchivos.h*

**GestorArchivos::recuperarDatosPacientes:** Carga los datos de los pacientes desde un archivo.

**GestorArchivos::recuperarDatosMedicos:** Carga los datos de los médicos desde un archivo.

**GestorArchivos::recuperarDatosCitas:** Carga los datos de las citas médicas desde un archivo.

**GestorArchivos::guardarDatosPacientes:** Guarda los datos de los pacientes en un archivo.

**GestorArchivos::guardarDatosMedicos:** Guarda los datos de los médicos en un archivo.

**GestorArchivos::guardarDatosCitas:** Guarda los datos de las citas médicas en un archivo.

**GestorArchivos::guardarEnArchivo:** Guarda información adicional, como el estado de enfermedades crónicas.

#### *Archivo paciente.h*

**Paciente::verPacientePorFecha:** Muestra pacientes registrados en una fecha específica.

**Paciente::verPacientePorID:** Muestra información de un paciente basado en su ID.

**Paciente::verPacientePorNombre:** Busca pacientes por nombre.

**Paciente::eliminarPaciente:** Elimina un paciente del sistema.

**Paciente::registrarPaciente:** Registra un nuevo paciente.

**Paciente::editarPaciente:** Permite modificar la información de un paciente.

#### *Archivo medico.h*

**Medico::verMedicoPorServicio:** Lista médicos que ofrecen un servicio específico.

**Medico::verMedicoPorID:** Busca médicos por ID.

**Medico::verMedicoPorNombre:** Busca médicos por nombre.

**Medico::verMedicoPorDisponibilidad:** Lista médicos disponibles en un horario.

**Medico::eliminarMedico:** Elimina un médico del sistema.

**Medico::registrarMedico:** Registra un nuevo médico.

**Medico::modificarMedico:** Permite editar la información de un médico.

#### *Archivo reporte.h*

**Reporte::verHistorialClinico:** Muestra el historial clínico de un paciente.

**Reporte::modificarEnfermedadCronica:** Permite registrar o modificar una enfermedad crónica para un paciente.

## Explicación detallada de las decisiones de diseño

### *Modularidad*

El archivo `main.cpp` se utiliza como controlador principal, mientras que las implementaciones específicas de lógica se delegan a clases externas. Esto mantiene el archivo principal limpio y fácil de leer.

### *Uso de vectores de punteros*

Los vectores de punteros (`std::vector<Paciente*>`, `std::vector<Medico*>`, etc.) permiten gestionar objetos dinámicamente, facilitando la creación, modificación y eliminación de elementos.

### *Gestión de memoria*

Se emplea memoria dinámica para crear instancias de pacientes, médicos y citas, asegurando que se liberen al final de la ejecución para evitar fugas de memoria.

### *Validación de entrada*

La función `leerEntero` centraliza la validación de entradas numéricas, reduciendo redundancias y mejorando la robustez del sistema.

### *Menús anidados*

Los menús están organizados jerárquicamente para una navegación clara y estructura lógica en las opciones. La validación de entrada en cada menú asegura robustez frente a datos incorrectos.

### *Uso de clases específicas*

Clases como `Paciente`, `Medico` y `CitaMedica` encapsulan la lógica relacionada con sus respectivas entidades, promoviendo un diseño orientado a objetos.

### *Flexibilidad en el manejo de datos*

Las funciones de búsqueda y modificación de datos permiten una interacción dinámica con los registros, adaptándose a diferentes criterios de filtrado.

## Archivos.txt

La carpeta `output` y los archivos `.txt` que contiene (excepto `servicios.txt`) almacenan los datos gestionados por las funciones de `gestorArchivos.h`. A continuación, se describe el propósito de cada archivo y se proporciona un ejemplo de los datos almacenados.

### `pacientes.txt`

Almacena la información de los pacientes registrados en el sistema.

Ejemplo de guardado:

Nombre: Juan Perez



ID: 1234

Fecha de ingreso: 01-01-2025

### `medicos.txt`

Almacena la información de los médicos registrados en el sistema.

Ejemplo de guardado:

Dr. Ana Garcia

ID: 0001

Servicio: Cardiología

Disponibilidad: 1

### `citas.txt`

Almacena la información de las citas médicas registradas en el sistema.

Ejemplo de guardado:

Fecha: 20-01-2025

Paciente: Juan Perez (ID: 1234)

Medico: Ana Garcia (ID: 0001)

Urgencia: 3

### `pacientes_cronicos.txt`

Almacena la información de los pacientes con enfermedades crónicas.

Ejemplo de guardado:

Nombre: Juan Perez

Enfermedad cronica: Si

### `servicios.txt`

Este archivo .txt no guardara nuevos datos, sino que tiene unos datos constantes e inmodificables que sirven para verificar si un servicio existe en el sistema.

### `carpeta backup`

Almacena archivos de backup en caso de pérdida de datos. Estos archivos backup se generan en `gestorArchivos.h` individualmente por cada función de guardado en un .txt. Por ejemplo, cuando se guardan los datos de un paciente modificado en su .txt, también se genera un archivo backup con la fecha y hora de guardado.

# Algoritmos de Búsqueda y Ordenación

## Búsqueda lineal

El algoritmo principal utilizado en el proyecto para encontrar elementos en los vectores de pacientes, médicos o citas es la búsqueda lineal. Se recorre el vector y se compara cada elemento con el criterio de búsqueda hasta encontrar el resultado o llegar al final del vector. A continuación, se muestran algunos casos de uso del algoritmo de búsqueda lineal.

### Búsqueda de pacientes por ID en main.cpp:

La función `buscarPacientePorID` recorre el vector de pacientes y compara cada ID con el proporcionado para encontrar al paciente correspondiente. Este es un caso típico de búsqueda lineal, ya que el código no utiliza estructuras de datos optimizadas para búsquedas rápidas (como árboles o tablas hash).

### Búsqueda de médicos por ID en main.cpp:

De manera similar a la búsqueda de pacientes, la función `buscarMedicoPorID` recorre el vector de médicos y compara los ID para encontrar el médico correspondiente.

### Búsqueda de citas médicas por fecha o urgencia en CitaMedica:

Aunque el código no proporciona un algoritmo específico de búsqueda para las citas, se pueden utilizar búsquedas lineales al recorrer el vector de citas y comparar cada cita con un valor específico (como una fecha o urgencia). Esto también puede considerarse una forma de búsqueda lineal.

## Posibles optimizaciones

Si el proyecto llegara a crecer y manejar una gran cantidad de datos, se podría considerar el uso de algoritmos de búsqueda más eficientes como búsqueda binaria o estructuras de datos como mapas o árboles de búsqueda binaria, pero por el momento parece que la búsqueda lineal es suficiente.

## Bubble Sort

El algoritmo principal utilizado en el proyecto para ordenar las citas médicas es el Bubble Sort. Este algoritmo recorre el vector de citas y compara elementos adyacentes, intercambiándolos si están en el orden incorrecto, hasta que todos los elementos estén ordenados de acuerdo con el criterio definido. A continuación, se muestran algunos casos de uso del algoritmo de ordenación por burbuja.

### Ordenación de citas por fecha en CitaMedica:

La función `ordenarPorFecha` recorre el vector de citas médicas y compara las fechas de las citas, desglosándolas en día, mes y año para determinar el orden correcto. Si una cita tiene una fecha anterior a la de la cita siguiente, las citas se intercambian.

### Ordenación de citas por urgencia en CitaMedica:

La función `ordenarPorUrgencia` compara los niveles de urgencia de las citas médicas en el vector y, si una cita tiene un nivel de urgencia menor que la siguiente, las dos citas se intercambian.

### Ordenación de citas en el historial clínico (`verHistorialClinico`):

En la función `verHistorialClinico`, también se utiliza el algoritmo de Bubble Sort para ordenar las citas por fecha antes de mostrar el historial de un paciente. Se recorre el vector de citas y se ordenan según la fecha, utilizando el mismo procedimiento de comparación por día, mes y año, garantizando que las citas se muestren en orden cronológico.

### Posibles optimizaciones

Aunque el algoritmo de Bubble Sort es sencillo de implementar y entender, no es el más eficiente para grandes volúmenes de datos debido a su complejidad de  $O(n^2)$ . Si el proyecto llegara a manejar una cantidad considerable de citas médicas, se podrían considerar algoritmos de ordenación más eficientes, como Quick Sort o Merge Sort, que tienen una complejidad promedio de  $O(n \log n)$  y son mucho más rápidos para conjuntos de datos grandes. Sin embargo, para un número moderado de citas, Bubble Sort sigue siendo suficiente, manteniendo la simplicidad y claridad del código.

## Conclusiones

Como conclusión, C++ es un lenguaje que permite una gestión eficiente de memoria y la implementación de estructuras de datos complejas, lo cual es esencial para un sistema de gestión hospitalaria. Con relación al desarrollo del proyecto, se ha buscado una separación de archivos firme para facilitar la organización del código. Además, desde un principio se planteó el programa entorno al archivo `main.cpp`, ya que tener un solo archivo ejecutable simplifica la distribución y ejecución del programa. También asegura que todas las funcionalidades estén integradas en un solo punto, facilitando la gestión de versiones y despliegues.

Puedes ver más detalles en el archivo README y en el archivo de configuración de CMake.

## Bibliografía

En este proyecto se ha utilizado IA

Se han usado herramientas, fuentes de información o páginas como:

<https://cplusplus.com/reference/>

<https://en.cppreference.com/w/>

<https://stackoverflow.com/>

<https://www.w3schools.com/>

luhan omar (2021). Método de búsqueda hash | implementación C++. Disponible en: <https://www.youtube.com/watch?v=XVgJY27UuM4> [Accedido el 22 de octubre de 2024].

SW Team (2023). Algoritmos de ordenación con ejemplos en C++. Disponible en:

<https://www.swhosting.com/es/comunidad/manual/algoritmos-de-ordenacion-con-ejemplos-en-c> [Accedido el 24 de octubre de 2024].

RUDE LABS (2024). Hospital Management System With C++ | C++ Project.

Disponible en: <https://www.youtube.com/watch?v=SxSR9UoLmlU> [Accedido el 15 de noviembre de 2024].

Torres A. (2016). Búsqueda lineal en vectores. Disponible en:

<https://www.youtube.com/watch?v=8DJNa1uIKUc> [Accedido el 20 de enero de 2025].

## Anexos

[https://github.com/PabloNSI/AB\\_SOTO\\_PABLO\\_UNIT20.git](https://github.com/PabloNSI/AB_SOTO_PABLO_UNIT20.git)