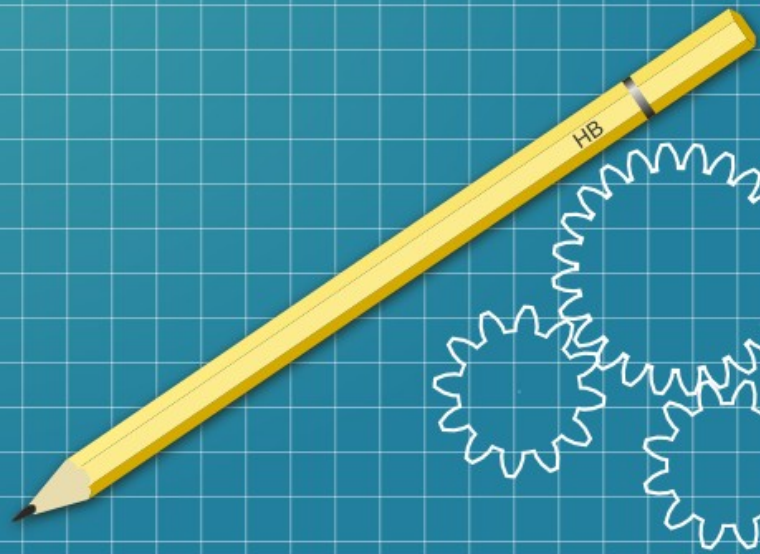
Programación orientada a objetos segunda parte

```
#include<stdio.h>

int main(){
  int a = 10;
  int b;
  b ~= a;
  return 0;
}
```

Palomeque Nestor Levi



Composición de clases

La composición de clases ocurre cuando una clase contiene objetos de otras clases como miembros. Este enfoque permite crear clases complejas que reutilizan funcionalidad ya implementada. Por ejemplo:

```
class Motor {
    // Definición de la clase Motor
};


class Auto {
 public:
    Motor motor; // Auto tiene un Motor (composición)
};
```

Composición de clases, ejemplo

```
class Direccion {
private:
    std::string calle;
    std::string ciudad;
public:
    Direccion(const std::string& c, const std::string& ci) : calle(c), ciudad(ci) {}    // Constructor

    std::string getCalle() const { return calle; }      // Métodos para obtener la calle y la ciudad
    std::string getCiudad() const { return ciudad; }
};
class Persona {
private:
    std::string nombre;
    Direccion direccion; // Miembro de tipo Direccion
public:
    // Constructor de Persona, llamando directamente al constructor de Direccion
    Persona(const std::string& n, const std::string& calle, const std::string& ciudad)
        : nombre(n), direccion(calle, ciudad) {} // Llamada al constructor de Direccion

    void mostrarDireccion() const {     // Método para mostrar la dirección de la persona
        std::cout << "Nombre: " << nombre << std::endl;
        std::cout << "Direccion: " << direccion.getCalle() << ", " << direccion.getCiudad() << std::endl;
    }
};
```

Crea una clase **Direccion** con atributos calle y ciudad. Luego, crea una clase **Persona** que tenga como miembro un objeto de tipo **Direccion**. Muestra la dirección de la persona usando un método en la clase **Persona**.

```
int main() {
    // Crear una persona con su dirección
    Persona persona("Juan Perez", "Calle 123", "Ciudad ABC");

    // Mostrar la dirección de la persona
    persona.mostrarDireccion();

    return 0;
}
```

Plantillas de clases (Templates)

Las plantillas permiten crear clases o funciones genéricas que funcionan con diferentes tipos de datos, sin duplicar código.
Las plantillas son muy útiles cuando el comportamiento es independiente del tipo, pero el tipo real de los datos varía.

Veamos el siguiente ejemplo:

```cpp
#include <iostream>

template<typename T>
class Caja {
    T valor; // El miembro 'valor' es de tipo genérico T
public:
    // Función para asignar valor
    void setValor(T v) {
        valor = v;
    }

    // Función para obtener el valor
    T getValor() const {
        return valor;
    }
};
```

```cpp
int main() {
    // Una Caja que almacena enteros
    Caja<int> cajaEntera;
    cajaEntera.setValor(100);   // Asignamos el valor 100
    std::cout << cajaEntera.getValor() << std::endl;  // Salida: 100

     // Una Caja que almacena strings
    Caja<std::string> cajaString;
    cajaString.setValor("Hola");
    std::cout << cajaString.getValor() << std::endl;  // Salida: Hola

    return 0;
}
```

Biblioteca estándar de C++ (STL)

La biblioteca estándar incluye clases como std::string, std::vector, entre otras, que facilitan el manejo de cadenas, listas, y más:

- std::string: Manejo de cadenas de caracteres.
- std::vector: Arreglo dinámico de elementos.

```
std::vector<int> numeros = {1, 2, 3};
std::string texto = "Hola";
```

Funciones amigas (friend)

Es una función que, aunque no es miembro de una clase, tiene acceso a sus miembros privados y protegidos, y se declara dentro de la clase utilizando la palabra clave **friend**. Aunque no pertenece a la clase, tiene privilegios especiales para acceder a todos sus miembros, lo que la hace útil para facilitar la interacción entre funciones y clases. Se puede declarar fuera, dentro de la clase o miembro de otra clase, y es particularmente útil en la sobrecarga de operadores. Aunque permiten un acceso conveniente a los detalles internos de una clase, es importante tener cuidado al usarlas, ya que pueden violar el principio de encapsulación.

```
class Caja {
    int valor;        //valor es un miembro privado
public:
    Caja(int v) : valor(v) {}


     // función amiga
    friend void mostrarValor(const Caja& c){
      std::cout << c.valor << std::endl;
   }
};
```

```
class Caja {
    int valor; // Miembro privado
public:
    // Constructor que inicializa el valor
    Caja(int v) : valor(v) {}

    // Declaración de la función amiga
    friend void mostrarValor(const Caja& c); // Aquí se declara como amiga
};

// Definición de la función amiga global
void mostrarValor(const Caja& c) {
    std::cout << "El valor de la caja es: " << c.valor << std::endl; // Acceso a
'valor'
}
```

Herencia de clases en C++

La herencia es un concepto fundamental en la programación orientada a objetos (POO) que permite que una clase (llamada clase derivada o subclase) herede atributos y métodos de otra clase (llamada clase base o superclase). Esto promueve la reutilización del código y establece una relación jerárquica entre las clases.

Herencia pública:

- Los miembros públicos de la clase base se convierten en miembros públicos de la clase derivada, y los miembros protegidos de la clase base se convierten en miembros protegidos de la clase derivada.
- Los miembros privados de la clase base no son accesibles directamente desde la clase derivada.

Herencia protegida:

- Los miembros públicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada.
- Esto significa que solo las clases derivadas pueden acceder a esos miembros, pero no los objetos creados a partir de esas clases derivadas.

Herencia privada:

- En la herencia privada, los miembros públicos y protegidos de la clase base se convierten en miembros privados de la clase derivada.
- Esto significa que ni los objetos de la clase derivada ni las clases derivadas de la clase derivada pueden acceder a esos miembros.

Nota:

Siempre que la clase base tenga un constructor que requiera parámetros, debes llamarlo desde el constructor de la clase derivada utilizando la lista de inicialización.

Esto garantiza que los miembros de la clase base se inicialicen correctamente antes de que se ejecute el cuerpo del constructor de la clase derivada.

Herencia pública

En el siguiente ejemplo crearemos la clase **Figura** (geométrica) para luego crear la clase derivada **Triangulo**.

```cpp
#include <iostream>
#include <string>

class Figura {
protected:
    std::string nombre; // Atributo protegido
private:
    double perimetro;   // Atributo privado
public:
    int ID;  // Número de identificación para la figura

    // Constructor
    Figura(int ID, const std::string& nombre, double perimetro)
        : ID(ID), nombre(nombre), perimetro(perimetro) {}

    void mostrarNombre() const {
        std::cout << "Nombre de la figura: " << nombre << std::endl;
    }

    void mostrarPerimetro() const {
        std::cout << "Perímetro de la figura: " << perimetro << std::endl;
    }
};
```

```cpp
class Triangulo : public Figura { // Herencia pública
private:
    double base;
    double altura;
public:
    Triangulo(int ID, const std::string& nombre, double perimetro, double base, double altura)
        : Figura(ID, nombre, perimetro), base(base), altura(altura) {} // Constructor

    double calcularArea() const {
        return (base * altura) / 2; // Método para calcular el área del triángulo
    }
    void mostrarDetalles() const {
       // mostrarNombre(); // Llama al método de la clase base
        std::cout << "Nombre de la figura: " << nombre << std::endl;
        mostrarPerimetro(); // Muestra el perímetro
        std::cout << "Área del triángulo: " << calcularArea() << std::endl; // Muestra el área
    }
};
int main() {
    Triangulo tri(1, "Triángulo Equilátero", 15.4, 5.0, 4.0);
    tri.mostrarDetalles(); // Muestra el nombre, el perímetro y el área del triángulo
    std::cout << "Identificación de la figura: " << tri.ID << std::endl; // Muestra el ID
    return 0;
}
```

Herencia protegida

En el siguiente ejemplo crearemos la clase **Figura** (geométrica) para luego crear la clase derivada **Triangulo**.

```cpp
#include <iostream>
#include <string>

class Figura {
protected:
    std::string nombre; // Atributo protegido
private:
    double perimetro;   // Atributo privado
public:
    int ID;  // Número de identificación para la figura

    // Constructor
    Figura(int ID, const std::string& nombre, double perimetro)
        : ID(ID), nombre(nombre), perimetro(perimetro) {}

    void mostrarNombre() const {
        std::cout << "Nombre de la figura: " << nombre << std::endl;
    }

    void mostrarPerimetro() const {
        std::cout << "Perímetro de la figura: " << perimetro << std::endl;
    }
};
```

```cpp
class Triangulo : protected Figura { // Herencia protegida
private:
    double base;
    double altura;
public:
    Triangulo(int ID, const std::string& nombre, double perimetro, double base, double altura)
        : Figura(ID, nombre, perimetro), base(base), altura(altura) {} // Constructor

    double calcularArea() const {
        return (base * altura) / 2; // Método para calcular el área del triángulo
    }
    void mostrarDetalles() const {
      // mostrarNombre(); // Llama al método de la clase base
        std::cout << "Nombre de la figura: " << nombre << std::endl;
        mostrarPerimetro(); // Muestra el perímetro
        std::cout << "Área del triángulo: " << calcularArea() << std::endl; // Muestra el área
    }
};
int main() {
    Triangulo tri(1, "Triángulo Equilátero", 15.4, 5.0, 4.0);
    tri.mostrarDetalles(); // Muestra el nombre, el perímetro y el área del triángulo
    //std::cout << "Identificación de la figura: " << tri.ID << std::endl; //Error: ID es protegido
    return 0;
}
```

Herencia privada

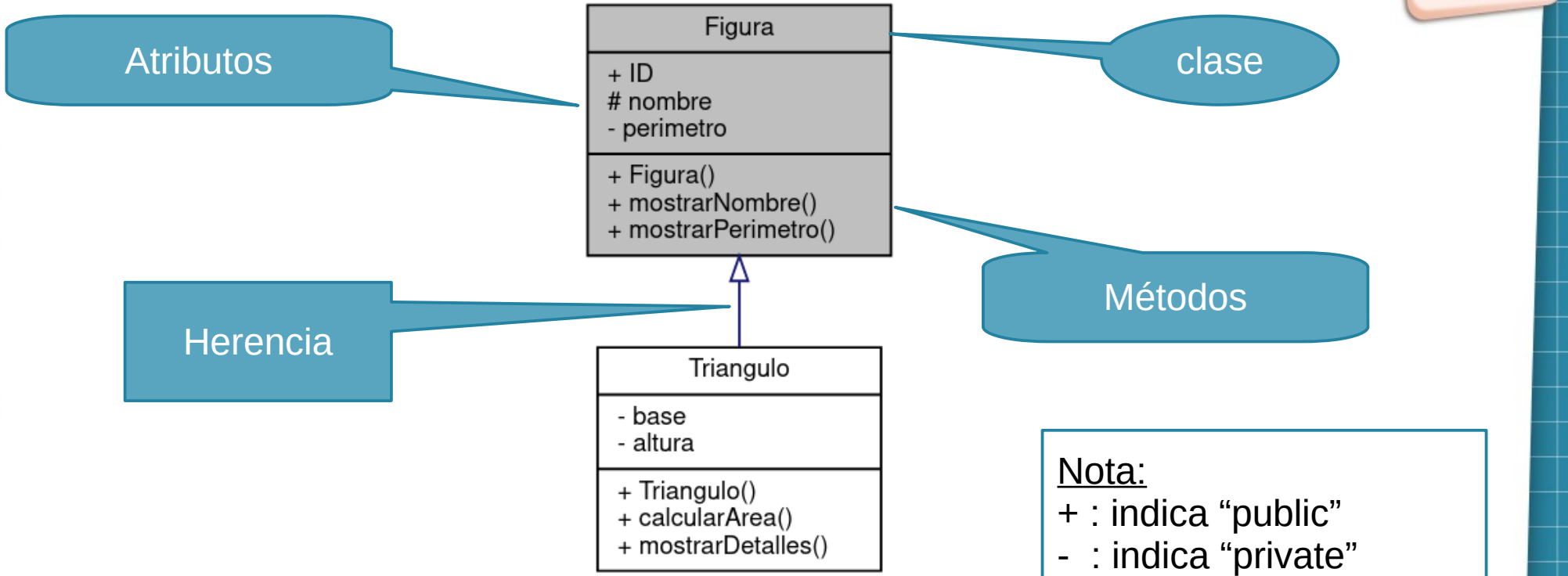
En el siguiente ejemplo crearemos la clase **Figura** (geométrica) para luego crear la clase derivada **Triangulo**.

```cpp
#include <iostream>
#include <string>

class Figura {
protected:
    std::string nombre; // Atributo protegido
private:
    double perimetro;   // Atributo privado
public:
    int ID;  // Número de identificación para la figura

    // Constructor
    Figura(int ID, const std::string& nombre, double perimetro)
        : ID(ID), nombre(nombre), perimetro(perimetro) {}

    void mostrarNombre() const {
        std::cout << "Nombre de la figura: " << nombre << std::endl;
    }

    void mostrarPerimetro() const {
        std::cout << "Perímetro de la figura: " << perimetro << std::endl;
    }
};
```

```cpp
class Triangulo : private Figura { // Herencia privada
private:
    double base;
    double altura;
public:
    Triangulo(int ID, const std::string& nombre, double perimetro, double base, double altura)
        : Figura(ID, nombre, perimetro), base(base), altura(altura) {} // Constructor

    double calcularArea() const {
        return (base * altura) / 2; // Método para calcular el área del triángulo
    }
    void mostrarDetalles() const {
       // mostrarNombre(); // Llama al método de la clase base
        std::cout << "Nombre de la figura: " << nombre << std::endl;
        mostrarPerimetro(); // Muestra el perímetro
        std::cout << "Área del triángulo: " << calcularArea() << std::endl; // Muestra el área
    }
};
int main() {
    Triangulo tri(1, "Triángulo Equilátero", 15.4, 5.0, 4.0);
    tri.mostrarDetalles(); // Muestra el nombre, el perímetro y el área del triángulo
    //std::cout << "Identificación de la figura: " << tri.ID << std::endl; //Error: ID es privado
    return 0;
}
```

Diagramas de clases UML

Un diagrama UML (Unified Modeling Language) de clases es una representación gráfica que muestra las relaciones entre clases, atributos, métodos y herencias. Normalmente incluye: nombre de la clase, atributos, métodos y relaciones como herencia.



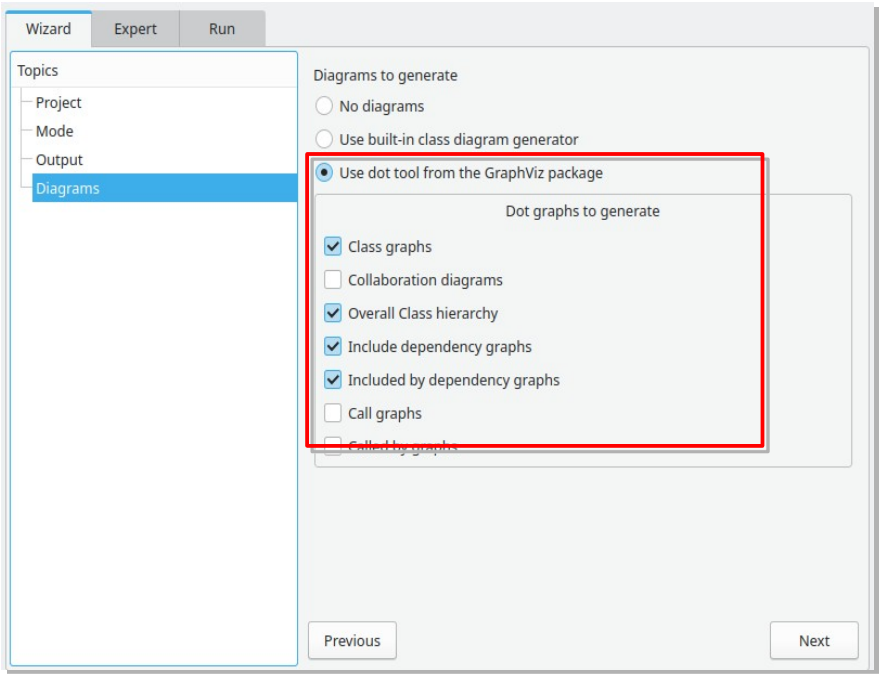
Nota:
+ : indica “public”
- : indica “private”
: indica “protected”

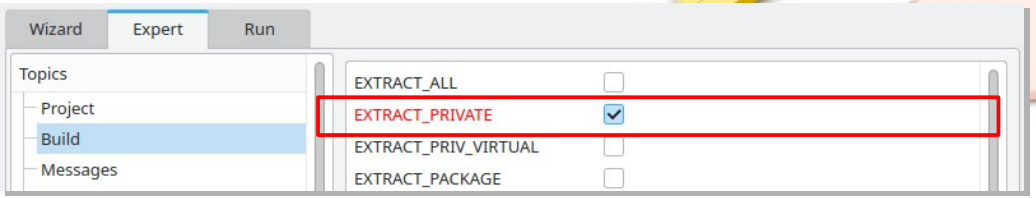
Diagramas de clases con Doxygen

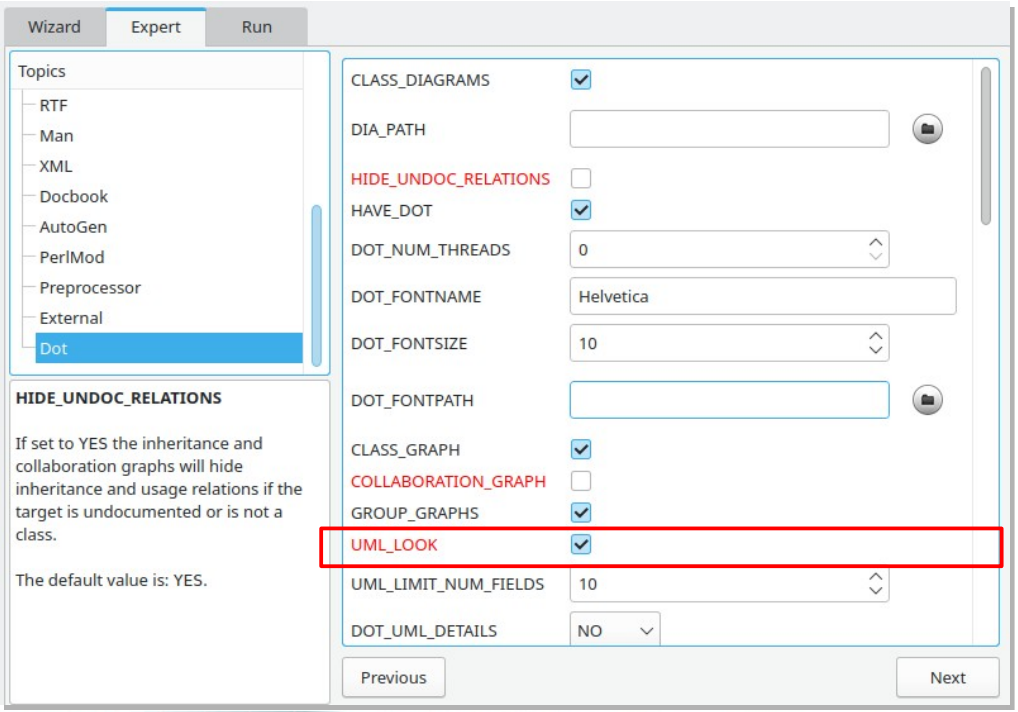
Para generar la documentación gráfica en doxygen debemos primero instalar la herramienta necesaria con el comando:

`sudo apt-get install graphviz`

Luego verificar las siguiente opciones:







Sobrecarga de operadores en C++

La sobrecarga de operadores permite redefinir el comportamiento de operadores (como +, -, ==) para tipos de datos definidos por el usuario. Esto se logra mediante una "función operador".

Por ejemplo, para sumar dos objetos de una clase:

```cpp
#include <iostream>
class Complejo {
    double real, imaginario;

public:
    // Constructor
    Complejo(double r = 0, double i = 0) : real(r), imaginario(i) {}

    // Sobrecarga del operador * para multiplicar con un número real
    Complejo operator*(double escalar) const {
        Complejo resultado(real * escalar, imaginario * escalar);
        return resultado;
    }

    // Sobrecarga del operador * para multiplicar con otro objeto Complejo
    Complejo operator*(const Complejo& otro) const {
        double r = (real * otro.real - imaginario * otro.imaginario);
        double i = (real * otro.imaginario + imaginario * otro.real);
        return Complejo(r, i);
    }

    // Método para mostrar el número complejo
    void mostrar() const {
        std::cout << "(" << real << " + " << imaginario << "i)" << std::endl;
    }
};
```

```cpp
int main() {
    Complejo c1(3, 4);  // Número complejo 3 + 4i
    Complejo c2(1, 2);  // Número complejo 1 + 2i

    // Multiplicar c1 por un escalar
    Complejo resultado1 = c1 * 2.0;  // Multiplica por 2.0
    resultado1.mostrar();            // Muestra (6 + 8i)

    // Multiplicar c1 por c2
    Complejo resultado2 = c1 * c2;    // Multiplica (3 + 4i) * (1 + 2i)
    resultado2.mostrar();            // Muestra (-5 + 10i)

    return 0;
}
```

Aclaraciones para sobrecarga de operadores

Definición como método de instancia

Cuando definimos el operador **+** como un método de la clase, estamos diciendo que el operador se aplica a un objeto de la clase. En este caso, el objeto que invoca el método se considera el primer operando, mientras que el segundo operando se pasa como parámetro.

```
Complejo operator+(const Complejo& otro) const {
    return Complejo(real + otro.real, imaginario + otro.imaginario);
}
```

Definición como función libre

Si decidimos definir el operador **+** como una función libre (como en el caso del operador **<<** empleado a continuación), necesitamos proporcionar ambos operandos como parámetros:

```
friend Complejo operator+(const Complejo& actual, const Complejo& otro) {
    return Complejo(actual.real + otro.real, actual.imaginario + otro.imaginario);
}
```

En ambos casos debe usar **Complejo c3 = c1 + c2;** en el main.

Algo sobre la entrada y salida estándar

std::ostream es una clase en la biblioteca estándar de C++ que se utiliza para manejar flujos de salida. Es fundamental para la manipulación de datos que deseas imprimir o enviar a un destino, como la consola o un archivo.

std::cout es un objeto de tipo **std::ostream** que representa la salida estándar (normalmente la consola).

std::istream es una **clase** en la biblioteca estándar de C++ que se utiliza para manejar flujos de entrada.

La instancia más común de **std::istream** es **std::cin**, que representa la entrada estándar (normalmente el teclado).

De esta forma los operadores **<<** y **>>** están sobrecargados para los flujos de salida y entrada estándar en C++, específicamente para std::cout y std::cin

Sobrecarga de operadores de inserción y extracción de flujos

Los operadores de inserción (<<) y extracción (>>) se utilizan para manejar la entrada y salida estándar (usualmente con std::cin y std::cout). Sobrecargarlos facilita imprimir o leer objetos de clases personalizadas.

Ejemplo de sobrecarga de << para imprimir un objeto:

```
#include <iostream>

class Punto {
    int x, y;
public:
    Punto(int x, int y) : x(x), y(y) {}

    // Sobrecarga de <<
    friend std::ostream& operator<<(std::ostream& out, const Punto& p) {
        out << "(" << p.x << ", " << p.y << ")";
        return out;
    }
};

int main() {
    Punto p(1, 2);
    std::cout << p; // Imprime: (1, 2)
}
```

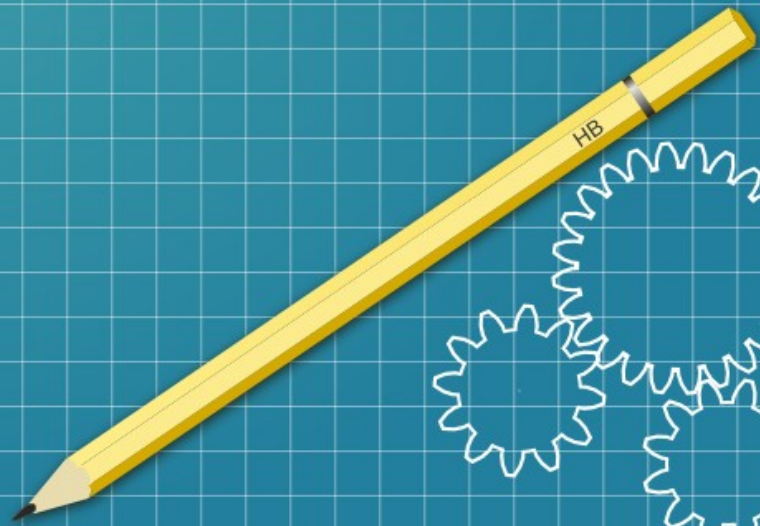
Definición de la sobrecarga:

- **friend**: La palabra clave friend permite que esta función acceda a los miembros privados de la clase Punto. Esto es necesario porque x e y son privados y no se pueden acceder directamente fuera de la clase.
- El tipo de retorno es **std::ostream&**, que es una referencia al flujo de salida.
- **std::ostream& out**: una referencia al flujo de salida donde se imprimirá el objeto.
- **const Punto& p**: una referencia constante al objeto Punto que se va a imprimir.

Implementación de la función:

- Se utiliza **out** para construir la cadena que representa el objeto Punto. Utiliza el flujo de salida out para imprimir las coordenadas en el formato (x, y).





HB