Nombre y Apellido: *Palavecino Axel David*
Legajo: *36778*

Homework 1

I102 - Paradigmas de Programación

Nombre y Apellido: *Palavecino Axel David*
Legajo: *36778*

Ejercicio 1: Matrices

En este ejercicio se implementa un programa en C++ que crea, llena e imprime una matriz cuadrada de números enteros. Se hace uso de punteros y memoria dinámica.

Descripción

El programa crea matrices cuadradas de diferentes tamaños, las llena con valores consecutivos y las imprime. Demuestra el manejo dinámico de memoria en C++ mediante la creación y liberación adecuada de matrices en R^2.

Funciones principales

1. `createMatrix(int n)`:
   - Crea una matriz cuadrada de enteros de tamaño n×n
   - Reserva memoria dinámicamente
   - Retorna un puntero a la matriz creada

2. `fillMatrix(int** matrix, int n)`:
   - Llena la matriz con números enteros consecutivos empezando en 1
   - Asigna los valores incrementalmente recorriendo fila por fila

3. `printMatrix(int** matrix, int n)`:
   - Imprime la matriz en orden inverso (desde la esquina inferior derecha)
   - Muestra cada elemento con formato "M_n[i][j] = valor"

4. `deleteMatrix(int** matrix, int n)`:
   - Libera la memoria asignada a la matriz
   - Evita fugas de memoria

Casos de Prueba

El programa incluye tres test que se utilizan como casos de prueba, para verificar el funcionamiento de las funciones principales:

- `test_code()`: Crea, llena, imprime y elimina una matriz 5×5

- `test_code2()`: Opera con una matriz 10×10, se utilizan ciclos para imprimir y liberar memoria
- `test_code3()`: Opera con una matriz 15×15, se utilizan ciclos para imprimir y liberar memoria

Compilación y Ejecución

Para compilar el programa:

```
bash
g++ main.cpp Matrices.cpp -std=c++20 -o matrices
```

Para compilar con información de depuración (necesaria para herramientas de análisis):

```
bash
g++ main.cpp Matrices.cpp -std=c++20 -g -o matrices
```

Para ejecutar con Valgrind para detección de fugas de memoria:

```
bash
valgrind --leak-check=full ./matrices
```

Para un análisis minucioso de fugas de memoria con Valgrind:

```
bash
valgrind --tool=memcheck --leak-check=full --show-leak-kinds=all
--track-origins=yes ./matrices
```

Para ejecutar:

```
bash
./matrices
```

Estructura del Ejercicio

- `Matrices.h`: Contiene las declaraciones de las funciones
- `Matrices.cpp`: Implementa las funciones declaradas en el header (Matrices.h)
- `main.cpp`: Contiene la función principal que ejecuta los casos de prueba.

Nombre y Apellido: *Palavecino Axel David*
Legajo: *36778*

Ejercicio 2: Sistema de Log

En este ejercicio se implementa un sistema de registro (sistema de log) en C++ que permite documentar diferentes tipos de eventos, errores y mensajes durante la ejecución de un programa.

Descripción

El sistema de log permite registrar mensajes con diferentes niveles de importancia (DEBUG, INFO, WARNING, ERROR, CRITICAL), mensajes de error específicos con información del archivo y línea de código, y mensajes de seguridad con información del usuario. Todos los mensajes se almacenan en un archivo de texto "log.txt". Para realizar este ejercicio se utilizaron sobrecarga de funciones, manejo de excepciones, enum, manejo de archivos,punteros, switch, etc.

Funciones Principales

1. `logMessage(string message, int levelImportance)`:
   - Registra un mensaje con un nivel de importancia específico
   - Los niveles son: DEBUG(0), INFO(1), WARNING(2), ERROR(3), CRITICAL(4)
   - Formato: "[NIVEL] mensaje"

2. `logMessage(string message, string file, unsigned int line_code)`:
   - Registra un mensaje de error con información del archivo y línea de código
   - Útil para rastrear excepciones
   - Formato: "[ERROR] mensaje | File: archivo | Line: línea"

3. `logMessage(string message, string user)`:
   - Registra un mensaje de seguridad con información del usuario
   - Usado para eventos de acceso o autorización
   - Formato: "[SECURITY] mensaje | User: usuario"

4. `logMessage(string message, string user)`:
   - Lee y muestra todo el contenido del archivo de log

5. `test_code()`:
   - Función que prueba el sistema de log con varios escenarios de error

- Incluye manejo de excepciones y registro de diferentes tipos de errores

Casos de Prueba

La función `test_code()` implementa varios escenarios para probar el sistema de log:

- Creación de matriz con tamaño inválido
- Manipulación de matriz con puntero nulo
- División por cero
- Acceso inválido a memoria
- Conversión de cadena a entero con formato incorrecto
- Lanzamiento de errores personalizados
- Eventos de seguridad (acciones no autorizadas)
- Registro de mensajes con diferentes niveles de importancia

Compilación y Ejecución

Para compilar el programa:

```
bash
g++ main.cpp sistema_log.cpp ../Ejercicio-1/Matrices.cpp -std=c++20 -o
sistema_log
```

Para compilar con información de depuración (necesaria para herramientas de análisis):

```
bash
g++ main.cpp sistema_log.cpp ../Ejercicio-1/Matrices.cpp -std=c++20 -g
-o sistema_log
```

Para ejecutar con Valgrind para detección de fugas de memoria:

```
bash
valgrind --leak-check=full ./sistema_log
```

Para un analisis minucioso de fugas de memoria con Valgrind:

```
bash
valgrind --tool=memcheck --leak-check=full --show-leak-kinds=all
```

```
--track-origins=yes ./sistema_log
```

Para ejecutar:

```
bash
./sistema_log
```

Estructura del Ejercicio

- sistema_log.h: Contiene las declaraciones de las funciones de logging
- sistema_log.cpp: Implementa las funciones declaradas en el header
- main.cpp: Contiene la función principal que ejecuta los casos de prueba
- log.txt: Archivo donde se guardan todos los mensajes de log (se crea automáticamente). Se define como una variable global.

Ejemplo de Uso

El sistema puede utilizarse en cualquier programa C++ para registrar eventos importantes:

```
#include "sistema_log.h"

try {
     // Alguna operación riesgosa
     int result = someRiskyFunction();
} catch (const std::exception& e) {
     // Registrar el error con información de archivo y línea
     logMessage(e.what(), "mi_archivo.cpp", 42);
}

// Registrar actividad de usuario
logMessage("Intento de acceso a datos confidenciales", "usuario123");

// Registrar mensaje informativo
logMessage("Aplicación iniciada correctamente",
static_cast<int>(Importance::INFO));
```

El archivo de log resultante mostrará un historial cronológico de todos los eventos registrados, facilitando la depuración y el seguimiento de la actividad del programa. Es bastante utilizado en aplicaciones web, sistemas embebidos, etc. para monitorear el comportamiento del sistema.

Nombre y Apellido: *Palavecino Axel David*
Legajo: *36778*

Ejercicio 3: Listas Enlazadas

Se implementa una lista simplemente enlazada en C++, es decir, el tipo más básico de lista enlazada, utilizando punteros inteligentes (smart pointers).

Descripción

En este ejercicio se implementa una lista enlazada simple en C++ utilizando punteros inteligentes (smart pointers). La lista permite almacenar valores enteros y realizar diversas operaciones como inserción y eliminación de elementos en diferentes posiciones. El uso de `shared_ptr` facilita la gestión automática de memoria y evita fugas de memoria.

Funciones Principales

1. `create_node(int value)`:
   - Crea un nuevo nodo con el valor especificado
   - Retorna un `shared_ptr<node>` que apunta al nodo creado

2. `push_front(shared_ptr<node> &head, int value)`:
   - Agrega un nuevo nodo con el valor especificado al inicio de la lista
   - Actualiza el puntero head para que apunte al nuevo nodo

3. `push_back(shared_ptr<node> &head, int value)`:
   - Agrega un nuevo nodo con el valor especificado al final de la lista
   - Recorre la lista hasta el último nodo y enlaza el nuevo nodo

4. `insert(shared_ptr<node> &head, int value, int position)`:
   - Inserta un nuevo nodo con el valor especificado en la posición indicada
   - Si la posición es 0, inserta al inicio (equivalente a push_front)
   - Si la posición es mayor que el tamaño de la lista, inserta al final

5. `erase(shared_ptr<node> &head, int position)`:
   - Elimina el nodo en la posición especificada
   - Si la posición es 0, elimina el primer nodo y actualiza head
   - Si la posición es mayor que el tamaño de la lista, elimina el último nodo

6. `print_list(shared_ptr<node> head)`:
   - Recorre la lista e imprime los valores de cada nodo
   - Muestra los nodos conectados con el formato "valor->valor->valor"

Casos de Prueba

El programa incluye tres funciones de prueba que verifican diferentes aspectos de la implementación:

`test_code1()`: Prueba básica de todas las operaciones de la lista
- Agrega elementos al inicio con push_front
- Agrega elementos al final con push_back
- Inserta elementos en posiciones específicas
- Elimina elementos en posiciones específicas
- Vacía completamente la lista

`test_code2()`: Prueba operaciones con una lista inicialmente vacía
- Demuestra cómo construir una lista desde cero
- Combina push_back y push_front para crear una lista con valores ordenados
- Prueba inserción en diferentes posiciones
- Prueba eliminación en diferentes posiciones

`test_code3()`: Prueba casos especiales y comportamiento en bordes
- Trabaja con una lista vacía
- Maneja inserción en posiciones fuera de rango
- Combina diferentes operaciones en secuencia
- Prueba eliminación de elementos en distintas posiciones

Compilación y Ejecución

Para compilar el programa:

```
bash
g++ main.cpp lista_enlazada.cpp -std=c++20 -o lista_enlazada
```

Para compilar con información de depuración:

```
bash
g++ main.cpp lista_enlazada.cpp -std=c++20 -g -o lista_enlazada
```

Para ejecutar con Valgrind para detección de fugas de memoria:

```
bash
valgrind --leak-check=full ./lista_enlazada
```

Para un análisis minucioso de memoria con Valgrind:

```
bash
valgrind --tool=memcheck --leak-check=full --show-leak-kinds=all
--track-origins=yes ./lista_enlazada
```

Para ejecutar normalmente:

```
bash
./lista_enlazada
```

Estructura del Ejercicio

- `lista_enlazada.h`: Contiene la definición de la estructura del nodo y las declaraciones de las funciones
- `lista_enlazada.cpp`: Implementa las funciones declaradas en el header
- `main.cpp`: Contiene la función principal que ejecuta los casos de prueba

Nombre y Apellido: *Palavecino Axel David*
Legajo: *36778*

Ejercicio 4: Comparación de strings y char*, tiempo de ejecución y compilación

Se implementa un programa en C++ que compara la eficiencia de comparar cadenas de texto de forma recursiva, utilizando diferentes tipos de datos para obtener el mejor rendimiento.

Descripción

En este ejercicio se implementa un programa en C++ que compara dos cadenas de texto (strings) de forma recursiva, utilizando diferentes representaciones (string y char*) y midiendo los tiempos de ejecución y compilación de cada enfoque. El objetivo es analizar las diferencias de rendimiento entre distintas implementaciones de comparación de strings y entre procesamiento en tiempo de ejecución versus tiempo de compilación.

Funciones Principales

1. `compare_recursive_string1(const string &str1, const string &str2, size_t index)`:
   - Compara dos objetos string de forma recursiva
   - Verifica primero si los tamaños son iguales
   - Compara carácter por carácter recursivamente hasta llegar al final
   - Utilizada en tiempo de ejecución

2. `compare_recursive_string2(const char *str1, const char *str2, size_t index)`:
   - Compara dos cadenas de estilo C (char*) de forma recursiva
   - Verifica recursivamente cada carácter hasta encontrar el terminador '\0'
   - Utilizada en tiempo de ejecución

3. `compare_strings1(const char *str1, const char *str2, size_t index)`:
   - Versión constexpr (evaluable en tiempo de compilación) para comparar cadenas char*
   - Permite que la comparación se realice durante la compilación si los argumentos son constantes

4. `compare_strings2(const string &str1, const string &str2, size_t index)`:
   - Versión constexpr para comparar objetos string
   - Permite que la comparación se realice durante la compilación si los argumentos son constantes

Casos de Prueba

El programa incluye dos funciones principales de prueba:

1. `test_code_ejecution()`:
   - Compara el rendimiento en tiempo de ejecución entre string y char*
   - Ejecuta 10,000 iteraciones de comparación con ambos métodos
   - Registra qué implementación es más rápida en cada iteración
   - Muestra estadísticas sobre cuál método fue más eficiente

2. `test_code_compile_time()`:
   - Evalúa el rendimiento de las funciones constexpr (tiempo de compilación)
   - También ejecuta 10,000 iteraciones para obtener resultados estadísticos
   - Compara la eficiencia de las implementaciones string vs char*
   - Muestra qué método tuvo mejor rendimiento

Compilación y Ejecución

Para compilar el programa:

```
bash
g++ main.cpp comparar_recursion.cpp -std=c++20 -o comparar_strings
```

Para compilar con información de depuración:

```
bash
g++ main.cpp comparar_recursion.cpp -std=c++20 -g -o comparar_strings
```

Para ejecutar:

```
bash
./comparar_strings
```

Nombre y Apellido: *Palavecino Axel David*
Legajo: *36778*

Resultados y Análisis

Al ejecutar el programa, se muestran estadísticas detalladas sobre:

- Cuántas veces la implementación con char* fue más rápida
- Cuántas veces la implementación con string fue más rápida
- Ejemplos de tiempos reales de ejecución para cada método
- Conclusiones sobre qué enfoque es generalmente más eficiente

El ejercicio demuestra las diferencias de rendimiento entre:
1. Tipos de datos diferentes (string vs char*)
2. Procesamiento en tiempo de compilación vs tiempo de ejecución

La expectativa teórica es que los char* sean más eficientes debido a su acceso directo a memoria, sin la sobrecarga de las funciones miembro y estructuras de datos de string. El programa verifica empíricamente si esta hipótesis se cumple en la práctica.

Estructura del Ejercicio

- `comparar_recursion.h`: Contiene las declaraciones de funciones y prototipos
- `comparar_recursion.cpp`: Implementa las funciones de comparación y pruebas
- `main.cpp`: Coordina la ejecución de las pruebas