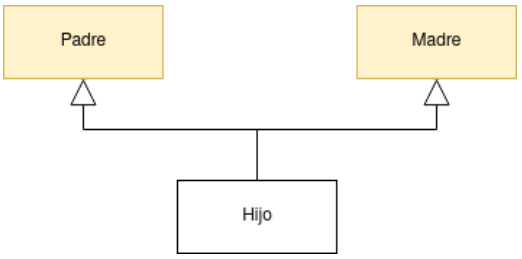
Herencia Múltiple

Néstor Eduardo Suat Rojas, Ing. Msc.
Docente Catedrático - Escuela de Ingeniería

INTRODUCCIÓN.

En la programación orientada a objetos **una clase puede heredar de más de un padre**.



Los constructores de las clases heredadas son llamados en el mismo orden en que son heredados.

```
class Dad{
public:
   Dad(){
          cout << "Constructor de la clase Padre" <<
endl;
   }
};
class Mom{
public:
   Mom(){
          cout << "Constructor de la clase Madre" <<
endl;
   }
};
class Son: public Dad, public Mom{
public:
   Son(){
       cout << "Constructor de la clase Hijo" << endl;
   }
};
int main() {
   Son hijo;
   return 0;
}
```

Output:

```
Constructor de la clase Padre
Constructor de la clase Madre
Constructor de la clase Hijo
```

PROBLEMA DE LA AMBIGÜEDAD

- Es muy posible que con la herencia múltiple ocurra la sobrecarga de funciones.
- Ocurre cuando dos clases padre tienen una misma función.
- Si el objeto de la clase hija intenta llamar a la función duplicada el compilador no sabe a qué función llamar.

```
class Dad{
public:
    void show(){
        cout << "Soy el padre" << endl;
    }
};

class Mom{
public:
    void show(){
        cout << "Soy la madre" << endl;
    }
};

class Son: public Dad, public Mom{
};

int main() {
    Son hijo;
    hijo.show(); // Error!
    return 0;
}
```

Solución

Este problema se puede solucionar utilizando la **función de resolución scope** para especificar qué función se debe clasificar.

```
int main() {
    Son hijo;
    hijo.Dad::show(); // Soy el padre
    hijo.Mom::show(); // Soy la madre
    return 0;
}
```

Ejemplo con atributo duplicado

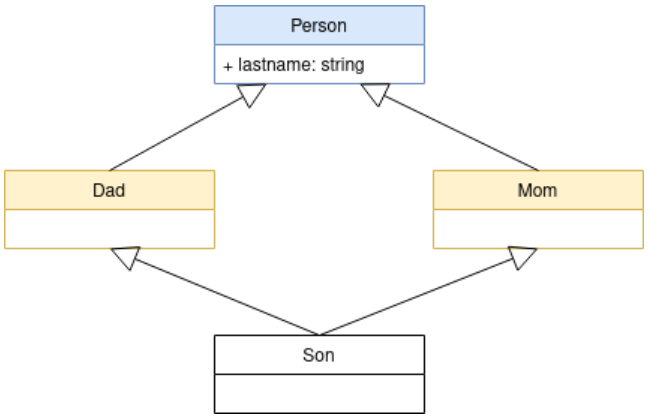
```
class Dad{
public:
    string lastname;
    void show(){
        cout << "Soy el padre" << endl;
    }
};

class Mom{
public:
    string lastname;
    void show(){
        cout << "Soy la madre" << endl;
    }
};

class Son: public Dad, public Mom{
};

int main() {
    Son hijo;
    hijo.Dad::lastname = "Suat";
    hijo.Mom::lastname = "Rojas";

    cout << hijo.Dad::lastname << " " << hijo.Mom::lastname;
    return 0;
}
```

Ejemplo con constructores

```
class Dad{
public:
    string lastname;

    Dad(string lastname){
        Dad::lastname = lastname;
    }

    void show(){
        cout << "Soy el padre" << endl;
    }
};

class Mom{
public:
    string lastname;

    Mom(string lastname){
        Mom::lastname = lastname;
    }

    void show(){
        cout << "Soy la madre" << endl;
    }
};

class Son: public Dad, public Mom{
public:
    Son(string lastname, string secondLastname): Dad(lastname), Mom(secondLastname){}
};

int main() {
    Son hijo("Suat", "Rojas");

    cout << hijo.Dad::lastname << " " << hijo.Mom::lastname;
    return 0;
}
```

El problema del diamante

Ocurre cuando dos clases padres de una clase tienen en común una clase base (abuelo).




```
class Person{
public:
    string lastName;
    Person(string lastName){
        cout << "Soy constructor(string) Persona" << endl;
    }
};
class Dad: public Person{
public:
    Dad(string lastName): Person(lastName){
        cout << "Soy constructor(string) Padre" << endl;
    }
};
class Mom: public Person{
public:
    Mom(string lastName): Person(lastName){
        cout << "Soy constructor(string) Madre" << endl;
    }
};
class Son: public Dad, public Mom{
public:
    Son(string name, string lastName, string SecondLastName): Dad(lastName),
Mom(SecondLastName){
        cout << "Soy constructor(string) Hijo" << endl;
    }
};
int main() {
    Son hijo("Néstor", "Suat", "Rojas");
    return 0;
}
```

```
Output:
Soy constructor(string) Persona
Soy constructor(string) Padre
Soy constructor(string) Persona
Soy constructor(string) Madre
Soy constructor(string) Hijo
```

Problema de ambigüedad

- En el programa anterior, el constructor de "`Person`" es llamado dos veces.
- El objeto `hijo` tiene dos copias de todos los miembros de "`Person`".
- La solución es utilizar la palabra `virtual`. En la herencia de `Dad` y `Mom` como clases base virtuales para evitar dos copias de `Person` en la clase `hijo`.

```
class Person{
public:
   string lastName;
   Person(){
        cout << "Constructor default Persona" << endl;
   }
   Person(string lastName){
        cout << "Soy constructor(string) Persona" << endl;
   }
};

class Dad: virtual public Person{
public:
   Dad(string lastName): Person(lastName){
        cout << "Soy constructor(string) Padre" << endl;
   }
};

class Mom: virtual public Person{
public:
   Mom(string lastName): Person(lastName){
        cout << "Soy constructor(string) Madre" << endl;
   }
};

class Son: public Dad, public Mom{
public:
   Son(string name, string lastName, string SecondLastName): Dad(lastName),
Mom(SecondLastName){
        cout << "Soy constructor(string) Hijo" << endl;
   }
};

int main() {
   Son hijo("Néstor", "Suat", "Rojas");
   return 0;
}
```

```
Output:
Constructor default Persona
Soy constructor(string) Padre
Soy constructor(string) Madre
Soy constructor(string) Hijo
```

Llamado por defecto constructor por default.

Cuando usamos la palabra clave "`virtual`", el constructor por default de la clase abuelo es llamado por defecto incluso si las clases padre llaman explícitamente al constructor parametrizado.

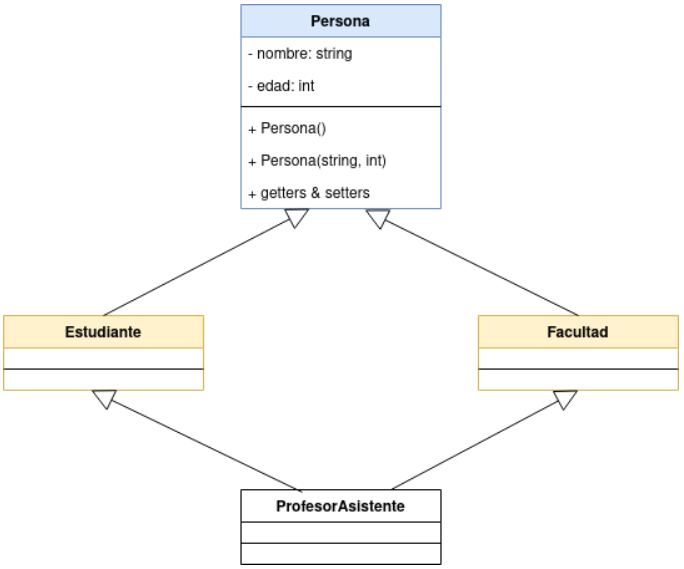
Solución: Llamar explícitamente al constructor del abuelo directamente.

```
class Person{
public:
   string lastName;
   Person(){
       cout << "Constructor default Persona" << endl;
   }
   Person(string lastName){
       cout << "Soy constructor(string) Persona" << endl;
   }
};

class Dad: virtual public Person{
public:
   Dad(string lastName): Person(lastName){
       cout << "Soy constructor(string) Padre" << endl;
   }
};

class Mom: virtual public Person{
public:
   Mom(string lastName): Person(lastName){
       cout << "Soy constructor(string) Madre" << endl;
   }
};

class Son: public Dad, public Mom{
public:
   Son(string name, string lastName, string SecondLastName): Dad(lastName),
Mom(SecondLastName), Person(lastName){
       cout << "Soy constructor(string) Hijo" << endl;
   }
};

int main() {
   Son hijo("Néstor", "Suat", "Rojas");
   return 0;
}
```

```
Output:
Soy constructor(string) Persona
Soy constructor(string) Padre
Soy constructor(string) Madre
Soy constructor(string) Hijo
```

- No está permitido llamar al constructor del abuelo directamente, tiene que ser a través de la clase padre. Sólo se permite cuando se utiliza la palabra clave "`virtual`".

Otro ejemplo de Herencia Múltiple



Problema constructor por default abuelo.

```
class Persona{
    string nombre;
    int edad;
public:
    Persona(){
        nombre = "";
        edad = 0;
    }
    Persona(string nombre, int edad){
        Persona::nombre = nombre;
        Persona::edad = edad;
    }
    string getNombre(){
        return nombre;
    }
    void setNombre(string nombre){
        Persona::nombre = nombre;
    }
    int getEdad(){
        return edad;
    }
    void setEdad(int edad){
        Persona::edad = edad;
    }
    friend ostream &operator<<(ostream &output, Persona p){
        output << "Nombre: " << p.getNombre() << endl;
        output << "Edad: " << p.getEdad() << endl;
    }
};
```

```
class Estudiante: virtual public Persona{
public:
   Estudiante(string nombre, int edad): Persona(nombre, edad){
   }
};

class Facultad: virtual public Persona{
public:
   Facultad(string nombre, int edad): Persona(nombre, edad){
   }
};

class ProfesorAsistente: public Estudiante, public Facultad{
public:
   ProfesorAsistente(string nombre, int edad): Estudiante(nombre, edad),
Facultad(nombre, edad){
   }

};

int main() {
   ProfesorAsistente pa1("Victor",23);
   cout << pa1;
   cout << pa1.getNombre();
   return 0;
}
```

```
Output:
Nombre:
Edad: 0
```

Solución: Llamar al constructor de la clase abuelo en la clase hija.

```
class Persona{
    string nombre;
    int edad;
public:
    Persona(){
        nombre = "";
        edad = 0;
    }
    Persona(string nombre, int edad){
        Persona::nombre = nombre;
        Persona::edad = edad;
    }
    string getNombre(){
        return nombre;
    }
    void setNombre(string nombre){
        Persona::nombre = nombre;
    }
    int getEdad(){
        return edad;
    }
    void setEdad(int edad){
        Persona::edad = edad;
    }
    friend ostream &operator<<(ostream &output, Persona p){
        output << "Nombre: " << p.getNombre() << endl;
        output << "Edad: " << p.getEdad() << endl;
    }
};

class Estudiante: virtual public Persona{
public:
    Estudiante(string nombre, int edad): Persona(nombre, edad){
    }
};

class Facultad: virtual public Persona{
public:
    Facultad(string nombre, int edad): Persona(nombre, edad){
    }
};

class ProfesorAsistente: public Estudiante, public Facultad{
public:
    ProfesorAsistente(string nombre, int edad): Estudiante(nombre, edad),
Facultad(nombre, edad), Persona(nombre, edad){
    }

};

int main() {
    ProfesorAsistente pa1("Victor",23);
    cout << pa1;
    cout << pa1.getNombre();
    return 0;
}
```

Output:

```
Nombre: Victor
Edad: 23
Victor
```

FUENTES:

- *Multiple inheritance in C++*. GeeksforGeeks. (2018, September 6). Retrieved January 23, 2022, from https://www.geeksforgeeks.org/multiple-inheritance-in-c/
- *C++ multiple, multilevel and hierarchical inheritance*. Programiz. (n.d.). Retrieved January 23, 2022, from https://www.programiz.com/cpp-programming/multilevel-multiple-inheritance