ASPECTOS AVANZADOS DE LA POO

MANI PULACIÓN DE MATRICES DINÁMICAS

Índice

Introducción

Objetivos

En esta práctica, abordaremos el diseño avanzado y la implementación de clases en C++ para profundizar en los principios de la Programación Orientada a Objetos (POO), con un enfoque en la herencia y el uso de plantillas. El propósito es desarrollar habilidades en la estructuración de clases que manejen matrices dinámicas y diversos tipos de datos, dentro de un marco reutilizable y eficiente.

Comenzaremos implementando una clase base **Matriz** que gestionará matrices dinámicas de números enteros, incorporando operaciones aritméticas básicas y operadores lógicos y unarios para comparaciones y transformaciones matriciales. Extendiendo esta clase mediante herencia, crearemos cuatro clases derivadas que especializarán su comportamiento para adaptarse a matrices potencia, cuadradas, simétricas y antisimétricas, respetando las propiedades estructurales específicas de cada tipo.

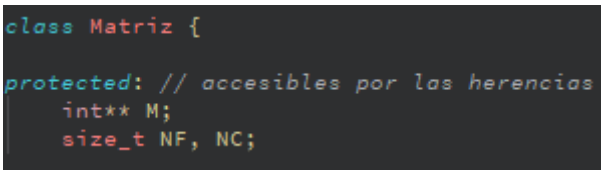
Adicionalmente, ampliaremos la flexibilidad del sistema utilizando plantillas para permitir el manejo de otros tipos de datos, como **float** o **double**. Integraremos también una clase **Capicua** en el sistema de plantillas, que permitirá realizar operaciones matemáticas con números capicúa, demostrando la capacidad de las plantillas para manipular tipos de datos no convencionales.

Herramientas de Programación Utilizadas

En este apartado, exploraremos las diversas herramientas utilizadas en el desarrollo de la práctica para garantizar la eficiencia del programa:

- **Clases y Objetos:** Definición de clases base y derivadas, La estructura de las clases **Matriz**, **MatrizA**, **MatrizS**, **MatrizC**, y **MatrizP** sugiere un uso intensivo de clases y objetos, que son fundamentales para la POO. Estas clases modelan diferentes tipos de matrices y su interacción.

  En el proyecto, la clase **Matriz** sirve como un ejemplo destacado de encapsulamiento y diseño orientado a objetos. Diseñada para manejar matrices bidimensionales, encapsula los datos en una estructura de matriz de enteros (**int** M**), con atributos protegidos **NF** y **NC** que indican el número de filas y columnas. Este diseño muestra cómo las clases pueden estructurar datos complejos y permitir el acceso controlado a través de la herencia, demostrando la eficacia del encapsulamiento en aplicaciones reales.



- **Constructores**: Cada clase de matriz, como se describe en los enunciados de la práctica, tiene constructores que inicializan matrices de diferentes tamaños y propiedades, lo cual es crucial para establecer un estado inicial adecuado y manejar la memoria dinámicamente.

En mi código, el constructor de la clase **MatrizA** se utiliza para inicializar matrices antisimétricas con dimensiones uniformes. Heredando de **MatrizC**, este constructor toma un único valor para establecer tanto filas como columnas y luego aplica configuraciones específicas para matrices antisimétricas, garantizando la coherencia y precisión desde el inicio.

```
MatrizA(size_t dimension) : MatrizC(dimension) {
    inicializarAntisimetrica();
}
```

- **Destructores:** Se utilizan para gestionar la liberación de recursos, especialmente importante en la gestión de matrices dinámicas
  En mi código, el destructor de la clase **Matriz** asegura una gestión de recursos eficiente al liberar adecuadamente la memoria asignada a la matriz bidimensional M. Itera a través de las filas, liberando primero la memoria de cada una y luego la del arreglo completo de punteros, previniendo así cualquier fuga de memoria al destruir objetos de esta clase.

```
~Matriz() {
    for(size_t i = 0; i < NF; ++i) {
        delete[] M[i];
    }
    delete[] M;
}
```

- **Sobrecarga de operadores**: Los archivos incluyen implementaciones para operadores como **+**, **-**, *****, +=, -=, *=, **==**, y **!=**, permitiendo que las matrices interactúen de manera intuitiva. También se menciona el uso de operadores unarios como la transposición (**~**) y la negación (**-**).

  Por ejemplo, sobrecargar el operador **+=** para la clase **Capicua**, permitiendo realizar auto-suma de matrices capicuas.

```
Capicua& operator+=(const Capicua& otra) {
    this->ordinal += otra.ordinal;
    return *this;
}
```

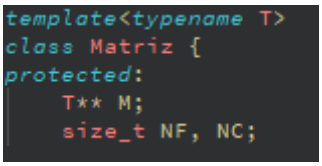
- **Memoria Dinámica:** El uso de punteros y memoria dinámica, dado que las matrices requieren dimensiones variables, es muy probable que se utilice memoria dinámica a través de punteros, especialmente en la clase **Matriz** que maneja matrices dinámicas compuestas por números enteros.

- **Herencia**: Se utiliza la herencia para extender la funcionalidad de la clase **Matriz** en clases especializadas como **MatrizP** (matriz potencia) y **MatrizC** (matriz cuadrada), a su vez **MatrizS** y **MatrizA** heredan de MatrizC**,** demostrando la relación entre clases y objetos en un contexto de aplicación real.

```
class MatrizS : public MatrizC {
```

- **Plantillas (Templates)**: Las clases están diseñadas para ser utilizadas con plantillas, lo que permite que los mismos mecanismos de matrices se apliquen a diferentes tipos de datos (por ejemplo, **int**, **double**, **float**), aumentando la reutilización del código.

  En mi código, la clase **Matriz** utiliza una plantilla para manejar matrices de cualquier tipo de dato **T**. Esta estructura generalizada permite crear matrices dinámicas para diversos tipos de datos, ofreciendo flexibilidad y reutilización del código en múltiples aplicaciones.



- **Encapsulación:** Las clases encapsulan el comportamiento y los datos específicos de las matrices, ocultando detalles internos y exponiendo solo las interfaces necesarias para su uso por otras partes del programa.

Código Fuente

El código fuente desarrollado en esta práctica constituye una implementación avanzada y detallada de técnicas de programación en C++ moderno, orientadas a la manipulación eficaz de estructuras de datos complejas como son las matrices dinámicas, todo bajo el paradigma de la Programación Orientada a Objetos (POO). A continuación, detallaremos las características fundamentales de la arquitectura del código y cómo estas técnicas se materializan en las diversas clases especializadas que hemos desarrollado.

Inicialmente, se presenta la clase **Matriz**, que funciona como la clase base central para todas las operaciones relacionadas con matrices. Esta clase gestiona matrices dinámicas compuestas por enteros y ofrece una suite de métodos fundamentales incluyendo varios constructores, que permiten la creación de matrices desde inicializadas a cero hasta matrices pobladas con valores aleatorios generados dinámicamente. Además, cuenta con un destructor diseñado para gestionar la desalocación eficiente de los recursos utilizados, asegurando la prevención de fugas de memoria. Los operadores sobrecargados facilitan la realización de operaciones aritméticas como suma, resta, y producto matricial, además de operaciones escalares y comparativas, junto con operadores unarios para la transposición y la negación de matrices, demostrando la potente capacidad de C++ para extender la funcionalidad de objetos mediante sobrecarga de operadores.

Sobre esta base, se extienden cuatro clases derivadas utilizando el mecanismo de herencia, cada una especializada para un tipo particular de matriz:

- **MatrizP**: Hereda de **Matriz** y se especializa en la generación de matrices de Vandermonde, que se caracterizan por no ser cuadradas. Esta clase adapta el constructor de la clase base para ajustar su comportamiento al especificado para este tipo de matrices.
- **MatrizC**: También una extensión de **Matriz**, esta clase está diseñada para manejar matrices cuadradas, es decir, matrices cuya altura y anchura son iguales.
- **MatrizS** y **MatrizA**: Estas clases son especializaciones de **MatrizC** y se encargan de implementar matrices simétricas y antisimétricas, respectivamente, añadiendo restricciones adicionales y ajustando el comportamiento heredado para cumplir con propiedades matemáticas muy específicas.

El empleo de plantillas en C++ se introduce para conferir a estas clases la capacidad de operar con distintos tipos de datos, desde tipos primitivos como enteros y flotantes hasta tipos definidos por el usuario, siempre y cuando estos tipos soporten los operadores necesarios. Esto no solo demuestra la versatilidad del sistema de tipos en C++, sino también cómo se pueden construir bibliotecas de software altamente flexibles y desacopladas de los tipos específicos de datos.

Adicionalmente, la clase **Capicua** se integra dentro del sistema de plantillas para ofrecer funcionalidad especializada en el manejo de números capicúa dentro de estructuras matriciales. Este diseño ilustra cómo tipos personalizados pueden ser incorporados de manera eficaz en sistemas más grandes, permitiendo operaciones específicas alineadas con conceptos matemáticos o de modelado particulares.

Desde una perspectiva de gestión de recursos, la manipulación cuidadosa de la memoria dinámica es fundamental, asegurando asignaciones eficientes y seguras y evitando problemas comunes en programación a bajo nivel como las fugas de memoria.

En conclusión, el conjunto de clases desarrolladas en esta práctica no solo aborda las necesidades operativas matriciales, sino que también encapsula principios de diseño de software críticos como la encapsulación, la herencia, el polimorfismo, y la generalización mediante el uso de plantillas, configurando así una solución integrada, cohesiva y altamente extensible que puede ser adaptada a diversas aplicaciones y tipos de datos.

Errores y Tecnologías

Durante el desarrollo de la práctica, enfrentamos retos técnicos significativos, particularmente en la gestión de memoria y la ejecución de operaciones matemáticas complejas. La gestión de memoria dinámica se mostró esencial, sobre todo para manejar matrices de gran tamaño y los voluminosos resultados intermedios que estas generan.

Además, surgieron desafíos específicos con la clase **Capicua** al gestionar y visualizar grandes secuencias de números capicúa, lo que resultó en problemas de rendimiento y errores por desbordamiento de memoria. Para mitigar estos problemas, implementamos optimizaciones en la gestión de memoria, utilizando técnicas como la reserva anticipada de espacio y la reutilización de objetos, lo cual redujo significativamente la sobrecarga de memoria.

También mejoramos la eficiencia de los cálculos realizados por la clase **Capicua**, ajustando algoritmos para manejar grandes números de manera más efectiva y reducir la cantidad de datos procesados en memoria. Estas mejoras no solo resolvieron los problemas de rendimiento, sino que también elevaron la estabilidad y funcionalidad del sistema, facilitando el trabajo con extensas secuencias de números capicúa.

Experimentación y validación

En la fase de experimentación, se llevaron a cabo pruebas concretas para evaluar la robustez y funcionalidad del programa. Algunos experimentos incluyeron:

EJERICICIO 1

Pruebas expuestas en el guion:

1. **En el siguiente ejemplo mostramos un posible uso de Matriz. La primera operación que se muestra produce una matriz válida pero la segunda intenta sumar dos matrices de distinto tamaño, por lo que el resultado por pantalla debe ser "MATRIZ NULA":**

```
Matriz M1(4,6),M2(4,6), M3(6,4);
cout<<"Matrices de entrada:"<<endl;
cout<<M1<<endl<<M2<<endl<<M3<<endl;
M1 = (M1 - M2) * M3;
M1 *= M2;
cout<<"operacion1:"<<endl<<M1;
cout<<"operacion2:"<<endl<<M2+M3;
```

```
Matrices de entrada:
4 2 4 8 6 6
5 6 3 9 3 6
6 0 7 2 9 9
9 8 1 3 5 9

4 6 6 5 5 7
3 0 6 7 3 9
0 5 4 2 0 2
5 9 9 0 0 8

2 2 9 4
4 8 7 3
1 2 7 1
5 2 1 0
7 7 3 9
8 7 2 5

operacion1:
-156 -304 -446 -306 -110 -454
171 313 452 328 142 472
1366 2241 2700 1362 840 2794
587 784 983 480 366 1092
operacion2:
MATRIZ NULA
```

2. **Ejemplo de uso en el que se mezclan distintos tipos de matrices y operadores:**

```
Matriz M(5,4);
MatrizS S(4);
MatrizA A(5);
MatrizP P(4,5);
cout<<"M=\n"<<M<<"S=\n"<<S<<"A=\n"<<A<<"P=\n"<<P<<endl;
if(M==~M && S==~S){
    cout<<"Propiedades traspuesta correctas"<<endl;
}else{
    cout<<"Algo ha ido mal"<<endl;
}
if(A == -~A){
    cout<<"Propiedad antisimetrica correcta"<<endl;
}else{
    cout<<"Algo ha ido mal"<<endl;
}
cout<<"Resultado todo a ceros:"<<endl;
cout<< A + ~A;
```

```
0 0 0 6
5 5 0 4
6 2 2 8
S=
-10 -10 1 5
-10 2 0 -5
1 0 -10 -4
5 -5 -4 -2
A=
0 -1 7 -4 2
1 0 0 -8 -2
-7 0 0 7 8
4 8 -7 0 -3
-2 2 -8 3 0
P=
1 6 36 216 1296
1 7 49 343 2401
1 4 16 64 256
1 4 16 64 256

Propiedades traspuesta correctas
Propiedad antisimetrica correcta
Resultado todo a ceros:
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

Pruebas personales:

1. **Prueba dimensión 0 en Matrices**

```
MatrizC M1(0);
Matriz M2(0,0);
MatrizP M3(0,0);
MatrizS M4(0);
MatrizA M5(0);

cout << "\033[1mMatrizC (0):\033[0m\n" << M1 << "\n";
cout << "\033[1mMatriz (0,0):\033[0m\n" << M2 << "\n";
cout << "\033[1mMatrizP (0,0):\033[0m\n" << M3 << "\n";
cout << "\033[1mMatrizS (0):\033[0m\n" << M4 << "\n";
cout << "\033[1mMatrizA (0):\033[0m\n" << M5 << "\n";
```

```
MatrizC (0):
MATRIZ NULA

Matriz (0,0):
MATRIZ NULA

MatrizP (0,0):
MATRIZ NULA

MatrizS (0):
MATRIZ NULA

MatrizA (0):
MATRIZ NULA
```

2. Prueba igualaciones

```
cout << "\033[1m------ IGUALACIONES ------\033[0m\n" << endl;
Matriz M(4, 7);
MatrizC MC(3);
MatrizP MP(M);
MatrizA MA(MC);
MatrizS MS(MC);
MatrizC MC1(M);

cout << "\033[1mMatriz MC(3) = M(4,7):\033[0m\n" << MC1 << "\n";
cout << "\033[1mMatriz MP(4,7) = M(4,7):\033[0m\n" << MP << "\n";
cout << "\033[1mMatriz MA = MC:\033[0m:\n" << MA << "\n";
cout << "\033[1mMatriz MS = MC:\033[0m:\n" << MS << "\n";
```

```
------ IGUALACIONES ------

Matriz MC(3) = M(4,7):
MATRIZ NULA

Matriz MP(4,7) = M(4,7):
MATRIZ NULA

Matriz MA = MC::
MATRIZ NULA

Matriz MS = MC::
MATRIZ NULA
```

3. Prueba operaciones con Matriz

```
------ MATRICES ------

Matriz A:
8 9 2
3 8 6
8 9 8

Matriz B:
6 7 8
6 4 7
6 5 8
```

```
------ OPERACIONES ------

SUMA A + B:
14 16 10
9 12 13
14 14 16

Resta A - B:
2 2 -6
-3 4 -1
2 4 0

Multiplicacion A * B:
114 102 143
102 83 128
150 132 191
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
14 16 10
9 12 13
14 14 16

A despues de A -= B:
8 9 2
3 8 6
8 9 8

X despues de X *= Y:
63 15
121 63
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
24 27 6
9 24 18
24 27 24

Producto de 2 por A:
16 18 4
6 16 12
16 18 16

A despues de A *= 4:
32 36 8
12 32 24
32 36 32
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
32 12 32
36 32 36
8 24 32

Opuesta de A:
-32 -36 -8
-12 -32 -24
-32 -36 -32
```

4. Prueba operaciones con MatrizC

```
------ MATRICES ------

Matriz A:
6 6 4
5 5 8
1 1 8

Matriz B:
4 8 2
7 3 3
9 9 3
```

```
------ OPERACIONES ------

SUMA A + B:
10 14 6
12 8 11
10 10 11

Resta A - B:
2 -2 2
-2 2 5
-8 -8 5

Multiplicacion A * B:
102 102 42
127 127 49
83 83 29
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
10 14 6
12 8 11
10 10 11

A despues de A -= B:
6 6 4
5 5 8
1 1 8

X despues de X *= Y:
40 44
38 30
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
18 18 12
15 15 24
3 3 24

Producto de 2 por A:
12 12 8
10 10 16
2 2 16

A despues de A *= 4:
24 24 16
20 20 32
4 4 32
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
24 20 4
24 20 4
16 32 32

Opuesta de A:
-24 -24 -16
-20 -20 -32
-4 -4 -32
```

5. **Prueba operaciones con MatrizP**

```
------ MATRICES ------

Matriz A:
1 7 49
1 4 16
1 7 49

Matriz B:
1 2 4
1 7 49
1 4 16
```

```
------ OPERACIONES ------

SUMA A + B:
2 9 53
2 11 65
2 11 65

Resta A - B:
0 5 45
0 -3 -33
0 3 33

Multiplicacion A * B:
57 247 1131
21 94 456
57 247 1131
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
2 9 53
2 11 65
2 11 65

A despues de A -= B:
1 7 49
1 4 16
1 7 49

X despues de X *= Y:
14 16
34 24
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
3 21 147
3 12 48
3 21 147

Producto de 2 por A:
2 14 98
2 8 32
2 14 98

A despues de A *= 4:
4 28 196
4 16 64
4 28 196
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
4 4 4
28 16 28
196 64 196

Opuesta de A:
-4 -28 -196
-4 -16 -64
-4 -28 -196
```

6. **Prueba operaciones con matrizA**

```
------ MATRICES ------

Matriz A:
0 -9 -3
9 0 -5
3 5 0

Matriz B:
0 -3 -1
3 0 0
1 0 0
```

```
------ OPERACIONES ------

SUMA A + B:
0 -12 -4
12 0 -5
4 5 0

Resta A - B:
0 -6 -2
6 0 -5
2 5 0

Multiplicacion A * B:
-30 0 0
-5 -27 -9
15 -9 -3
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
0 -12 -4
12 0 -5
4 5 0

A despues de A -= B:
0 -9 -3
9 0 -5
3 5 0

X despues de X *= Y:
73 105
77 108
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
0 -27 -9
27 0 -15
9 15 0

Producto de 2 por A:
0 -18 -6
18 0 -10
6 10 0

A despues de A *= 4:
0 -36 -12
36 0 -20
12 20 0
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
0 36 12
-36 0 20
-12 -20 0

Opuesta de A:
0 36 12
-36 0 20
-12 -20 0
```

7. Prueba operaciones con matrizS

```
------ MATRICES ------

Matriz A:
8 6 9
6 -5 -2
9 -2 -3

Matriz B:
-10 -3 5
-3 -10 -9
5 -9 8
```

```
------ OPERACIONES ------

SUMA A + B:
-2 3 14
3 -15 -11
14 -11 5

Resta A - B:
18 9 4
9 5 7
4 7 -11

Multiplicacion A * B:
-53 -165 58
-55 50 59
-99 20 39
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
-2 3 14
3 -15 -11
14 -11 5

A despues de A -= B:
8 6 9
6 -5 -2
9 -2 -3

X despues de X *= Y:
61 45
84 60
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
24 18 27
18 -15 -6
27 -6 -9

Producto de 2 por A:
16 12 18
12 -10 -4
18 -4 -6

A despues de A *= 4:
32 24 36
24 -20 -8
36 -8 -12
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
32 24 36
24 -20 -8
36 -8 -12

Opuesta de A:
-32 -24 -36
-24 20 8
-36 8 12
```

8. **Prueba operaciones con todas las matrices**

```
MatrizS M1(3);
MatrizP M2(3,3);
MatrizA M3(3);
MatrizC M4(3);

M4 = (M1*(M2+M3))-M4;

cout<<M1<<endl<<M2<<endl<<M3<<endl<<M4;
```

```
8 5 2
5 2 1
2 1 -4

1 3 9
1 3 9
1 8 64

0 3 3
-3 0 -1
-3 1 0

-15 80 257
-5 37 135
2 -21 -233
```

EJERICICIO 2

Pruebas expuestas en el guion:

1. **Prueba con matrices de flotantes:**

```
Matriz<float> M(5,4);
MatrizS<float> S(4);
MatrizA<float> A(5);
MatrizP<float> P(4,5);
cout<<"M=\n"<<M<<"S=\n"<<S<<"A=\n"<<A<<"P=\n"<<P<<endl;
if(M==~M && S==~S){
    cout<<"Propiedades traspuesta correctas"<<endl;
}else{
    cout<<"Algo ha ido mal"<<endl;
}
if(A == -~A){
    cout<<"Propiedad antisimetrica correcta"<<endl;
}else{
    cout<<"Algo ha ido mal"<<endl;
}
cout<<"Resultado todo a ceros:"<<endl;
cout<< A + ~A;
```

```
M=
9.3 9.9 5.9 0.2
6.9 8.2 7.4 1.3
9 4 8.7 2.4
7 6.2 5.3 1.4
7.2 4.7 4.7 7.8
S=
6.5 0.8 1.8 3.6
0.8 3.7 0 0.7
1.8 0 8.1 5.9
3.6 0.7 5.9 1.8
A=
0 -6.7 -2.1 -8.5 -9.3
6.7 0 -5.3 -3.5 -9.8
2.1 5.3 0 -0.4 -9.3
8.5 3.5 0.4 0 -2.9
9.3 9.8 9.3 2.9 0
P=
1 8.9 79.21 704.969 6274.22
1 4.7 22.09 103.823 487.968
1 4.8 23.04 110.592 530.842
1 4.5 20.25 91.125 410.062

Propiedades traspuesta correctas
Propiedad antisimetrica correcta
Resultado todo a ceros:
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

2. **Prueba clase Capicua, tanto a nivel independiente como actuando de base en las matrices de templates**

```
cout<<"Ejemplo de serie Capicua:"<<endl;
for(int i=-10;i<=10;i++){
    cout<<"Capicua("<<i<<")="<<Capicua(i)<<endl;
}
cout<<"Ejemplo de matrices basadas en Capicua:"<<endl;
```

```
Ejemplo de serie Capicua:
Capicua(-10)=11
Capicua(-9)=9
Capicua(-8)=8
Capicua(-7)=7
Capicua(-6)=6
Capicua(-5)=5
Capicua(-4)=4
Capicua(-3)=3
Capicua(-2)=2
Capicua(-1)=1
Capicua(0)=1
Capicua(1)=1
Capicua(2)=2
Capicua(3)=3
Capicua(4)=4
Capicua(5)=5
Capicua(6)=6
Capicua(7)=7
Capicua(8)=8
Capicua(9)=9
Capicua(10)=11
```

```
Matriz<Capicua> M(5,5);
MatrizA<Capicua> A(5);
MatrizS<Capicua> S(5);
MatrizP<Capicua> P(4,5);
cout<<"M=\n"<<M<<"A=\n"<<A<<"S=\n"<<S<<"P=\n"<<P<<endl;
cout<<"P*(M+S): "<<endl<<P*(M+S);
```

```
Ejemplo de matrices basadas en Capicua:
M=
9 6 9 7 1
5 5 1 1 9
6 4 1 4 4
7 8 5 4 6
1 1 7 4 3
A=
1 9 3 3 6
9 1 9 7 1
3 9 1 6 1
3 7 6 1 1
6 1 1 1 1
S=
2 2 2 8 5
2 7 4 6 8
2 4 6 1 7
8 6 1 8 8
5 8 7 8 1
P=
1 3 9 181 727
1 1 1 1 1
1 7 404 24442 1402041
1 4 77 555 15751

P*(M+S):
89798 225522 387783 379973 76067
393 434 353 434 444
100050001 168979861 260636062 232393232 50722705
1665661 2386832 3113113 2965692 1172711
```

Pruebas personales:

1. **Prueba dimensión 0 en Matrices**

```
cout << "\033[1m------ DIMENSION 0 ------\033[0m\n" << endl;
Matriz<float> M(0, 0);
MatrizC<float> MC(0);
MatrizP<float> MP(0,0);
MatrizA<float> MA(0);
MatrizS<float> MS(0);

cout << "\033[1mMatriz M:\033[0m\n" << M << "\n";
cout << "\033[1mMatriz MC:\033[0m:\n" << MC << "\n";
cout << "\033[1mMatriz MP:\033[0m:\n" << MP << "\n";
cout << "\033[1mMatriz MA:\033[0m:\n" << MA << "\n";
cout << "\033[1mMatriz MS:\033[0m:\n" << MS << "\n";
```

```
------ DIMENSION 0 ------

Matriz M:
MATRIZ NULA

Matriz MC::
MATRIZ NULA

Matriz MP::
MATRIZ NULA

Matriz MA::
MATRIZ NULA

Matriz MS::
MATRIZ NULA
```

2. Prueba igualaciones

```
cout << "\033[1m------ IGUALACIONES ------\033[0m\n" << endl;
Matriz<float> M(4, 7);
MatrizC<float> MC(3);
MatrizP<float> MP(M);
MatrizA<float> MA(MC);
MatrizS<float> MS(MC);
MatrizC<float> MC1(M);

cout << "\033[1mMatriz MC(3) = M(4,7):\033[0m\n" << MC1 << "\n";
cout << "\033[1mMatriz MP(4,7) = M(4,7):\033[0m\n" << MP << "\n";
cout << "\033[1mMatriz MA = MC:\033[0m:\n" << MA << "\n";
cout << "\033[1mMatriz MS = MC:\033[0m:\n" << MS << "\n";
```

```
------ IGUALACIONES ------

Matriz MC(3) = M(4,7):
MATRIZ NULA

Matriz MP(4,7) = M(4,7):
MATRIZ NULA

Matriz MA = MC::
MATRIZ NULA

Matriz MS = MC::
MATRIZ NULA
```

3. Prueba operaciones con Matriz<float>

```
------ MATRICES ------

Matriz A:
0.2 3.7 4.7
8.9 1.3 6.3
1.2 5.4 6.7

Matriz B:
5.4 8.2 0.4
9.6 6.1 1.6
5.7 7.8 7.2
```

```
------ OPERACIONES ------

SUMA A + B:
5.6 11.9 5.1
18.5 7.4 7.9
6.9 13.2 13.9

Resta A - B:
-5.2 -4.5 4.3
-0.700001 -4.8 4.7
-4.5 -2.4 -0.5

Multiplicacion A * B:
63.39 60.87 39.84
96.45 130.05 51
96.51 95.04 57.36
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
5.6 11.9 5.1
18.5 7.4 7.9
6.9 13.2 13.9

A despues de A -= B:
0.2 3.7 4.7
8.9 1.3 6.3
1.2 5.4 6.7

X despues de X *= Y:
34.52 59.82
74.17 111.63
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
0.599999 11.1 14.1
26.7 3.9 18.9
3.6 16.2 20.1

Producto de 2 por A:
0.4 7.4 9.4
17.8 2.6 12.6
2.4 10.8 13.4

A despues de A *= 4:
0.799999 14.8 18.8
35.6 5.2 25.2
4.8 21.6 26.8
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
0.799999 35.6 4.8
14.8 5.2 21.6
18.8 25.2 26.8

Opuesta de A:
-0.799999 -14.8 -18.8
-35.6 -5.2 -25.2
-4.8 -21.6 -26.8
```

4. Prueba operaciones con MatrizC<float>

```
------ MATRICES ------

Matriz A:
5.1 3.6 6.9
5.9 6.1 3.9
6.4 1.6 3.5

Matriz B:
7.1 5.6 1.7
6 4.5 8.1
2.5 8.7 9.1
```

```
------ OPERACIONES ------

SUMA A + B:
12.2 9.2 8.6
11.9 10.6 12
8.9 10.3 12.6

Resta A - B:
-2 -2 5.2
-0.0999999 1.6 -4.2
3.9 -7.1 -5.6

Multiplicacion A * B:
75.06 104.79 100.62
88.24 94.42 94.93
63.79 73.49 55.69
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
12.2 9.2 8.6
11.9 10.6 12
8.9 10.3 12.6

A despues de A -= B:
5.1 3.6 6.9
5.9 6.1 3.9
6.4 1.6 3.5

X despues de X *= Y:
81.32 77.72
110.77 107.79
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
15.3 10.8 20.7
17.7 18.3 11.7
19.2 4.8 10.5

Producto de 2 por A:
10.2 7.2 13.8
11.8 12.2 7.8
12.8 3.2 7

A despues de A *= 4:
20.4 14.4 27.6
23.6 24.4 15.6
25.6 6.4 14
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
20.4 23.6 25.6
14.4 24.4 6.4
27.6 15.6 14

Opuesta de A:
-20.4 -14.4 -27.6
-23.6 -24.4 -15.6
-25.6 -6.4 -14
```

5. **Prueba operaciones con MatrizP<float>**

```
------ MATRICES ------

Matriz A:
1 2.9 8.41
1 0.9 0.81
1 9.3 86.49

Matriz B:
1 2.7 7.29
1 9.4 88.36
1 0.9 0.81
```

```
------ OPERACIONES ------

SUMA A + B:
2 5.6 15.7
2 10.3 89.17
2 10.2 87.3

Resta A - B:
0 0.2 1.12
0 -8.5 -87.55
0 8.4 85.68

Multiplicacion A * B:
12.31 37.529 270.346
2.71 11.889 87.4701
96.79 167.961 899.095
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
2 5.6 15.7
2 10.3 89.17
2 10.2 87.3

A despues de A -= B:
1 2.9 8.41
1 0.9 0.809998
1 9.3 86.49

X despues de X *= Y:
42.54 68.26
71.98 128.34
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
3 8.7 25.23
3 2.7 2.42999
3 27.9 259.47

Producto de 2 por A:
2 5.8 16.82
2 1.8 1.62
2 18.6 172.98

A despues de A *= 4:
4 11.6 33.64
4 3.6 3.23999
4 37.2 345.96
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
4 4 4
11.6 3.6 37.2
33.64 3.23999 345.96

Opuesta de A:
-4 -11.6 -33.64
-4 -3.6 -3.23999
-4 -37.2 -345.96
```

6. Prueba operaciones con MatrizS<float>

```
------ MATRICES ------

Matriz A:
1.1 4.6 7.1
4.6 2.7 8.5
7.1 8.5 1.3

Matriz B:
9.8 8.8 2.7
8.8 7.1 9.6
2.7 9.6 0.2
```

```
------ OPERACIONES ------

SUMA A + B:
10.9 13.4 9.8
13.4 9.8 18.1
9.8 18.1 1.5

Resta A - B:
-8.7 -4.2 4.4
-4.2 -4.4 -1.1
4.4 -1.1 1.1

Multiplicacion A * B:
70.43 110.5 48.55
91.79 141.25 40.04
147.89 135.31 101.03
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
10.9 13.4 9.8
13.4 9.8 18.1
9.8 18.1 1.5

A despues de A -= B:
1.1 4.6 7.1
4.6 2.7 8.5
7.1 8.5 1.3

X despues de X *= Y:
30.4 64.55
24.73 21.57
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
3.3 13.8 21.3
13.8 8.1 25.5
21.3 25.5 3.9

Producto de 2 por A:
2.2 9.2 14.2
9.2 5.4 17
14.2 17 2.6

A despues de A *= 4:
4.4 18.4 28.4
18.4 10.8 34
28.4 34 5.2
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
4.4 18.4 28.4
18.4 10.8 34
28.4 34 5.2

Opuesta de A:
-4.4 -18.4 -28.4
-18.4 -10.8 -34
-28.4 -34 -5.2
```

7. Prueba operaciones con MatrizA<float>

```
------ MATRICES ------

Matriz A:
0 -5.1 -4.8
5.1 0 -7.8
4.8 7.8 0

Matriz B:
0 -3.5 -1.9
3.5 0 -1.9
1.9 1.9 0
```

```
------ OPERACIONES ------

SUMA A + B:
0 -8.6 -6.7
8.6 0 -9.7
6.7 9.7 0

Resta A - B:
0 -1.6 -2.9
1.6 0 -5.9
2.9 5.9 0

Multiplicacion A * B:
-26.97 -9.12 9.69
-14.82 -32.67 -9.69
27.3 -16.8 -23.94
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
0 -8.6 -6.7
8.6 0 -9.7
6.7 9.7 0

A despues de A -= B:
0 -5.1 -4.8
5.1 0 -7.8
4.8 7.8 0

X despues de X *= Y:
40.17 17.28
85.15 72.56
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
0 -15.3 -14.4
15.3 0 -23.4
14.4 23.4 0

Producto de 2 por A:
0 -10.2 -9.6
10.2 0 -15.6
9.6 15.6 0

A despues de A *= 4:
0 -20.4 -19.2
20.4 0 -31.2
19.2 31.2 0
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
0 20.4 19.2
-20.4 0 31.2
-19.2 -31.2 0

Opuesta de A:
-0 20.4 19.2
-20.4 -0 31.2
-19.2 -31.2 -0
```

8. **Prueba operaciones con todas las matrices<float>**

```
MatrizS<float> M1(3);
MatrizP<float> M2(3,3);
MatrizA<float> M3(3);
MatrizC<float> M4(3);

M4 = (M1*(M2+M3))-M4;

cout<<M1<<endl<<M2<<endl<<M3<<endl<<M4;
```

```
7 1.8 5.9
1.8 3.5 0.4
5.9 0.4 4.3

1 0.7 0.49
1 8.8 77.44
1 7.2 51.84

0 -3 -6.5
3 0 -2.6
6.5 2.6 0

50.45 54.66 397.198
18.4 25.18 266.058
30.15 30.89 208.889
```

9. **Prueba operaciones con Matriz<Capicua>**

```
------ MATRICES ------

Matriz A:
1 7 4
7 1 3
1 3 9

Matriz B:
4 3 6
3 1 5
4 2 1
```

```
------ OPERACIONES ------

SUMA A + B:
5 11 11
11 2 8
5 5 11

Resta A - B:
3 4 2
4 1 2
3 1 8

Multiplicacion A * B:
333 101 373
353 202 424
414 161 222
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
5 11 11
11 2 8
5 5 11

A despues de A -= B:
1 7 4
7 1 3
1 3 9

X despues de X *= Y:
848 858
808 989
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
3 121 33
121 3 9
3 9 181

Producto de 2 por A:
2 55 8
55 2 6
2 6 99

A despues de A *= 4:
4 191 77
191 4 33
4 33 272
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
4 191 4
191 4 33
77 33 272

Opuesta de A:
4 191 77
191 4 33
4 33 272
```

10. Prueba operaciones con MatrizC<Capicua>

```
------ MATRICES ------

Matriz A:
8 7 1
3 4 6
9 7 8

Matriz B:
3 4 8
8 7 9
1 5 1
```

```
------ OPERACIONES ------

SUMA A + B:
22 22 9
22 22 66
11 33 9

Resta A - B:
5 3 7
5 3 3
8 2 7

Multiplicacion A * B:
737 787 3003
393 626 585
838 2772 4554
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
22 22 9
22 22 66
11 33 9

A despues de A -= B:
8 7 1
3 4 6
9 7 8

X despues de X *= Y:
232 1001
333 3773
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
151 121 3
9 33 99
181 121 151

Producto de 2 por A:
77 55 2
6 8 33
99 55 77

A despues de A *= 4:
232 191 4
33 77 151
272 191 232
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
232 33 272
191 77 191
4 151 232

Opuesta de A:
232 191 4
33 77 151
272 191 232
```

11. Prueba operaciones con MatrizP<Capicua>

```
------ MATRICES ------

Matriz A:
1 1 1
1 9 727
1 8 555

Matriz B:
1 1 1
1 8 555
1 4 77
```

```
------ OPERACIONES ------

SUMA A + B:
2 2 2
2 88 4664
2 33 717

Resta A - B:
1 1 1
1 1 88
1 4 393

Multiplicacion A * B:
4 55 737
838 29992 875578
656 22322 539935
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
2 2 2
2 88 4664
2 33 717

A despues de A -= B:
1 1 1
1 9 727
1 8 555

X despues de X *= Y:
88 232
66 171
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
3 3 3
3 181 14441
3 151 9339

Producto de 2 por A:
2 2 2
2 99 6336
2 77 2992

A despues de A *= 4:
4 4 4
4 272 22522
4 232 15751
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
4 4 4
4 272 232
4 22522 15751

Opuesta de A:
4 4 4
4 272 22522
4 232 15751
```

12. Prueba operaciones con MatrizS<Capicua>

```
------ MATRICES ------

Matriz A:
1 2 9
2 1 1
9 1 9

Matriz B:
7 7 1
7 3 1
1 1 8
```

```
------ OPERACIONES ------

SUMA A + B:
8 9 11
9 4 2
11 2 88

Resta A - B:
6 5 8
5 2 1
8 1 1

Multiplicacion A * B:
222 141 676
141 101 33
717 676 747
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
8 9 11
9 4 2
11 2 88

A despues de A -= B:
1 2 9
2 1 1
9 1 9

X despues de X *= Y:
393 393
969 757
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
3 6 181
6 3 3
181 3 181

Producto de 2 por A:
2 4 99
4 2 2
99 2 99

A despues de A *= 4:
4 8 272
8 4 4
272 4 272
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
4 8 272
8 4 4
272 4 272

Opuesta de A:
4 8 272
8 4 4
272 4 272
```

13. Prueba operaciones con MatrizA<Capicua>

```
------ MATRICES ------

Matriz A:
1 4 8
4 1 7
8 7 1

Matriz B:
1 1 7
1 1 4
7 4 1
```

```
------ OPERACIONES ------

SUMA A + B:
2 5 66
5 2 22
66 22 2

Resta A - B:
1 3 1
3 1 3
1 3 1

Multiplicacion A * B:
494 272 2
343 212 292
141 4 737
```

```
------ AUTO-OPERACIONES ------

A despues de A += B:
2 5 66
5 2 22
66 22 2

A despues de A -= B:
1 4 8
4 1 7
8 7 1

X despues de X *= Y:
828 585
565 383
```

```
------ PRODUCTO ESCALAR ------

Producto de A por 3:
3 33 151
33 3 121
151 121 3

Producto de 2 por A:
2 8 77
8 2 55
77 55 2

A despues de A *= 4:
4 77 232
77 4 191
232 191 4
```

```
------ OPERADORES UNARIOS ------

Traspuesta de A:
4 77 232
77 4 191
232 191 4

Opuesta de A:
4 77 232
77 4 191
232 191 4
```

14. Prueba operaciones con todas las matrices<Capicua>

```
MatrizS<Capicua> M1(3);
MatrizP<Capicua> M2(3,3);
MatrizA<Capicua> M3(3);
MatrizC<Capicua> M4(3);

M4 = (M1*(M2+M3))-M4;

cout<<M1<<endl<<M2<<endl<<M3<<endl<<M4;
```

```
4 1 9
1 3 6
9 6 8

1 9 727
1 7 404
1 6 272

1 7 7
7 1 9
7 9 1

797 4884 56865
585 1771 30903
3113 8118 200002
```

Conclusión

El desarrollo de esta práctica ha constituido una oportunidad excepcionalmente valiosa para la aplicación práctica de los principios teóricos aprendidos en la programación en C++. No solo hemos podido implementar estos conocimientos en un escenario real, sino que también hemos enfrentado retos significativos en la resolución de problemas y en la optimización de nuestro código. A través del diseño, implementación y pruebas de distintos tipos de matrices y su interacción, hemos adquirido una comprensión más profunda de cómo aplicar eficientemente los conceptos de herencia y plantillas en la programación orientada a objetos, lo cual ha sido crucial para desarrollar aplicaciones robustas y eficaces