# Seminario de Lenguajes de Programación: Herencia

La **herencia** es un mecanismo fundamental en la **Programación Orientada a Objetos (POO)** que permite crear nuevas clases (**clases hijas**) a partir de clases existentes (**clases padre**), reutilizando sus atributos y métodos.

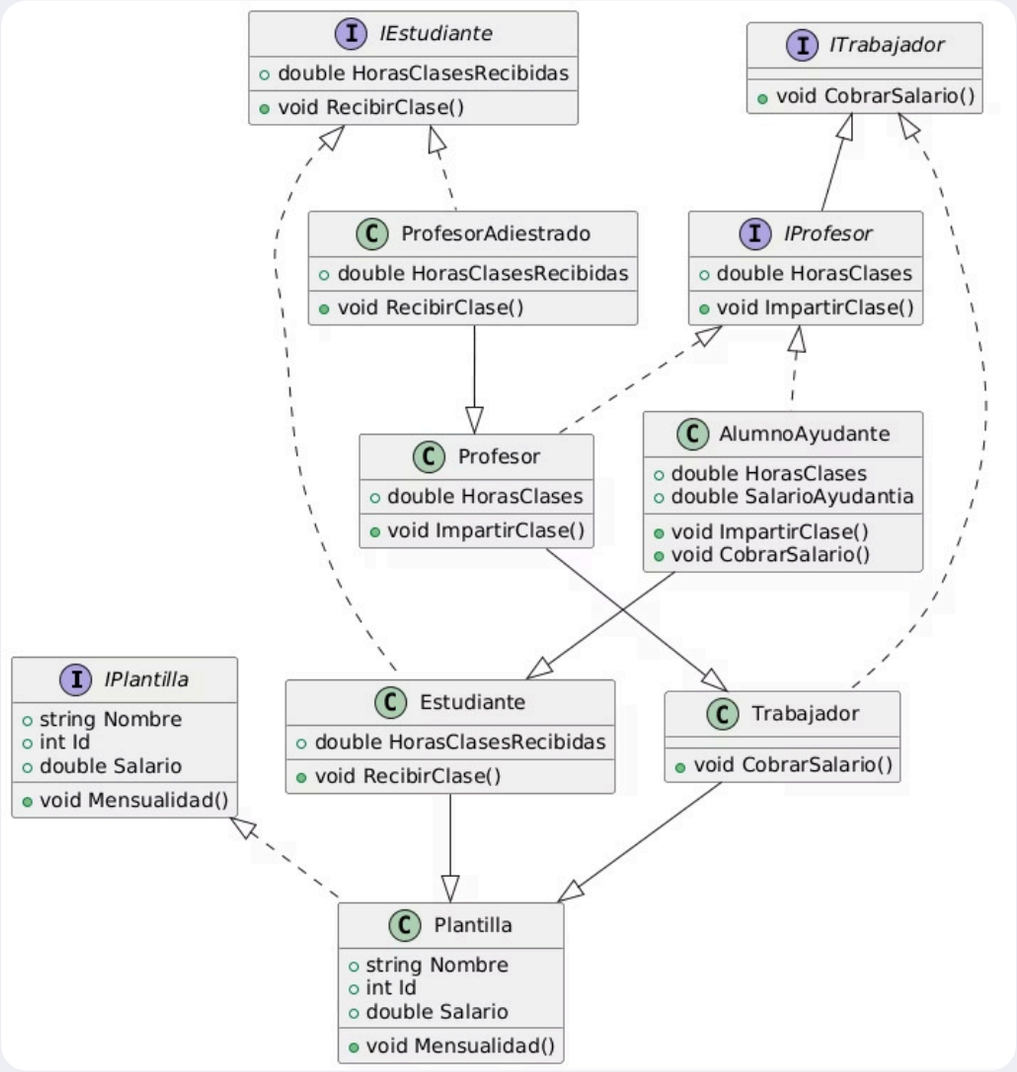
## Propósito de la Herencia

1. **Evitar duplicación de código**: Reutilizar funcionalidades ya definidas.
2. **Organización jerárquica**: Estructurar clases de forma lógica.
3. **Extender/modificar comportamientos**: La clase hija puede añadir nuevos métodos o modificar los heredados.

# Problema: Modelado de Roles Universitarios

En la Universidad, una persona (que se identifica por su Nombre) es plantilla si recibe algún tipo de pago y puede representar diferentes roles:

- Estudiante (Acción: `RecibirClase()` y `CobrarSalario()`)
- Profesor (Acción: `ImpartirClase()`)
- Alumno Ayudante (Estudiante que no es profesor pero actúa como tal en un momento dado, es decir, puede realizar `ImpartirClase()`, además puede cobrar un salario como trabajador separado del salario como estudiante)
- Trabajador (no todo trabajador es profesor, pero sí todos los profesores son trabajadores. Acción: `CobrarSalario()`)
- Profesor Adiestrado (Profesor que aún recibe cursos de adiestramiento, Acción: `RecibirClase()`)



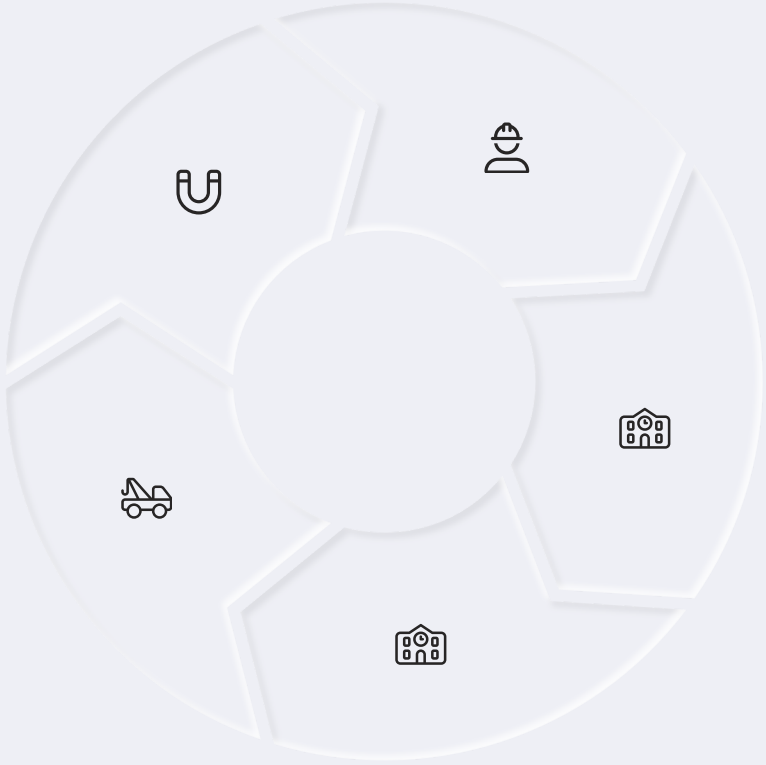
# Diagrama de Clases

### Plantilla (Abstracta)

Clase base con propiedades comunes: Nombre y Apellidos

### AlumnoAyudante

Hereda de Estudiante e implementa ITrabajador



### Trabajador

Hereda de Plantilla e implementa ITrabajador

### Profesor

Especialización de Trabajador con atributos de enseñanza

### Estudiante

Hereda de Plantilla con atributos académicos

# Contenidos a abordar

**Diseño de Clases**

**Herencia Múltiple**

**Visibilidad (modificadores de acceso)**

**Redefinición de miembros (sobrescritura)**

**Representación de objetos en Memoria**

**Ocultamiento de miembros**

**Polimorfismo**

**Casteo (conversión de tipos)**

# Herencia Múltiple C++

En C++, la sintaxis `class Derivada : public Base1, private Base2 { ... };` especifica la herencia y el nivel de acceso. El orden de construcción se rige por el orden de declaración de bases, y el destructor invoca el orden inverso. Para resolver el diamante—cuando dos rutas de herencia incluyen la misma base—se emplea la herencia virtual (`virtual Base`). Esto garantiza que sólo exista una instancia de la base común, evitando datos duplicados y llamadas múltiples al constructor

---

▼ Codigo C++

```
class A { /* ... */ };
class B : virtual public A { /* ... */ };
class C : virtual public A { /* ... */ };
class D : public B, public C { /* Única A compartida */ };
```

Sin `virtual`, D contendría dos subobjetos A y la llamada a A::método() sería ambigua.

# Ambigüedad en C++

En **C++** una clase puede heredar de más de una clase base separadas por comas, por ejemplo `class Hijo: public Base1, public Base2`. Esto permite combinar funcionalidades, pero puede generar ambigüedades: el famoso *problema del diamante* ocurre si `Base1` y `Base2` heredan de un mismo ancestro, dando ambigüedad en los datos comunes.

---

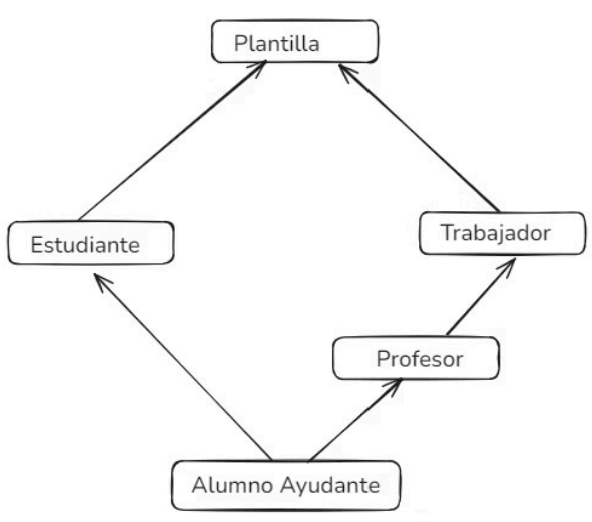
▼ Codigo C++

```cpp
// Clase base problemática (genera ambigüedad)
class Profesor : public Trabajador {
public:
    void CobrarSalario() const override {
        cout << nombre << " cobra salario de profesor: " << salario << endl;
    }
};

// Implementación AMBIGUA de AlumnoAyudante
class AlumnoAyudante : public Estudiante, public Profesor {  // ¡Herencia múltiple concreta!
    double salarioAyudantia;
public:
    AlumnoAyudante(string n, int i, double s1, double s2, double hr)
        : Estudiante(n, i, s1, hr), Profesor(n, i, s2, 0), salarioAyudantia(s2) {}

    // ¿Cuál CobrarSalario() se usa? ¿El de Estudiante o el de Profesor?
};
```

**Ambigüedad en** `CobrarSalario()`

El compilador no sabe si usar `Estudiante::CobrarSalario()` o `Profesor::CobrarSalario()`

▼ Problema del Diamante




C++ por defecto sigue cada ruta de herencia por separado, por lo que un objeto "`Alumno Ayudante`" realmente contendría dos objetos "`Plantilla`" separados, y el uso de los miembros de "`Plantilla`" debe ser adecuadamente definido. Si la herencia de "`Plantilla`" a "`Estudiante`" y la herencia de "`Plantilla`" a "`Trabajador`" están marcadas como "virtuales", C++ se preocupa por crear sólo un objeto "`Plantilla`", y el uso de sus miembros funciona correctamente. Si herencia virtual y herencia no virtual son mezcladas, hay un solo "`Plantilla`" virtual y un solo "`Plantilla`" no virtual para cada ruta de herencia no virtual a "`Plantilla`".

En nuestro ejemplo si quisiéramos usar el miembro X de la clase "`Plantilla`" que deriva en "`Estudiante`" usaríamos `Alumno Ayudante::Estudiante.`**X** o `Alumno Ayudante::Estudiante::Plantilla.`**X** por mas precisión.

# Solución de Ambigüedad en C++

---

▼ Herencia virtual

> Para evitar la duplicación de `Plantilla`, usamos herencia virtual en `Estudiante` y `Trabajador`:
>
> ```
> class Estudiante : virtual public Plantilla { /* ... */ };
> class Trabajador : virtual public Plantilla { /* ... */ };
> ```
>
> **Resultado**:
>
> - Ahora `AlumnoAyudante` contiene **solo una instancia** de `Plantilla`.
> - No hay ambigüedad al acceder a `nombre` o `CobrarSalario()`:

▼ Codigo C++

```
// Implementación CORRECTA (sin ambigüedad)
class AlumnoAyudante : public Estudiante, public IProfesor {
    double salarioAyudantia;
public:
    AlumnoAyudante(string n, int i, double s1, double s2, double hr)
        : Estudiante(n, i, s1, hr), salarioAyudantia(s2) {}

    // Implementación única de CobrarSalario()
    void CobrarSalario() const override {
        cout << nombre << " cobra salario de ayudantía: " << salarioAyudantia << endl;
    }
};
```

1. `IProfesor` **es una interfaz pura** (sin implementación de `CobrarSalario()`).
2. `Estudiante` **no hereda de** `Trabajador`, rompiendo el diamante.
3. `AlumnoAyudante` implementa `CobrarSalario()` explícitamente.

# Herencia Múltiple Python

En **Python**, la herencia múltiple también se permite usando paréntesis: `class A(B, C):`. Una clase hereda atributos y métodos de todos los padres. Python resuelve conflictos de nombre mediante el *MRO* (Method Resolution Order): el orden izquierdo-derecho en la declaración determina cuál método se llama primero.

---

▼ Codigo Python

```python
class A: def f(self): print("A")
class B(A): def f(self): print("B")
class C(A): def f(self): print("C")
class D(B, C): pass
print(D.mro())  # [D, B, C, A, object]
```

# Ambigüedad en Python

La herencia múltiple en **Python** puede causar **ambigüedad** principalmente en estos casos:

1. **Cuando dos o más clases base tienen métodos o atributos con el mismo nombre**, y una clase derivada hereda de ambas. Cuando no queda claro a cuál método/atributo llamar si no se maneja correctamente el orden de resolución.
2. **Cuando las clases bases derivan de una clase común (problema del diamante)**. Si no se maneja bien el `super()`, un método de la clase base común podría ser llamado múltiples veces o no llamado en absoluto.

---

▼ Codigo Python

```python
class A:
    def saludo(self):
        print('Hola desde A')

class B:
    def saludo(self):
        print('Hola desde B')

class C(A, B):
    pass

c = C()
c.saludo()  # ¿Cuál saludo debería llamar?
```

**Python usa el MRO (Method Resolution Order)** para decidir: va de izquierda a derecha según la declaración (`A` antes que `B`), entonces `c.saludo()` imprimirá **'Hola desde A'**.

```python
class A:
    def metodo(self):
        print('A')

class B(A):
    def metodo(self):
        print('B')
        super().metodo()

class C(A):
    def metodo(self):
        print('C')
        super().metodo()

class D(B, C):
    def metodo(self):
        print('D')
        super().metodo()

d = D()
d.metodo()
```

Aquí Python gestiona bien el problema del diamante gracias al **MRO**. Cada clase llama `super().metodo()`, pero siguiendo un orden que evita repetir llamadas a `A.metodo()`.

# Herencia Múltiple C#

En **C#**, directamente no se permite la herencia múltiple de clases, pero se puede lograr cierta funcionalidad similar utilizando interfaces y composición.

---

▼ Codigo C#

```
1 reference
public class AlumnoAyudante : Estudiante, IProfesor
{
    2 references
    public double HorasClases { get; set; }
    2 references
    private double SalarioAyudantia {get;set;}
    0 references | Tabnine | Edit | Test | Explain | Document
    public AlumnoAyudante(string nombre, int id, double salarioEstudiante, double salarioAyudantia,
                          double horasClasesRecibidas, double horasClases)
                          : base(nombre, id, salarioEstudiante, horasClasesRecibidas)
    {
        HorasClases = horasClases;
        SalarioAyudantia = salarioAyudantia;
    }

    1 reference | Tabnine | Edit | Test | Explain | Document
    public void ImpartirClase() => Console.WriteLine($"{Nombre} imparte clases");
    1 reference | Tabnine | Edit | Test | Explain | Document
    public void CobrarSalario() => Console.WriteLine($"{Nombre} recibe un salario de ayudantia de {SalarioAyudantia}");

}
```

# Solución de Ambigüedad en C#

---

▼ Ambigüedad

Si modificamos `Estudiante` para que **también implemente** `ITrabajador`, surgirá una ambigüedad:

```
public class Estudiante : Plantilla, IEstudiante, ITrabajador // ¡Ahora implementa ITrabajador!
{
    // ...
    public void CobrarSalario() => Console.WriteLine($"{Nombre} cobra beca estudiantil");
}
```

- `AlumnoAyudante` hereda:
  - `CobrarSalario()` de `Estudiante` (vía `ITrabajador`).
  - `CobrarSalario()` de `IProfesor` (vía `ITrabajador`).

▼ Solución

1. Indica cuál versión de `CobrarSalario()` corresponde a cada interfaz:

```
public class AlumnoAyudante : Estudiante, IProfesor
{
    // ...

    // Implementación explícita para IProfesor/ITrabajador
    void ITrabajador.CobrarSalario() => Console.WriteLine($"{Nombre} cobra ayudantía: {SalarioAy
udantia}");

    // Implementación de Estudiante (se usa por defecto)
    public override void CobrarSalario() => Console.WriteLine($"{Nombre} cobra beca estudianti
l");
}
```

```
var ayudante = new AlumnoAyudante(...);

// Llama a la implementación de Estudiante
ayudante.CobrarSalario();

// Llama a la implementación de ITrabajador
((ITrabajador)ayudante).CobrarSalario();
```

2. Si la herencia múltiple complica el diseño, usa **composición**:

```
public class AlumnoAyudante : Estudiante
{
    private IProfesor _rolProfesor;

    public AlumnoAyudante(..., IProfesor rolProfesor) : base(...)
    {
        _rolProfesor = rolProfesor;
    }

    public void ImpartirClase() => _rolProfesor.ImpartirClase();
    public void CobrarSalarioProfesor() => ((ITrabajador)_rolProfesor).CobrarSalario();
}
```

3. La idea es que la clase base resuelva la ambigüedad al proporcionar una implementación predeterminada para el miembro en cuestión, y luego las clases derivadas pueden heredar esa implementación o sobrescribirla según sea necesario.

# En Resumen la resolución de Ambigüedad por Lenguaje

### C++

- Especificar la clase base deseada usando el operador de resolución de alcance
- Usar herencia virtual para compartir una única instancia de la clase base
- Requiere intervención explícita del programador

### Python

- Emplea C3-linearización (MRO) para determinar el orden de búsqueda
- Resolución automática siguiendo un orden predefinido
- Permite invocar explícitamente a la superclase deseada

### C#

- No permite heredar de múltiples clases
- Usa implementación explícita de interfaces para resolver conflictos
- Cada interfaz puede tener su propia implementación de un método

# Colisión de interfaces e implementación implícita/implícita (C#)

En **C#**, cuando una clase **implementa múltiples interfaces** que contienen **miembros (métodos, propiedades) con el mismo nombre**, **ocurre una colisión de interfaces**.Esto pasa porque **el compilador no sabe cuál versión** de ese miembro deberías usar cuando accedes al objeto.

```
interface I1 { void Pintar(); }
interface I2 { void Pintar(); }
class MiClase : I1, I2 {
    public void Pintar() {
        Console.WriteLine("Pintar en MiClase");
    }
}
```

```
class MiClase : I1, I2 {
    void I1.Pintar() { Console.WriteLine("I1.Pintar"); }
    void I2.Pintar() { Console.WriteLine("I2.Pintar"); }
}
MiClase obj = new MiClase();
((I1)obj).Pintar(); // "I1.Pintar"
((I2)obj).Pintar(); // "I2.Pintar"
```

# Visibilidad (modificadores de acceso) C++

Los modificadores de acceso controlan qué partes del código pueden ver miembros de clase

## 1. Visibilidad en Clases Individuales

| Modificador | Acceso en la propia clase | Acceso en clases derivadas | Acceso externo (fuera de la clase) |
|---|---|---|---|
| `public` | ✅ Sí | ✅ Sí | ✅ Sí |
| `protected` | ✅ Sí | ✅ Sí | ❌ No |
| `private` | ✅ Sí | ❌ No | ❌ No |

Existen `public`, `protected` y `private`.

## 2. Herencia y Cambio de Visibilidad

| Tipo de Herencia | Miembro `public` en Base | Miembro `protected` en Base | Miembro `private` en Base |
|---|---|---|---|
| `public` | `public` en Derivada | `protected` en Derivada | ❌ No accesible |
| `protected` | `protected` en Derivada | `protected` en Derivada | ❌ No accesible |
| `private` | `private` en Derivada | `private` en Derivada | ❌ No accesible |

Además, en herencia se puede especificar `public`, `protected` o `private` para la clase base, cambiando la visibilidad heredada.

# Visibilidad (modificadores de acceso) C#

En C#, los **modificadores de acceso** definen la visibilidad de clases, métodos, propiedades y otros miembros. A diferencia de C++, C# incluye modificadores adicionales para controlar el acceso en contextos más específicos, como ensamblados (proyectos) o jerarquías de herencia.

| Modificador | Descripción | Acceso en la misma clase | Acceso en clases derivadas | Acceso en el mismo ensamblado | Acceso en otros ensamblados |
|---|---|---|---|---|---|
| `public` | Acceso sin restricciones. | ✅ | ✅ | ✅ | ✅ |
| `private` | Acceso **solo dentro de la clase contenedora**. | ✅ | ❌ | ❌ | ❌ |
| `protected` | Acceso dentro de la clase y sus **subclases** (incluso en otros ensamblados). | ✅ | ✅ | ❌ | ✅ (solo subclases) |
| `internal` | Acceso **dentro del mismo ensamblado** (proyecto). | ✅ | ✅ (si están en el ensamblado) | ✅ | ❌ |

| Modificador | Descripción | Acceso en la misma clase | Acceso en clases derivadas | Acceso en el mismo ensamblado | Acceso en otros ensamblados |
|---|---|---|---|---|---|
| `protected internal` | Acceso **dentro del mismo ensamblado** o desde **subclases en cualquier ensamblado** (combinación de `protected` e `internal`). | ✅ | ✅ | ✅ | ✅ (solo subclases) |
| `private protected` | Acceso **dentro de la misma clase** o **subclases en el mismo ensamblado** (C# 7.2+). | ✅ | ✅ (solo mismo ensamblado) | ✅ | ❌ |

public, protected , private, internal, protected internal, private protected

## 2. Herencia y Visibilidad

En C#, la herencia **no modifica los niveles de acceso** de los miembros de la clase base, pero sí determina si son accesibles desde la subclase:

| Modificador en Clase Base | Acceso en Clase Derivada |
|---|---|
| `public` | ✅ Accesible |
| `protected` | ✅ Accesible |
| `internal` | ✅ Accesible (solo si la subclase está en el mismo ensamblado) |
| `protected internal` | ✅ Accesible (en el mismo ensamblado o desde subclases en otros ensamblados) |
| `private` | ❌ No accesible |
| `private protected` | ❌ No accesible (a menos que la subclase esté en el mismo ensamblado) |

En C#, la herencia **no modifica los niveles de acceso** de los miembros de la clase base, pero sí determina si son accesibles desde la subclase

# Visibilidad (modificadores de acceso) Python

En Python, los modificuadores de acceso (**public**, **protected**, **private**) no se implementan de forma explícita como en C++ o C#, sino que se manejan mediante **convenciones de nomenclatura** y un mecanismo llamado *name mangling*.

| Acceso | Sintaxis | Acceso en la misma clase | Acceso en subclases | Acceso desde código externo | Herencia |
|---|---|---|---|---|---|
| **Público** | `nombre` | ✅ | ✅ | ✅ | Accesible directamente. |
| **Protegido** | `_nombre` | ✅ | ✅ (convención) | ✅ (no recomendado) | Accesible en subclases, pero se espera respeto a la convención ( `_nombre` ). |
| **Privado** | `__nombre` | ✅ (con `_Clase__nombre` ) | ❌ (solo con *name mangling*) | ❌ (solo con `_Clase__nombre` ) | No se hereda directamente. Requiere *name mangling* para acceder. |

# Redefinición de Miembros en C++, C# y Python

## C++

Se usan funciones miembro **virtual** para habilitar la redefinición.

- Clase base declara función virtual.
- Clase derivada redefine con la misma firma y la palabra **override**.

## C#

Métodos **virtual** y **override** controlan la redefinición.

- Clase base declara método virtual.
- Clase derivada usa override para redefinirlo.

## Python

Redefinición simplemente con redefinir método en la clase derivada.

- No se requieren palabras clave especiales.
- Mayor flexibilidad comparado con C++ y C#.

# Ocultamiento en C++, C# y Python

1

## Ocultamiento en C++

Control mediante **private**, **protected** y **public**. Miembros privados ocultan datos y métodos.

2

## Ocultamiento en C#

Se usan modificadores: **private**, **protected**, **public**, **internal** y **protected internal**.

3

## Ocultamiento en Python

Basado en convenciones con guion bajo (_) para indicar miembros privados, sin restricciones formales.

# Polimorfismo en C++, C# y Python

El polimorfismo es un concepto central en la programación orientada a objetos que permite que un mismo nombre de función pueda tener diferentes comportamientos en función del tipo de objeto que lo invoca.

## C++

- Se implementa mediante **funciones miembro virtuales** en clases base.
- Las clases derivadas **sobrescriben** esas funciones usando override.
- **Resolución en tiempo de ejecución** según el tipo real del objeto.

▼ Codigo

```cpp
class Base {
public:
    virtual void metodo() {
        cout << "Implementación en Base" << endl;
    }
};
class Derivada : public Base {
public:
    void metodo() override {
        cout << "Implementación en Derivada" << endl;
    }
};

Base* objeto = new Derivada();
objeto->metodo(); // Imprime: Implementación en Derivada
```

## C#

- Usa **métodos virtuales** (virtual) en la clase base.
- Se sobrescriben en las clases derivadas usando override.
- También resuelve el método correcto **en tiempo de ejecución**.

▼ Código

```cpp
class Base {
    public virtual void Metodo() {
        Console.WriteLine("Implementación en Base");
    }
}
class Derivada : Base {
    public override void Metodo() {
        Console.WriteLine("Implementación en Derivada");
    }
}

Base objeto = new Derivada();
objeto.Metodo(); // Imprime: Implementación en Derivada
```

## Python

- Se logra de forma **natural** gracias a la **naturaleza dinámica** del lenguaje.
- No se requiere virtual ni override.
- Utiliza el principio de **duck typing**: "si camina como un pato y suena como un pato, es un pato".

▼ Código

```cpp
class Base:
    def metodo(self):
        print("Implementación en Base")

class Derivada(Base):
    def metodo(self):
        print("Implementación en Derivada")

objeto = Derivada()
objeto.metodo() # Imprime: Implementación en Derivada
```

# Casteo en C++, C# y Python

El casteo, también conocido como conversión de tipos, es el proceso de cambiar un objeto de un tipo a otro tipo compatible. Cada lenguaje tiene su propio conjunto de reglas y mecanismos para realizar el casteo

| Característica | C++ | C# | PYTHON |
|---|---|---|---|
| **Tipado** | Estático | Estático | Dinámico |
| **Casteo Estático** | `static_cast<Tipo>(valor)` | `(Tipo)valor` | Conversión automática en algunos casos |
| **Casteo Dinámico** | `dynamic_cast<Tipo>(ptr)` | `as` y `is` | No aplica (resuelto dinámicamente) |
| **Casteo Seguro** | Verifica en tiempo de ejecución con `dynamic_cast` (puede devolver `nullptr`) | `as` devuelve `null` si falla | No nativo, se maneja con excepciones si es necesario |
| **Funciones de Conversión** | No tan comunes (`reinterpret_cast`, etc.) | Algunos métodos como `Convert.ToInt32()`, etc. | Funciones integradas (`int()`, `float()`, `str()`) |
| **Control de Errores** | Necesario manejar errores en `dynamic_cast` | Controlar `null` tras `as` y validar con `is` | Generalmente falla en tiempo de ejecución si conversión inválida |
| | | | |

# Representación de Objetos con Herencia Simple y Múltiple C#, C++, Python

## Objetos de una clase sin herencia

| Lenguaje | Representación en Memoria |
|---|---|
| C# | - Campos en heap- Puntero a la VMT para métodos virtuales |
| C++ | - Atributos en stack o heap- Puntero a vtable si hay métodos virtuales |
| Python | - Atributos en `__dict__` en heap- Puntero a la clase para acceder a métodos |

### C#

```
var plantilla = new Plantilla("Juan", 1, 1000);
```

### Representación en memoria:

- **Heap**:
  - Nombre: "Juan"
  - Id: 1
  - Salario: 1000
  - **Puntero a la VMT**: Este puntero permite acceder a los métodos virtuales de la clase, como Mensualidad().

### C++

```
Plantilla plantilla("Juan", 1, 1000);
```

### Representación en memoria:

- **Stack** o **Heap**:
  - nombre: "Juan"
  - id: 1
  - salario: 1000
  - **Puntero a la vtable**: Si la clase tiene métodos virtuales, este puntero permite localizar las implementaciones de los métodos.

### Python

```
plantilla = Plantilla("Juan", 1, 1000)
```

### Representación en memoria:

- **Heap**:
  - __dict__: {'_nombre': 'Juan', '_id': 1, '_salario': 1000}
  - **Puntero a la clase**: Este puntero permite acceder a los métodos definidos en la clase Plantilla.

# Representación de Objetos con Herencia Simple y Múltiple C#, C++, Python

## Objeto de una clase con herencia simple

| Lenguaje | Representación en Memoria |
|---|---|
| C# | - Campos de clase base y derivada en heap- Puntero a la VMT para métodos virtuales |
| C++ | - Atributos de clase base y derivada en stack o heap- Puntero a vtable si hay métodos virtuales |
| Python | - `__dict__` con atributos de clase base y derivada en heap- Puntero a la clase para métodos |

### C#

```
var trabajador = new Trabajador("Ana", 2, 1500);
```

### Representación en memoria:

- **Heap**:
  - Nombre: "Ana"
  - Id: 2
  - Salario: 1500
  - **Puntero a la VMT**: Este puntero permite acceder a los métodos virtuales de la clase base y derivada.

### C++

```
Trabajador trabajador("Ana", 2, 1500);
```

### Representación en memoria:

- **Stack** o **Heap**:
  - nombre: "Ana"
  - id: 2
  - salario: 1500
  - **Puntero a la vtable**: Si hay métodos virtuales, este puntero permite localizar las implementaciones.

### Python

```
trabajador = Trabajador("Ana", 2, 1500)
```

### Representación en memoria:

- **Heap**:
  - __dict__: {'_nombre': 'Ana', '_id': 2, '_salario': 1500}
  - **Puntero a la clase**: Este puntero permite acceder a los métodos definidos en la clase Trabajador.

# Representación de Objetos con Herencia Simple y Múltiple C#, C++, Python

## Objeto de una clase con herencia múltiple

| Lenguaje | Representación en Memoria |
| --- | --- |
| C# | - Campos de clase base en heap- Implementaciones de interfaces- Puntero a la VMT |
| C++ | - Atributos de todas las clases base en stack o heap- Múltiples punteros a vtables (uno por clase base virtual) |
| Python | - `__dict__` con atributos de todas las clases base- Puntero a la clase para acceder a métodos |

## C#

```
var alumnoAyudante = new AlumnoAyudante("Luis", 4, 800, 500, 5, 15);
```

### Representación en memoria:

- **Heap**:
  - Nombre: "Luis"
  - Id: 4
  - Salario: 800
  - HorasClasesRecibidas: 5
  - HorasClases: 15
  - SalarioAyudantia: 500
  - **Puntero a la VMT**: Este puntero permite acceder a los métodos virtuales de las interfaces implementadas.

## C++

```
AlumnoAyudante alumnoAyudante("Luis", 4, 800, 500, 5, 15);
```

### Representación en memoria:

- **Stack** o **Heap**:
  - nombre: "Luis"
  - id: 4
  - salario: 800
  - horasRecibidas: 5
  - horas: 15
  - salarioAyudantia: 500
  - **Punteros a las vtables**: Si hay métodos virtuales, se incluyen punteros para cada clase base.

## Python

```
alumno_ayudante = AlumnoAyudante("Luis", 4, 800, 500, 5, 15)
```

### epresentación en memoria:

- **Heap**:
  - __dict__: {'_nombre': 'Luis', '_id': 4, '_salario': 800, '_horas_clases_recibidas': 5, '_horas_clases': 15, '_salario_ayudantia': 500}
  - **Puntero a la clase**: Este puntero permite acceder a los métodos definidos en la clase AlumnoAyudante.

# Conclusiones

### Diseño Óptimo

Combinación de herencia simple e interfaces

### Solución a Limitaciones

Interfaces para simular roles múltiples

### Estructura Clara

Jerarquía bien definida sin ambigüedades

El modelado de roles universitarios en C# se logra combinando herencia simple de clases para jerarquías claras e interfaces para roles compuestos. Esta estrategia evita los problemas de ambigüedad que pueden surgir con la herencia múltiple en otros lenguajes, mientras mantiene un diseño flexible y extensible.

La representación en memoria sigue un patrón natural: un bloque contiguo con metadatos y campos ordenados según la jerarquía. Este enfoque proporciona una base sólida para sistemas orientados a objetos complejos como el modelado de una universidad.