

PRÁCTICA 6: Operaciones con polinomios.

OBJETIVOS: Repaso de listas dinámicas. Objetos como datos miembro de otros objetos (introducción a la herencia). Objetos con partes dinámicas. Sobrecarga de operadores.

TEMPORIZACIÓN:

Publicación del enunciado: Semana del 21 de octubre.

Entrega: Semana del 11 de noviembre.

Límite de entrega (con penalización): Semana del 18 de noviembre.

BIBLIOGRAFÍA

Programación orientada a objetos con C++

Autor: Fco. Javier Ceballos

Editorial: RA-MA.

Se deben completar las clases `CMonomio`, `CTermino` y `CPolinomio`, cada una en su(s) propio(s) fichero(s), para realizar un programa que opere con polinomios y ponga a prueba todas las funciones implementadas. Se recomienda realizar la práctica siguiendo los pasos indicados al final del enunciado (NOTAS); destacamos especialmente que se debe implementar cuanto antes el operador de inserción (`<<`) de monomios, ya que facilitará enormemente la implementación de otros métodos.

La clase `CMonomio` encapsula un término de un polinomio. Un monomio con coeficiente 3 y exponente 2 equivale a $3x^2$. Todos los métodos de la clase son muy sencillos, así que pueden ser definidos `inline`. Esto quiere decir que el compilador podrá sustituir las llamadas a los métodos por el propio código de éstos.

```
class CMonomio
{
    private:

        double m_dCoeficiente;
        int m_nExponente;

    public:

        CMonomio(double dCoef=0, int nExp=0)
            : m_dCoeficiente(dCoef), m_nExponente(nExp) {}

        double GetCoef() const;           // <-- definir aquí
        int GetExp() const;               // <-- definir aquí

        void SetCoef(double dCoef);       // <-- definir aquí
        void SetExp(int nExp);            // <-- definir aquí

        CMonomio operator-() const
        { return CMonomio(-m_dCoeficiente, m_nExponente); }
};

ostream& operator<<(ostream& os, const CMonomio& Mono);
```

El constructor recibe dos parámetros con valores por defecto. Su lista de iniciadores copia los valores recibidos en los datos miembro, o más bien *construye* los datos miembro a partir de los valores recibidos. El cuerpo del constructor no hace nada (llaves vacías).

Se ha sobrecargado el operador *menos unario*, que devuelve un monomio cambiado de signo. Esto permitirá hacer cosas como:

```
CMonomio a(3,2), b;      // a == 3x^2
b = -a;                  // b == -3x^2
```

No hace falta definir un constructor copia ni sobrecargar el operador de asignación porque los objetos de esta clase no tienen partes dinámicas. La copia que se hará por defecto (miembro a miembro) es perfectamente válida.

Falta definir los métodos `Get...` y `Set...`; Serán métodos inline.

Para mostrar un monomio se puede sobrecargar el operador de inserción como se indica a continuación:

```
ostream& operator<<(ostream& os, const CMonomio& mono)
{
    os << showpos;          // Poner + para valores positivos
    if (!mono.GetExp())      // Si la x está elevada a 0,
        os << mono.GetCoef(); // vale 1 y se muestra sólo el coef.
    else                    // Si no...
    {
        if (mono.GetCoef() != 1) // Mostrar el coeficiente
            os << mono.GetCoef(); // sólo si es distinto de 1
        os << noshowpos;        // quitar + para valores positivos
        if (mono.GetExp() == 1) // Si el exponente es 1,
            os << 'x';          // basta "x"
        else                  // Si no,
            os << "x^" << mono.GetExp(); // hay que poner "x^..."
    }
    os << noshowpos;
    return os;              // Devolver ref. al ostream recibido
}                          // para permitir encadenamiento: cout << m1 << m2...
```

Escribir esta función en el fichero `.cpp` correspondiente.

La clase `CTermino` representa un monomio perteneciente a un polinomio. Los polinomios estarán formados por listas de objetos de la clase `CTermino`. Cada uno de estos objetos contendrá un monomio y un puntero al siguiente término.

```
class CTermino
{
private:
    CMonomio m_Monomio;      // Monomio contenido en el término
    CTermino *m_pSig;        // Puntero al siguiente término

public:
```

```
// Constructores ...

// Métodos Get:
double GetCoef() const { return m_Monomio.GetCoef(); }
int GetExp() const { return m_Monomio.GetExp(); }
CMonomio GetMono() const { return m_Monomio; }
CTermino * GetSig() const { return m_pSig; }

// Métodos Set:
void SetCoef(double dCoef) { m_Monomio.SetCoef(dCoef); }
void SetExp(int nExp) { m_Monomio.SetExp(nExp); }
void SetMono(const CMonomio& mono) { m_Monomio = mono; }
void SetSig(CTermino * pSig) { m_pSig = pSig; }
};
```

Se han definido métodos `Get...` y `Set...` para proporcionar una interfaz que permita acceder a los datos de la clase. Cabe destacar que algunos de ellos llaman directamente a los métodos correspondientes (`Get...` y `Set...`) de la clase `CMonomio` para el dato miembro `m_Monomio`. Este rodeo es propio de la *inclusión* (también conocida como *agregación* de objetos). En prácticas posteriores se estudiará un mecanismo más apropiado para este tipo de casos: la *herencia*.

Definir tres constructores para la clase `CTermino`. El primero de ellos debe construir el objeto a partir de un coeficiente, un exponente y un puntero al siguiente objeto `CTermino`. Los tres parámetros deben tener valores por defecto (0, 0 y `NULL` respectivamente). El segundo constructor debe construir el objeto a partir de un objeto monomio (este primer parámetro debe ser una referencia a un objeto `const`) y un puntero al siguiente objeto `CTermino`. El puntero deberá tener por defecto el valor `NULL`. El tercer constructor debe construir el objeto a partir de otro objeto `CTermino` (cuyo monomio será copiado) y un puntero al siguiente objeto `CTermino`. El puntero deberá tener por defecto el valor `NULL`. Al haber un valor por defecto para el puntero, este constructor servirá también como constructor copia.

Los tres constructores pueden ser definidos como el de la clase `CMonomio`: *iniciando* los datos miembro en su lista de iniciadores y dejando el cuerpo vacío. Al ser tan sencillos interesa definirlos `inline`.

La clase `CPolinomio` encapsula a una lista de términos ordenados de mayor a menor según el exponente (el objeto `CTermino` apuntado por `m_pCabecera` es el de mayor grado). No habrá dos términos con el mismo exponente ni términos con coeficiente nulo.

```
class CPolinomio
{
private:
    CTermino * m_pCabecera;    // Primer término (el de mayor grado)

public:
    // ...
};

ostream& operator<<(ostream& os, const CPolinomio& Poli);
```

Puesto que los polinomios sí contienen datos dinámicos propios, habrá que definir cuidadosamente los constructores (constructores con o sin parámetros y constructor copia), el destructor y el operador de asignación.

Se deberá definir:

- 1) Un constructor sin parámetros que inicie el puntero `m_pCabecera` a `NULL`.
- 2) Un constructor copia que copie un polinomio. Su parámetro debe ser una referencia a un objeto `const`. Implementar el constructor copia de forma que realice la misma iniciación del constructor sin parámetros, seguida de una llamada al operador de asignación indicado en el punto 7.
- 3) Un constructor que reciba un coeficiente y un exponente para construir un polinomio con un sólo término. El exponente deberá tener el valor 0 por defecto. Si el coeficiente recibido es 0, la lista deberá dejarse vacía (con `m_pCabecera` apuntando a `NULL`).
- 4) Un constructor que reciba un `vector<CMonomio>` y construya un polinomio formado por los monomios contenidos en dicho vector. Para ello puedo utilizar la sobrecarga del operador de inserción especificado un poco más adelante.
- 5) Un constructor que reciba un monomio para construir un polinomio con un sólo término. Si el coeficiente del monomio recibido es 0, la lista deberá dejarse vacía (con `m_pCabecera` apuntando a `NULL`).
- 6) Un destructor que libere la memoria dinámica del polinomio.
- 7) Un operador de asignación que copie un polinomio (utilice la sobrecarga del operador de inserción especificado a continuación). Tener en cuenta que el objeto destino puede contener datos anteriores. La memoria dinámica correspondiente a esos datos debe ser liberada antes de empezar a reservar memoria para copiar los nuevos datos.

El operador de inserción indicado en la página anterior, que sirve para enviar un polinomio a un flujo, llamará a la función `MostrarPoli` siguiente, que, a su vez, llama al operador de inserción de `CMonomio`:

```
void CPolinomio::MostrarPoli(ostream& os) const
{
    const CTermino * pPos = m_pCabecera;

    if (pPos)
        do
        {
            os << pPos->GetMono() << ' ';
            pPos = pPos->GetSig();
        }
        while (pPos);
}
```

```
else
    os << "0 ";
}
```

La introducción de datos en un polinomio se hará mediante una nueva sobrecarga del operador de inserción:

```
CPolinomio& CPolinomio::operator<<(const CMonomio& mono);
```

Al definirlo habrá que tener en cuenta que:

- 1) El monomio recibido puede tener coeficiente 0. En ese caso no habrá que añadirlo al polinomio.
- 2) Si la lista está vacía, bastará añadir un nuevo término con el puntero `m_pSig` a `NULL`. Deberá almacenarse en `m_pCabecera` la dirección de ese nuevo término.
- 3) Si no existe ningún término con el exponente del nuevo monomio, bastará insertar un nuevo término en la posición adecuada. Habrá que respetar el orden (de mayor a menor según el exponente).
- 4) Si ya existe un término con el exponente del nuevo monomio, habrá que sumar a su coeficiente el coeficiente del nuevo monomio (porque no puede haber exponentes repetidos). Si el resultado es 0 (los coeficientes se anulan), habrá que retirar el término de la lista.

Para poder realizar la inserción del nuevo término habrá que recorrer la lista con dos punteros que apunten a términos consecutivos de la lista, buscando un término con exponente menor o igual al del monomio que hay que insertar.

Este operador de inserción debe ser puesto a prueba con el siguiente código:

```
CPolinomio P, Q, R, S, T, U, V, W, X, Y, Z, N;
CMonomio m1, m2(2,2), m3(3,3), m4(4,4);

P << m1 << m2 << -m4 << m3;
Q << m4 << m1 << m2 << -m3;
R << m3 << m4 << -m2;
S << m2 << m3 << m4 << -m2;
T << -m2 << -m3 << -m4 << m3;
U << m2 << m3 << m4 << -m4;
V << m2 << m3 << m4 << -m2 << -m3 << -m4;
W << m2 << m3 << -m2 << -m3;
X << m2 << -m2;
Y << m2 << m3 << m4 << m2;
Z << -m2 << -m3 << -m4 << -m3;
N << m2 << m3 << m4 << m4;

cout << "P = " << P << endl
    << "Q = " << Q << endl
    << "R = " << R << endl
    << "S = " << S << endl
    << "T = " << T << endl
```

```
<< "U = " << U << endl  
<< "V = " << V << endl  
<< "W = " << W << endl  
<< "X = " << X << endl  
<< "Y = " << Y << endl  
<< "Z = " << Z << endl  
<< "N = " << N << endl;
```

Véase el resultado que se obtiene al final del enunciado.

Se deberá definir la siguiente función miembro pública:

```
int CPolinomio::Grado() const;
```

Esta función miembro devolverá el mayor exponente de todos los términos del polinomio. Se deberán sobrecargar además los siguientes operadores:

```
<, > y ==  
- unario  
+, -  
+=, -=  
[] y ()
```

Todos ellos deberán ser definidos como miembros de la clase CPolinomio. Los operadores de comparación se limitarán a comparar el grado de los polinomios. Los operadores += y -= pueden definirse fácilmente utilizando los demás (+, -, y =).

El operador [] recibirá un número de exponente (int) y devolverá el coeficiente (double) del término que tenga ese exponente. Si ningún término del polinomio tiene ese exponente, devolverá 0.

El operador () recibirá un valor de la x del polinomio (double) y devolverá el valor del polinomio (double) para ese valor de x. Este operador servirá para convertir a los polinomios en *funciones matemáticas*. Por ejemplo:

```
CPolinomio P;  
P << CMonomio(3,2) << CMonomio(-2,1);    // P == 3x^2 -2x  
double y, x=0.5;  
y = P(x);                                // y = 3*0.5^2 - 2*0.5 = -0.25
```

NOTAS:

Se sugiere el uso de la función pow.

Se recomienda realizar la práctica siguiendo los siguientes pasos:

- Implemente los constructores y destructores.
- Escriba una función main para probarlos.
- Implemente los operadores <<.
- Añada a main el código necesario para probarlos.
- Implemente la función Grado.
- Añada a main el código necesario para probarlos.
- Implemente el resto de operadores.

- Añada a `main` el código necesario para probarlos.

El objetivo de este desarrollo escalonado es compilar y ejecutar en cada paso para descubrir los errores lo antes posible.

Como ampliación opcional, puede añadir a la clase `CPolinomio` los operadores necesarios para la multiplicación de polinomios.

RESULTADOS DESPUÉS DE EJECUTAR LA PRÁCTICA:

***** PRACTICA 6 DE PROGRAMACION AVANZADA *****

Construcción de Polinomios

Probando el operador de inserción de monomios (<<)

P = $-4x^4 + 3x^3 + 2x^2$

Q = $4x^4 - 3x^3 + 2x^2$

R = $4x^4 + 3x^3 - 2x^2$

S = $4x^4 + 3x^3$

T = $-4x^4 - 2x^2$

U = $3x^3 + 2x^2$

V = 0

W = 0

X = 0

Y = $4x^4 + 3x^3 + 4x^2$

Z = $-4x^4 - 6x^3 - 2x^2$

N = $8x^4 + 3x^3 + 2x^2$

CPolinomio A = P (constructor copia)

A = $-4x^4 + 3x^3 + 2x^2$

Presione una tecla para continuar . . .

B = Q (operador de asignación)

B = $4x^4 - 3x^3 + 2x^2$

Presione una tecla para continuar . . .

Probando el constructor con un coef. y un exp.

C = 2.5

D = $2.5x^5$

Presione una tecla para continuar . . .

Probando el constructor con un monomio

E = $4x^4$

Presione una tecla para continuar . . .

Probando la función Grado.

El grado de U es: 3

Presione una tecla para continuar . . .

Probando el constructor con vector<CMonomio>

F = $4x^4 + 3x^3 + 2x^2$

Presione una tecla para continuar . . .

Probando el operador >

P = $-4x^4 + 3x^3 + 2x^2$

U = $3x^3 + 2x^2$

P es de mayor grado que U

Presione una tecla para continuar . . .

```
There are NO memory leaks!!
Presione una tecla para continuar . . .
```