Programación Avanzada
Proyecto 3: Polinomios

En esta práctica se desarrollará la clase `Polinomio` mediante la cual se podrán obtener todas las raíces (reales y complejas) de un polinomio con coeficientes complejos. Para desarrollar esta clase se requiere tener la clase `Complejo` de la tarea anterior completamente implementada y depurada.

Sugerencia: Después de implementar cada función o método, hacer un pequeño programa (en la función `main()`) para probar y depurar el método con múltiples casos. Asegurarse de que el método funciona correctamente antes de implementar el siguiente.

La clase `Polinomio` debe contener los siguientes miembros de datos **privados**:

- `Complejo *coef` - apuntador al arreglo de coeficientes (de tamaño arbitratio). El coeficiente `coef[i]` corresponderá al término de grado `i`.
- `int grado` - grado del polinomio (el arreglo `coef` debe tener capacidad para al menos `(grado+1)` elementos.

Sobrecargar el operador `<<` para poder enviar un objeto `Polinomio` a un flujo de salida de clase `ostream`. Imprimir el polinomio partiendo del término de mayor grado al de menor grado, saltándose aquellos términos con coeficiente igual a cero y dejando un espacio entre cada término y los símbolos `+` y `-`. Los coeficientes complejos deben aparecer entre paréntesis. Por ejemplo,

```
3x^3 + (4-2i)x^2 - 5
```

En este ejemplo, el coeficiente de primer grado es cero, mientras que el de segundo grado es complejo. Note también que `x^1` puede aparecer simplemente como `x`, mientras que `x^0` puede no aparecer en absoluto.

Además, la clase debe implementar los siguientes métodos públicos:

- `void destruye()` - Libera la memoria reservada para el arreglo de coeficientes (en caso de haber sido reservada) y reinicia los valores de `coef` y `grado` con `NULL` y `-1`, respectivamente.
- `bool crea(int g)` - Libera la memoria reservada (llamando a `destruye()`) y vuelve a reservar memoria para un polinomio de grado `g`. En caso de éxito, inicializa los coeficientes con ceros y devuelve `true`, de lo contrario devuelve `false`.

- `Polinomio(int g = 0)` - Constructor que crea un polinomio de grado `g` (llamando al método `crea()`). Es importante inicializar `coef` con `NULL` antes de llamar a `crea()` para evitar tratar de liberar memoria que no ha sido reservada.
- `Polinomio(Polinomio &p)` - Constructor que inicializa el polinomio haciendo una copia del polinomio `p` (llamar a `crea()` y luego copiar los coeficientes de `p`).
- `~Polinomio()` - Destructor de la clase `Polinomio`. Puede simplemente llamar a `destruye()`.
- `int getGrado()` - método para obtener el grado del polinomio.
- `Complejo &operator[](int i)` - método que devuelve una referencia al `i`-ésimo coeficiente del polinomio. El método debe verificar que `i` se encuentra en el rango correcto (entre 0 y `grado`, inclusive), de lo contrario debe devolver una referencia a un objeto de clase Complejo declarado de forma global y cuyo valor siempre sea cero.
- `Complejo evalua(Complejo x)` - evalúa el polinomio en el punto `x` y devuelve el resultado. Por razones de eficiencia, conviene no utilizar el operador de potencia definido para la clase Complejo, sino conservar en una variable el valor de la potencia y multiplicarlo por `x` en cada iteración.
- `void derivadaDe(Polinomio &p)` - hace el polinomio (desde el cual se llama al método) igual a la derivada del polinomio `p`.
- `bool raizNR(Complejo &x, double eps = 1e-12, int maxiter = 100)` - Este método implementará el algoritmo de Newton-Raphson para aproximar una raíz del polinomio a partir de una estimación inicial `x`. El método iterará hasta que se llegue al máximo de iteraciones (`maxiter`), o hasta que el error sea menor que `eps`. El error se calcula en cada iteración como $|x_n - x_a|/|x_n|$ donde x_n es la nueva estimación y x_a la anterior. La función devuelve un valor booleano indicando si encontró una raíz (`true`) o no (`false`). Esto puede saberse evaluando el polinomio en `x` y verificando que la magnitud del resultado es cercana a cero (digamos, menor que `eps`).
- `void reduceRaiz(Complejo r)` - Utiliza división sintética para dividir el polinomio entre un binomio de la forma $(x-r)$. El algoritmo para realizar eso es el siguiente, para un polinomio de grado n con coeficientes a_i:

  1. Inicializar un complejo c con cero.
  2. Para $i = 0, \ldots, n$, hacer lo siguiente
     (a) Hacer $t = a_{n-i}$
     (b) Hacer $a_{n-i} = c$
     (c) Hacer $c = cr + t$

3. Decrementar n.

- `Complejo *raices(double eps = 1e-12, int maxiter = 100)` - Este método encontrará todas las raíces del polinomio y las devolverá en un arreglo. El procedimiento es el siguiente:

  1. Declarar un polinomio `p` e inicializarlo con una copia de `*this` usando el constructor adecuado.
  2. Declarar un apuntador a complejo `raiz` y asignarle memoria para un arreglo de `grado` elementos.
  3. Para `i` desde 0 hasta `(grado-1)`, hacer lo siguiente:
     (a) Iniciar un ciclo (e.g., `do-while`)
     (b) Asignar a un complejo `r` un valor real aleatorio
     (c) Con un 50% de probabilidad, asignarle un valor aleatorio a la parte imaginaria de `r`
     (d) Llamar al método `p.raizNR()` pasándole `r` como estimación inicial, con el umbral de error y máximo de iteraciones que se recibieron como argumentos. Si el método devuelve falso, continuar con el ciclo `do-while` para probar con otro valor inicial.
     (e) Si se encontró una raíz (`raizNR()` devolvió verdadero), guardarla en `raiz[i]` y reducir `p` dividiéndolo entre $(x - r)$, usando el método `p.reduceRaiz()`.
  4. Devolver el apuntador `raiz`.

Implementar un programa en la función `main()` para obtener las raíces de varios polinomios dados como ejemplo, donde algunos de ellos tengan coeficientes únicamente reales, mientras que otros tengan coeficientes complejos. Por ejemplo, las raíces del polinomio que se muestra arriba, $p(x) = 3x^3 + (4 - 2i)x^2 - 5$, son $-1.293 + 1.2i$, $0.853 + 0.107i$, y $-0.893 - 0.641i$, aproximadamente.