Programación Orientada a Objetos

UNEG

Programación Orientada a Objetos

- ¿Cómo llevar a código todos los conceptos, principios y términos técnicos de la orientación a objetos?
- Lenguajes de programación más populares: C++, Java y C#.
- Hay lenguajes puros (casados con los conceptos de OO), e híbridos (permiten hacer cosas no OO).
- Es importante evaluar la plataforma de desarrollo a utilizar basado en: entorno operativo, amigabilidad, soporte, tiempo de compilación, biblioteca de clases predefinida, costo (adquisición, instalación, mantenimiento, etc.), experiencia, etc.
- Usar un lenguaje orientado a objetos no necesariamente implica hacer programación orientada a objetos. La clave está en respetar los principios.

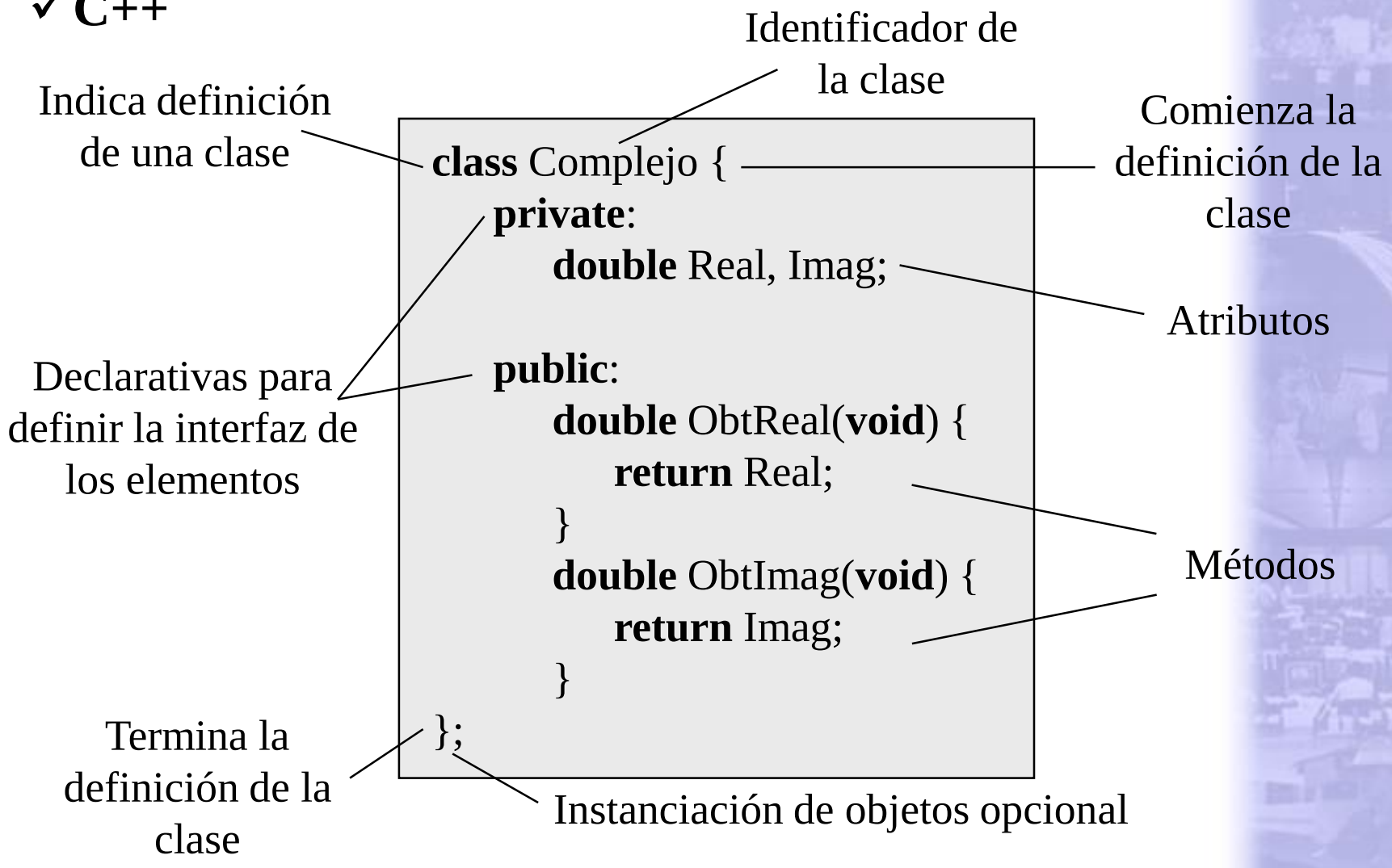
Programación Orientada a Objetos

✓ **C++**

- Compilado => permite generar programas ejecutables.
- No es puro porque viene del lenguaje C y mantiene todos los conceptos no orientado a objetos de C.
- Bajo nivel de portabilidad, alta dependencia en la plataforma operativa.
- Sintaxis simple y semántica compleja.
- Muy rico conceptualmente en lo que a orientación a objetos se refiere (manejo de herencia múltiple y sobrecarga de operadores por ejemplo).
- Extensión de los archivos de código fuente: “cpp”.

Definición de Clases

✓ **C++**



Definición de Clases

✓ **C++**

- Las declarativas de interfaz pueden aparecer más de una vez y en cualquier orden. Si no se indica nada es privada por defecto.
- El cuerpo de los métodos se puede definir tanto dentro (inline) como fuera (outline) de la definición de la clase. Hay una diferencia en la llamada del mensaje o ejecución del método:

  inline => se hace una copia fiel del código del cuerpo del método; código más rápido pero más grande

  outline => se agrega una referencia donde se encuentrael código del cuerpo del método; código más lento pero más pequeño

Definición de Clases

✓ **C++**

```
class Complejo {
        double Real;
    public:
         double ObtReal(void);
    private:
        double Imag;
    public:
        double ObtImag(void);
};
```

El cuerpo del método no está aquí

Indica relación de dependencia con una clase

```
double Complejo::ObtReal(void)
{
    return Real;
}

double Complejo::ObtImag(void)
{
    return Imag;
}
```

Definición de Clases

✓ **C++**

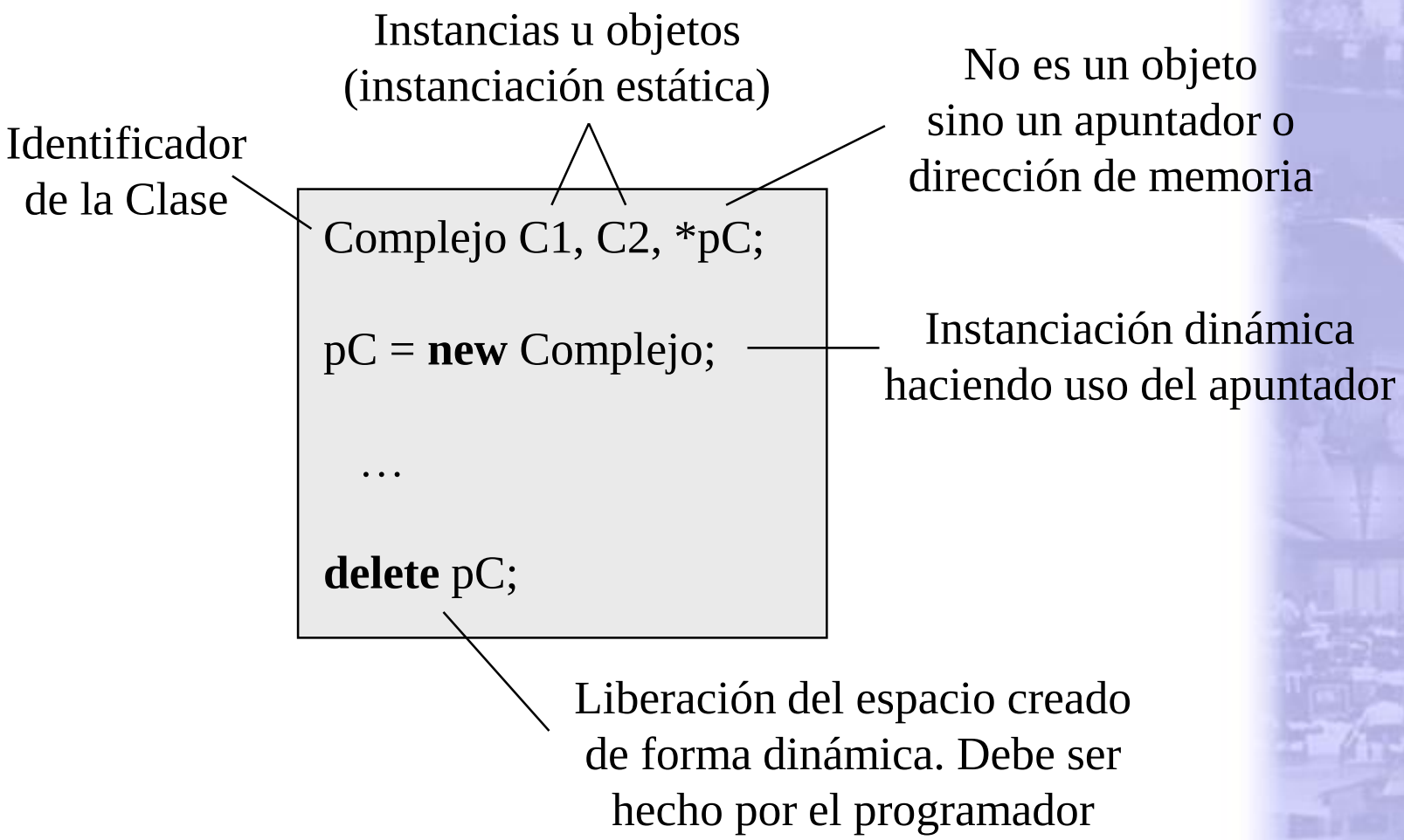
- El cuerpo de los métodos *inline* debe ser simple, sin estructuras de control complejas (ciclos, ruptura, condicionales múltiples).
- Se puede forzar la declaración de los métodos *inline* fuera de la definición de la clase. Permite mantener las ventajas de los métodos *inline* escondiendo el código a terceros.

```
inline double Complejo::ObtReal(void) {
    return Real;
}

inline double Complejo::ObtImag(void) {
    return Imag;
}
```

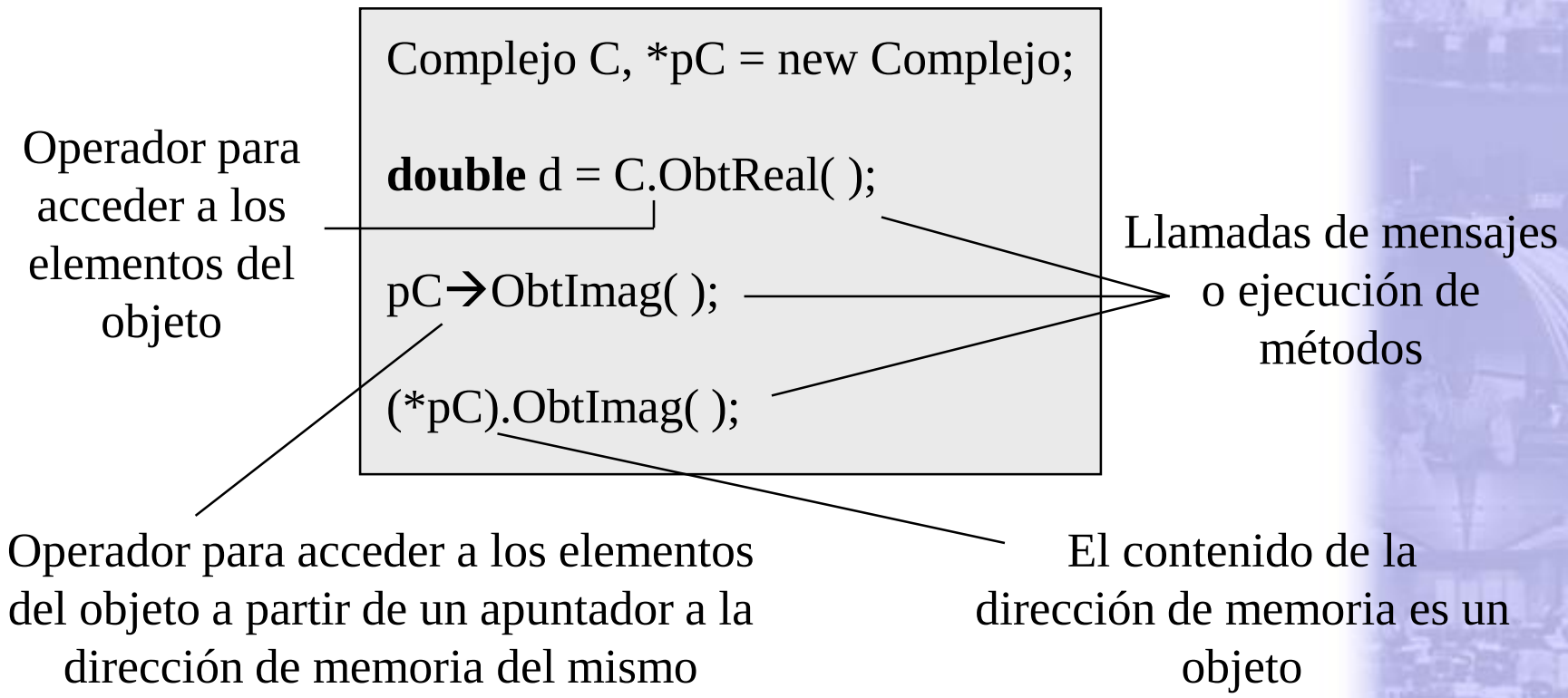
Instanciación de Objetos

✓ **C++**



Acceso a Elementos de la Clase

✓ **C++**



NOTA: Recordar que los permisos de acceso a los elementos están regidos por la interfaz asociada en la definición de la clase.

Constructor

- Basado en el principio de persistencia.
- Es un método particular que es ejecutado automáticamente durante la instanciación del objeto (tanto estática como dinámicamente).
- Es útil para definir el estado inicial del objeto y realizar otras inicializaciones necesarias.
- Como método/operación tiene las siguientes características:
  - Es opcional.
  - Debe ser público.
  - Se identifica de la misma forma como se identifica la clase.
  - No se le especifica tipo de retorno.
  - Puede tener cualquier número de parámetros.

Constructor

✓ **C++**

```
class Complejo {
    private:
        double Real, Imag;

    public:
        Complejo(void) {
            Real = Imag = 0.0;
        }
        double ObtReal(void) {
            return Real;
        }
        double ObtImag(void) {
            return Imag;
        }
};
```

No está permitido la inicialización de variables en este contexto

Un constructor para la clase

NOTA: Si no se especifica un constructor la data del objeto queda sin inicializar (embasurada).

Polimorfismo

- Basado en el principio que lleva el mismo nombre.
- Definir dos o más métodos con el mismo nombre, tipo de retorno e interfaz, pero diferente descripción de parámetros (cantidad y/o tipo de los mismos). Los parámetros determinan el contexto.
- No hay límite sobre la cantidad de métodos definidos de forma polimórfica.
- No es una redefinición, cada método se trata de forma independiente (declaración, definición y llamada).
- Un caso particular es hacer polimorfismo sobre el constructor. Permite dar al usuario diferentes formas de inicializar un objeto durante la instanciación del mismo.

Polimorfismo

✓ **C++**

```
class Complejo {
    private:
        double Real, Imag;

    public:
        Complejo(void) {
            Real = Imag = 0.0;
        }
        Complejo(double R, double I) {
            Real = R, Imag = I;
        }
        …
} C1, C2(1.0, -1.0);
```

Polimorfismo sobre el constructor

Dos objetos instanciados cada uno de los cuales llama a un constructor diferente

Inicialización por defecto

- Permite especificar un valor inicial por defecto a los parámetros de un método para los casos en los cuales el usuario obvia el parámetro.
- Da al usuario la idea de que los parámetros son opcionales.
- Cuando hay más de un parámetro, el orden de asignación entre el valor por defecto y el parámetro es de izquierda a derecha. No pueden haber saltos en esta relación de asignación.
- Es útil para reducir el número de métodos de la clase (simplificar el uso del polimorfismo).
- Es válido para cualquier método, en particular para los constructores.

Inicialización por defecto

✓ **C++**

```
class Complejo {
    private:
        double Real, Imag;

    public:
        Complejo(double R = 0.0, double I = 0.0) {
            Real = R, Imag = I;
        }
        …
} C1, C2(1.0, -1.0), C3(1.0);
```

Valor por defecto del parámetro

Tres objetos instanciados, todos hacen la llamada al mismo constructor

Inicialización por defecto

✓ **C++**

```
class Complejo {
    …
    public:
        Complejo(double = 0.0, double = 0.0);
        …
};
…
Complejo::Complejo(double R, double I) {
    Real = R; Imag = I;
}
```

NOTA: Cuando el cuerpo del constructor está fuera de la definición de la clase, las inicializaciones por defecto se especifican dentro de la clase en la declaración del método y no en su definición.

Destructor

- Basado en el principio de persistencia.
- Es un método particular que es ejecutado automáticamente durante la destrucción o pérdida de alcance del objeto (tanto estática como dinámicamente).
- Es útil para realizar código necesario cuando el objeto ya no va a ser mas utilizado, como por ejemplo la liberación o recuperación de espacios de memoria.
- No aplica en lenguajes con recuperación automática de memoria.
- Como método/operación tiene las siguientes características:
  - Es opcional.
  - Debe ser público.
  - No se le especifica tipo de retorno.
  - No tiene parámetros.

Destructor

✓ **C++**

El destructor se llama igual que la clase pero con el prefijo ~

Expresa que el método tiene un cuerpo vacío o no tiene código asociado

```
class Complejo {
    private:
        double Real, Imag;

    public:
        Complejo(double R = 0.0, double I = 0.0) {
            Real = R, Imag = I;
        }
        ~Complejo(void) { }
        …
};
```

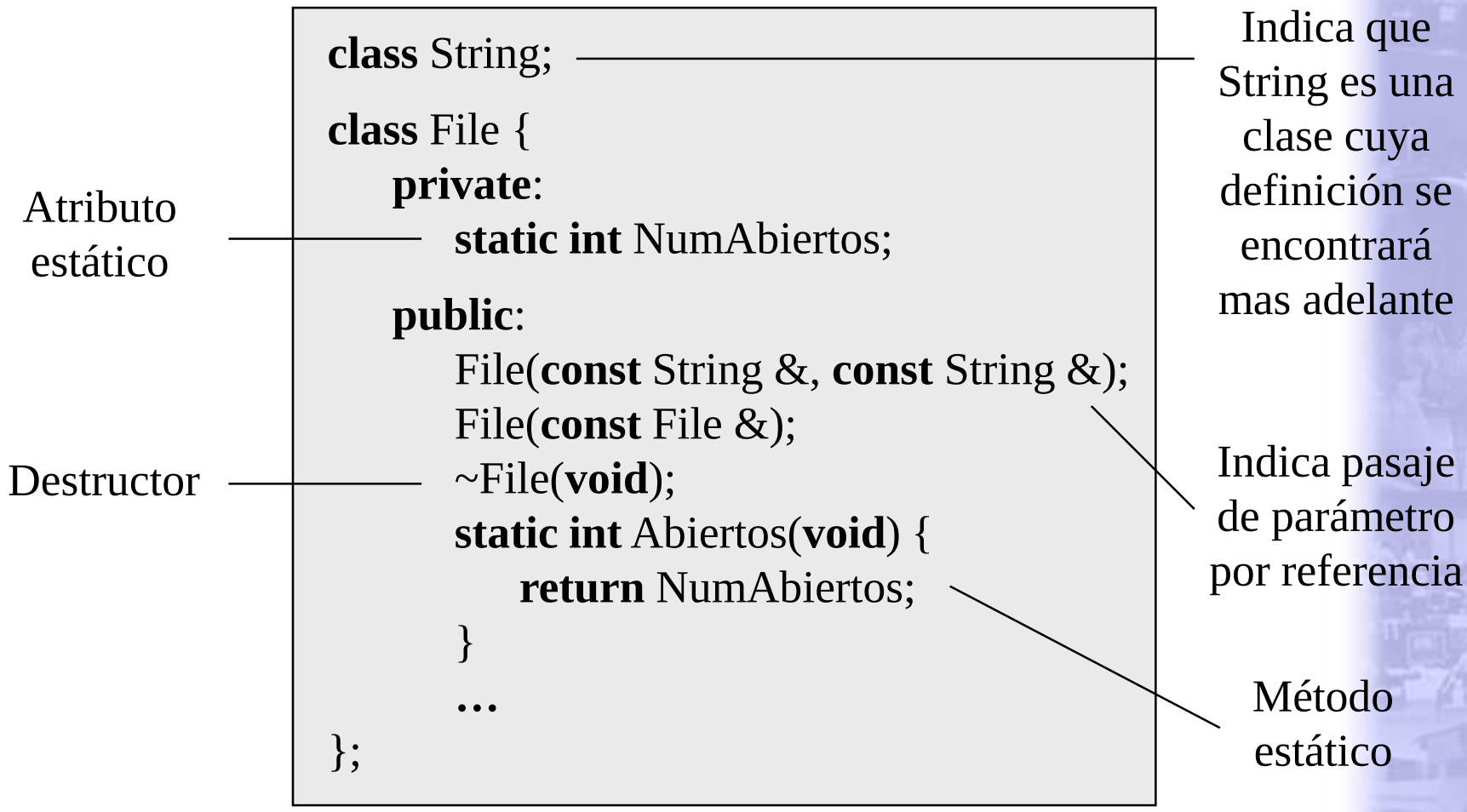
NOTA: La destrucción ocurre cuando el objeto pierde su alcance de vida. En el caso de la instanciación dinámica, esto ocurre cuando se hace la operación **delete**

Elementos estáticos

- Basado en el principio de concurrencia.
- Permite que todos los objetos instanciados de una misma clase y que están activos en un mismo instante, compartan la misma información asociada a uno o varios de sus atributos y/o compartan el mismo código asociado a uno o varios de sus métodos.
- La idea es forzar a todos los objetos a ver / usar el mismo elemento.
- Un ejemplo típico de Java / C#, es el caso particular del método que representa el programa principal (*main* / *Main*); este debe ser estático porque no tiene sentido dos instancias de objetos cada uno de los cuales con un programa principal de ejecución; todas las instancias comparten el mismo método.

Elementos estáticos

✓ **C++**



Elementos estáticos

✓ **C++**

```
int File::NumAbiertos = 0;

File::File(const String &N, const String &M) {
    …
    NumAbiertos++;
}

File::~File(void) {
    …
    NumAbiertos--;
}
…
```

Asignación del valor inicial al atributo estático, nótese que está fuera de la definición de la clase

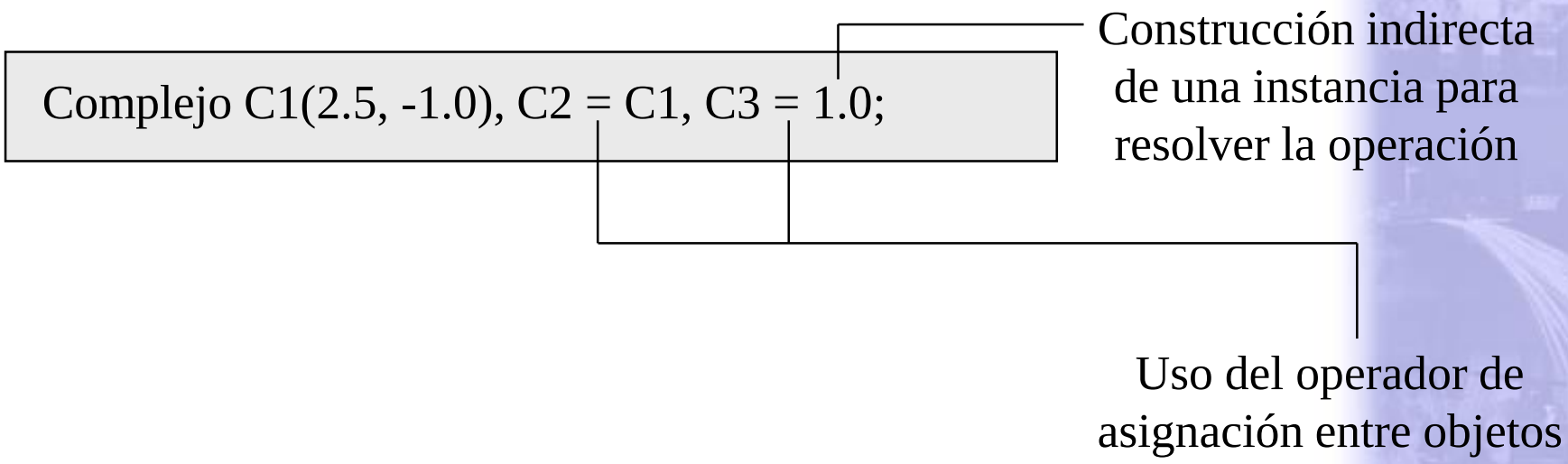
Constructor y destructor definidos fuera de la definición de la clase

Operador de Asignación o Copia

- Los lenguajes orientados a objetos manejan internamente el uso del operador natural de asignación cuando es aplicado sobre una instancia u objeto.
- El efecto es copiar el contenido de la memoria que corresponde a los atributos de la expresión que resulta del lado derecho de la asignación, en el espacio de memoria equivalente del objeto que recibe el valor en el lado izquierdo de la operación.
- Hay que tener cuidado con direcciones de memoria o uso de apuntadores como atributos (si aplica en el lenguaje).
- Los lenguajes que soportan la sobrecarga de operadores pueden dar su propia definición del operador de asignación y así evitar problemas como el caso anterior.
- En algunos lenguajes se realiza una llamada indirecta al constructor cuando el contexto es claro.

Operador de Asignación o Copia

✓ **C++**



Sobrecarga de Operadores

- Basado en el principio de polimorfismo.
- Permite escribir una implementación propia sobre el uso de un símbolo predeterminado, usado por el lenguaje para representar operaciones sobre elementos primitivos.
- Ejemplo: Para representar la suma de dos objetos C1 y C2 de tipo Complejo y guardar el resultado en el Complejo C3, es más elegante hacer

  C3 = C1 + C2

  que hacer

  C3 = C1.Suma(C2) ó
  C3 = Suma(C1, C2)

Sobrecarga de Operadores

✓ **C++**

- Se pueden sobrecargar los siguientes operadores:
  - Aritméticos: +, -, *, /, %, ++, --, +=, -=, *=, /=, %=.
  - Asignación: =.
  - Relacionales: >, <, >=, <=, ==, !=.
  - Manejo de bits: &, |, ^, >>, <<, ~, &=, |=, ^=, >>=, <<=.
  - Lógicos: &&, ||, !.
  - Conversión de tipos (casting): ( )
  - Direccionamiento de arreglos: [ ]
  - Manejo dinámico de memoria: new, delete.
- No es posible modificar la asociatividad del operador y tampoco la semántica de interpretación (por ejemplo, los operadores de relación deben retornar un valor lógico válido).

Sobrecarga de Operadores

✓ **C++**

```
class Complejo {
    private:
        double Real, Imag;

    public:
        …
        // Sobrecarga de un operador binario asociativo

        Complejo &operator +(Complejo &C) {
            return *new Complejo(Real + C.Real, Imag + C.Imag);
        }
        …
} C1, C2(5.0, -1.0), C3(2.5);
…
C1 = C2 + C3;          // Llamada del mensaje
```

Sobrecarga de Operadores

✓ **C++**

```
class Complejo {
    private:
        double Real, Imag;

    public:
        …
        operator double( ) {      // Sobrecarga de un operador
            return Real;          // unario de conversión de tipo
        }
        …
} C(5.0, -1.0);
…
double d = (double)C;        // Llamada del mensaje
```

Referencia al objeto actual

- Variable predefinida que permite al programador hacer referencia al objeto actualmente instanciado. Para los tres lenguajes en estudio la variable se identifica como **this**.
- Es útil cuando en el cuerpo de un método se hace referencia a un elemento de la clase con más de una instancia, incluyendo la actual y, se quiere representar la diferencia.
- Para los lenguajes que permiten la sobrecarga de operadores, es útil para retornar la instancia actual cuando el operador es unario y asociativo.
- En C++, **this** es un apuntador por lo que se debe usar con el operador →.
- En Java, **this** se puede usar como una función dentro de un constructor para hacer una llamada de otro constructor de la misma clase.

Referencia al objeto actual

✓ C++

```
class Complejo {
        …
    public:
        …
        Complejo &operator ++(void) {          // Operador unario
            *this = *this + Complejo(1.0, 1.0);  // asociativo con
            return *this;                      // modificación del
        }                                      // estado del objeto
        …                                      // que hace la llamada
} C(5.0, -1.0);
…
(++C)++;      // Ejecución del operador dos veces, la primera vez
              // como prefijo y la segunda como postfijo
```

Herencia

✓ **C++**

Constructor clase Base

Interfaz de la herencia

(¿Cómo serán vistos en clase Derivada los elementos públicos de la clase Base?)

Constructor clase Derivada

Llamada al constructor de la clase Base

```
class Carro {
    private:
        int Mod, Cap;
    public:
        Carro(int M = 0, int C = 0) { … }
    …
};

class Carga : public Carro {
    private:
        int CapCarga;
    public:
        Carga(int M = 0, int C = 0, int P = 0) :
            Carro(M, C) { … }
    …
};
```

Operador de relación entre clase Base y clase Derivada

Herencia

✓ C++

```
class A {
    private: int A1;
    protected: int A2;
    public: int A3;
};

class B : private A {
    private: int B1;
    protected: int B2;
    public: int B3;
};
```

	Dentro de B	En Deriva-das de B	Fuera de B
A1	✗	✗	✗
A2	✓	✗	✗
A3	✓	✗	✗
B1	✓	✗	✗
B2	✓	✓	✗
B3	✓	✓	✓

Herencia

✓ **C++**

```
class A {
    private: int A1;
    protected: int A2;
    public: int A3;
};

class C : protected A {
    private: int C1;
    protected: int C2;
    public: int C3;
};
```

	Dentro de C	En Deriva-das de C	Fuera de C
A1	✗	✗	✗
A2	✓	✗	✗
A3	✓	✓	✗
C1	✓	✗	✗
C2	✓	✓	✗
C3	✓	✓	✓

Herencia

✓ C++

```
class A {
    private: int A1;
    protected: int A2;
    public: int A3;
};

class D : public A {
    private: int D1;
    protected: int D2;
    public: int D3;
};
```

	Dentro de D	En Deriva-das de D	Fuera de D
A1	✗	✗	✗
A2	✓	✗	✗
A3	✓	✓	✓
D1	✓	✗	✗
D2	✓	✓	✗
D3	✓	✓	✓

Herencia

✓ **C++**

```
class B1 {
    private: int IB1;
    public:
        B1(int I = 0) { IB1 = I; }
    …
};
```

```
class B2 {
    private: int IB2;
    public:
        B2(int I = 0) { IB2 = I; }
    …
};
```

Herencia múltiple

```
class D : public B1, public B2 {
    public:
        D(int I1 = 0, int I2 = 0) : B1(I1), B2(I2) { };
    …
};
```

Constructores de clases Base

Herencia

✓ **C++**

```
class B1 {
    public: int B;
    …
};
```

```
class B2 {
    public: int B;
    …
};
```

```
class D : public B1, public B2 {
    public: int B;
    …
} UnD;
…
UnD.B = 0;          // Acceso al elemento B de D
UnD.B1::B = 0;      // Acceso al elemento B de B1
UnD.B2::B = 0;      // Acceso al elemento B de B2
```

Operador que indica búsqueda sobre un contexto superior (en la jerarquía) al actual

Herencia

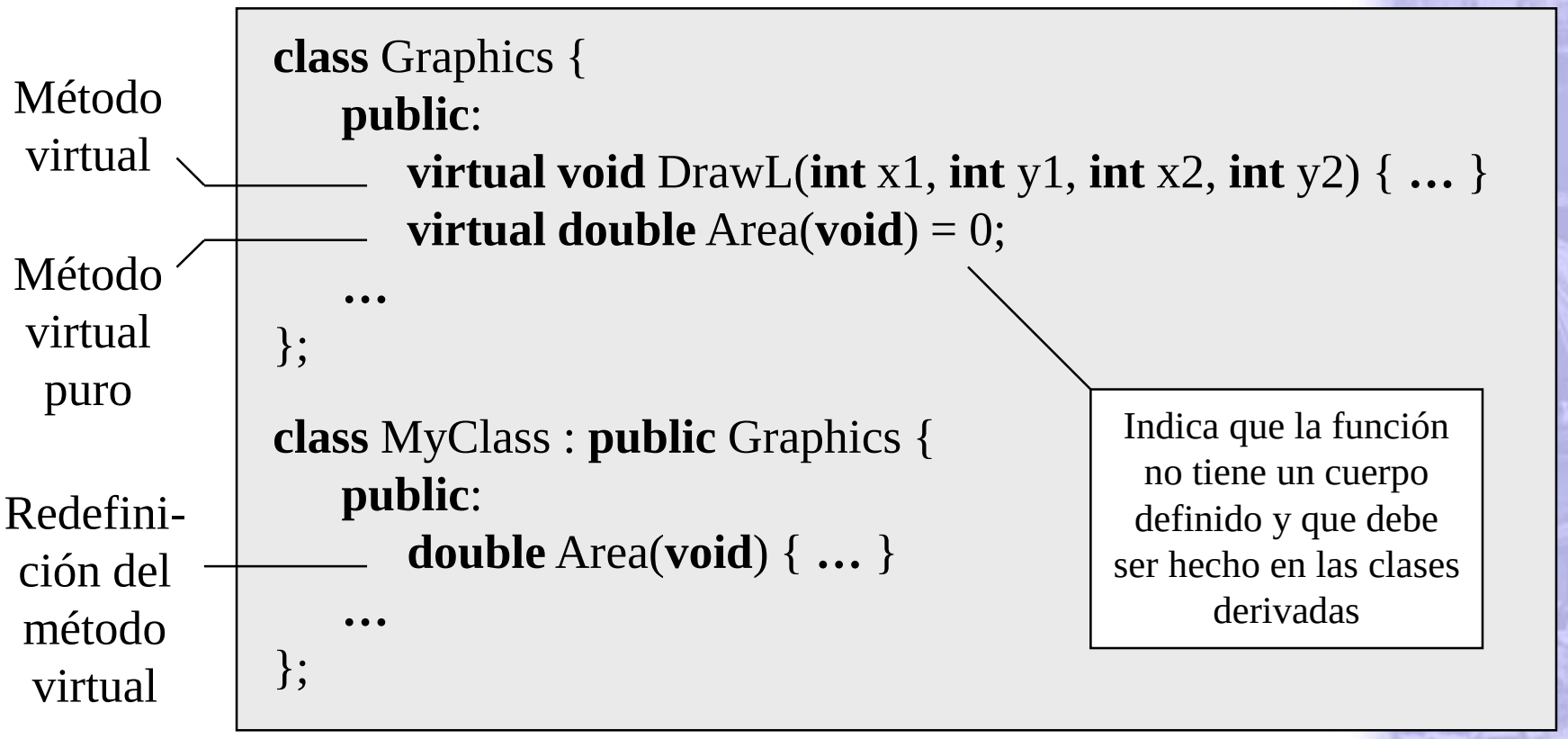
✓ **C++**

```
class Cb { public: int I; };
class Cd1 : public Cb { };
class Cd2 : public Cb { };
class Chm : public Cd1, public Cd2 { };
…
Chm hm; hm.Cd1::I = 1; hm.Cd2::I = 2;    // Cada I es independiente
cout << hm.Cd1::I << '\n' << hm.Cd2::I;   // La salida es:  1  2
```

```
class Cb { public: int I; };
class Cd1 : public virtual Cb { };
class Cd2 : public virtual Cb { };
class Chm : public Cd1, public Cd2 { };
…
Chm hm; hm.Cd1::I = 1; hm.Cd2::I = 2;    // I es compartido (el mismo)
cout << hm.Cd1::I << '\n' << hm.Cd2::I;   // La salida es:  2  2
```

Clases abstractas y métodos virtuales

✓**C++**



NOTA: Una clase que tenga al menos un método virtual puro es una clase abstracta (no es posible crear instancias con ella).