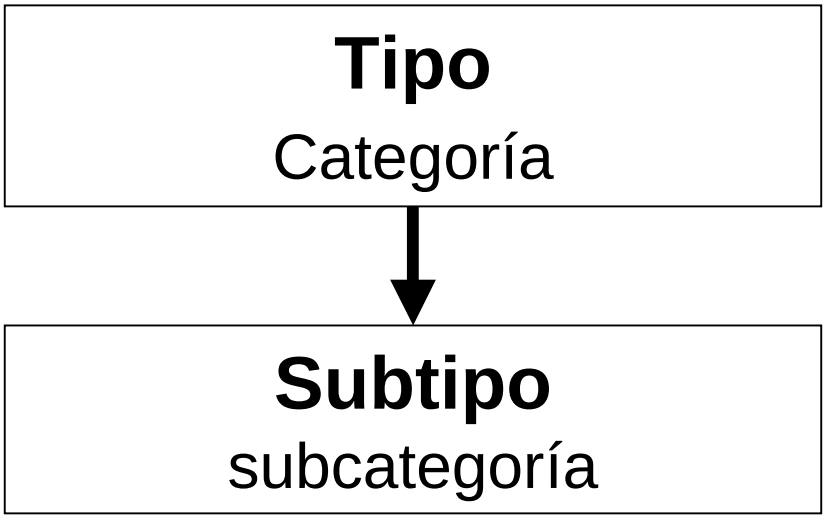
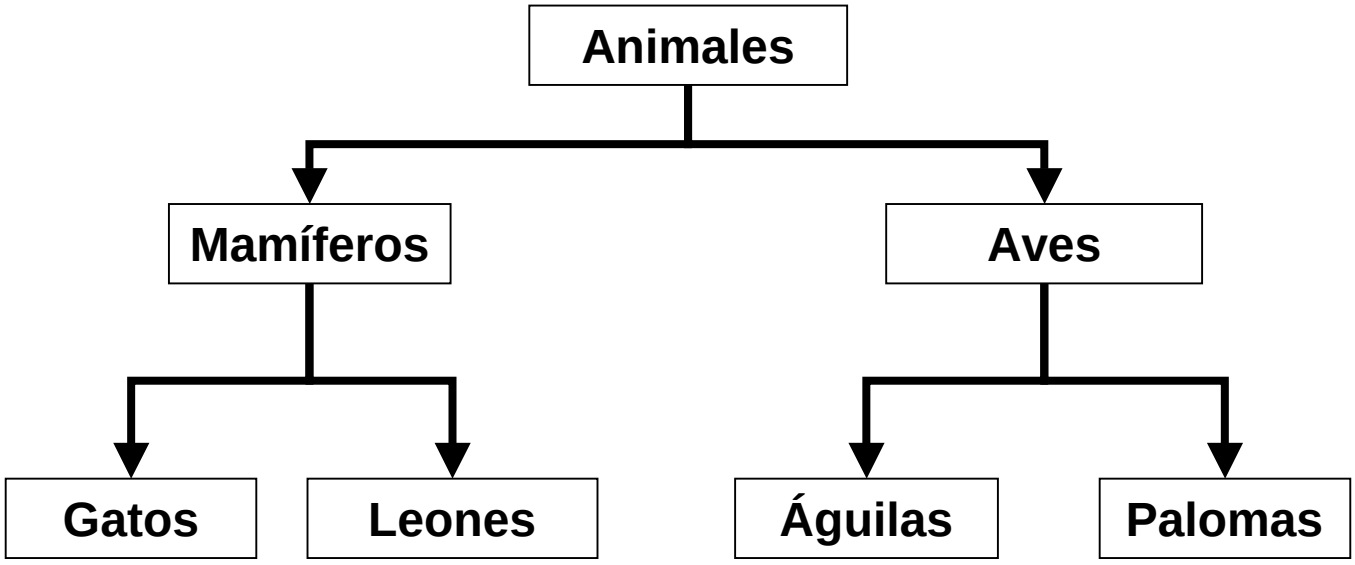
La herencia

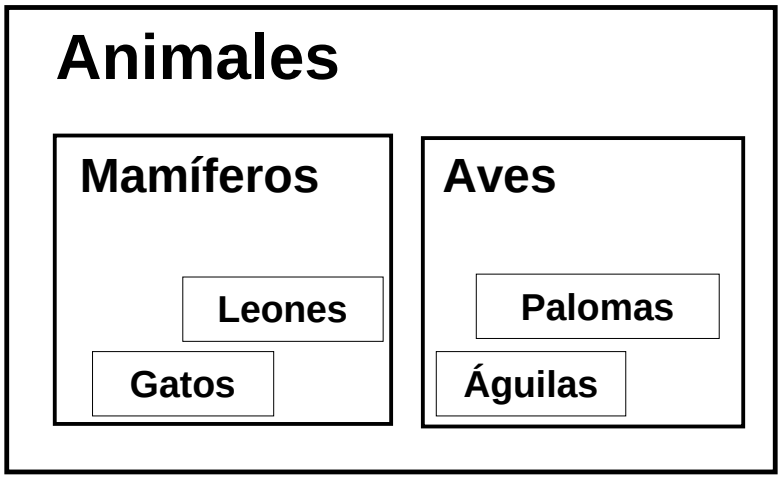
- Recurso muy importante de los lenguajes P.O.O.
- Definir una nueva clase:
  - como extensión de otra previamente definida.
  - sin modificar la ya existente.
- La nueva clase hereda de la clase anterior:
  - las variables.
  - las operaciones .
- Principal objetivo/ventaja:
  - Reutilización del código.
    - Ahorro de esfuerzo.
    - Mayor confianza en el código.



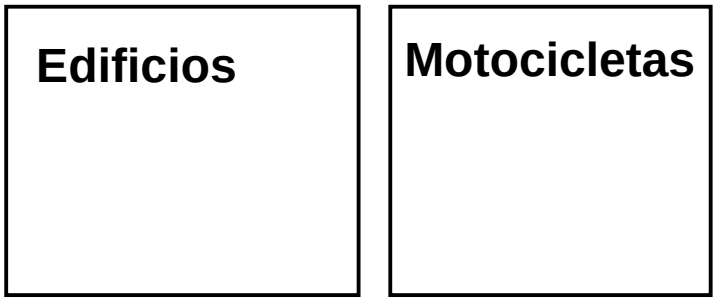


- **Organización jerárquica de categorías.**
- **Relación es-un.**
- **Relación supertipo-subtipo.**

- El conjunto de elementos que pertenecen a un tipo incluye a los elementos que pertenezcan a sus subtipos.



Conjuntos anidados de objetos. Relación entre tipos y subtipos.

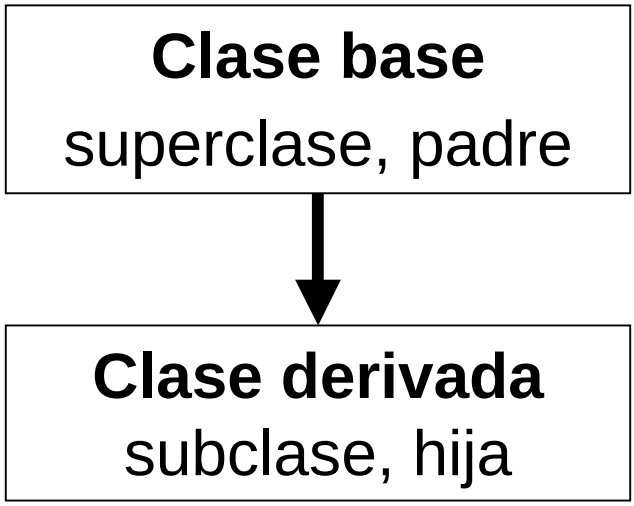


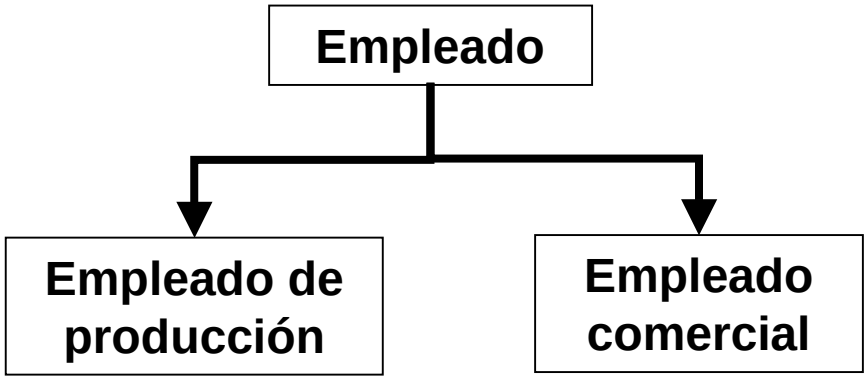
Conjuntos disjuntos. No hay relación de subtipado entre estos tipos.

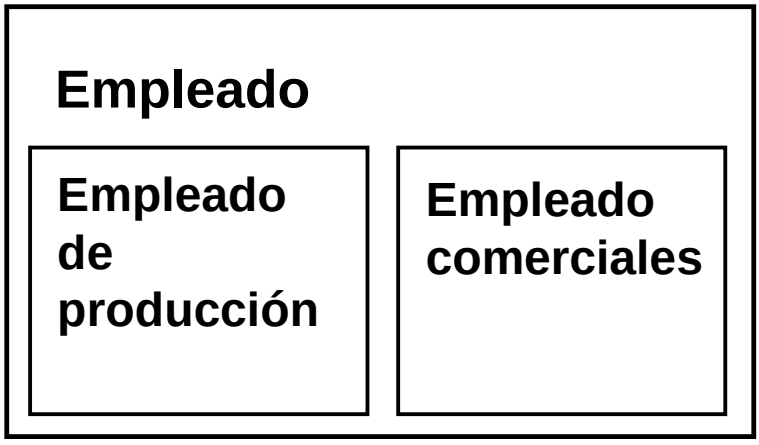
- Principio de subtipos:

  "*Un objeto de un subtipo puede aparecer en cualquier lugar donde se espera que aparezca un objeto del supertipo*."

  - Los animales son capaces de moverse por sí mismos.
  - Los mamíferos son capaces de moverse por sí mismos.
  - Las aves son capaces de moverse por sí mismas.
  - Los gatos son capaces de moverse por sí mismos.
- A la inversa no es cierto.
  - Los gatos maullan.
  - Los mamíferos maullan.
  - Los animales maullan.


Clase base
superclase, padre
Clase derivada
subclase, hija


Empleado
Empleado de producción
Empleado comercial


Empleado
Empleado de producción
Empleado comerciales

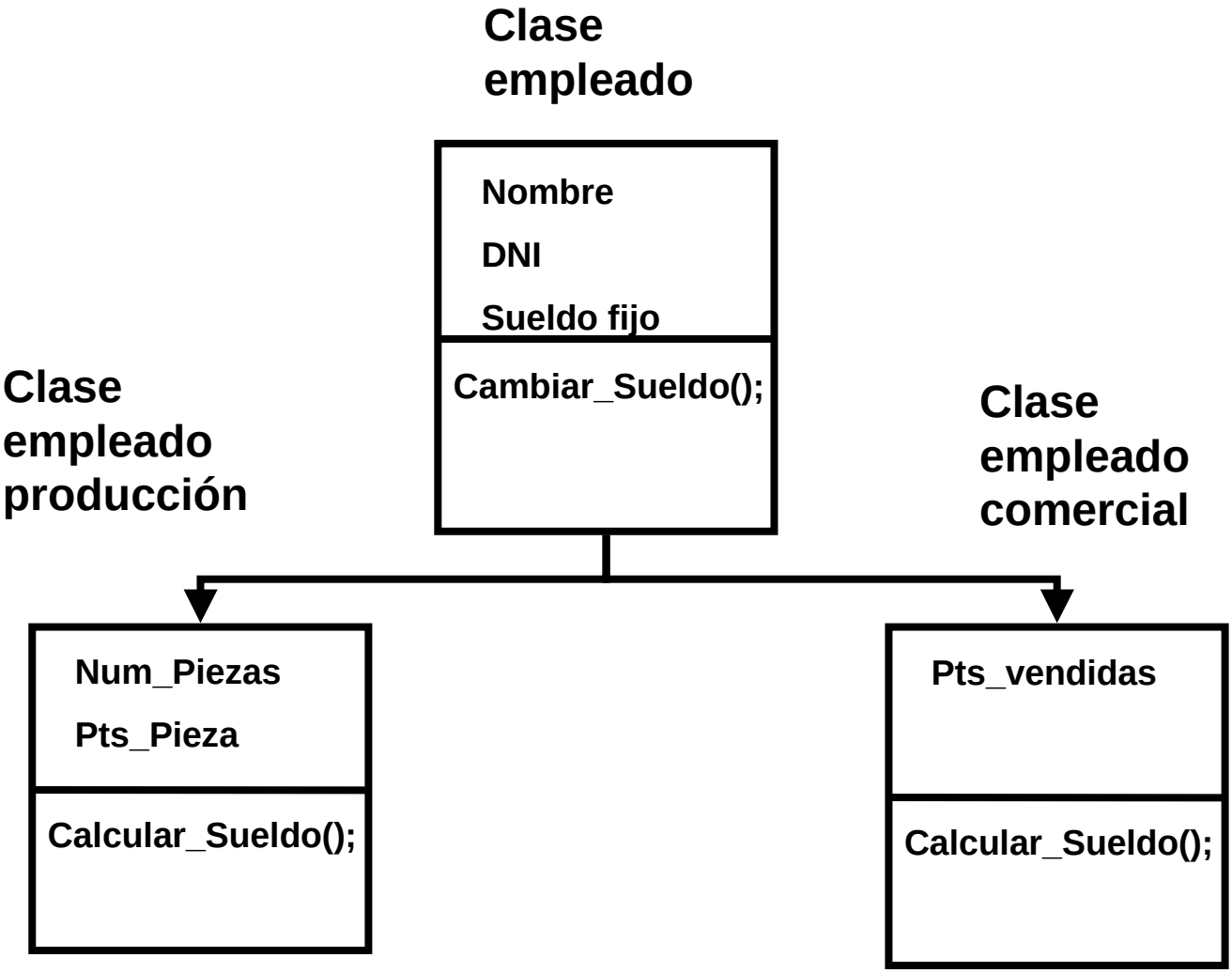
- El principio de los subtipos también se cumple en la P.O.O.
- Una función que recibe un objeto de la clase base.

```
void Despedir ( empleado);
```

- Puede recibir objetos de la clase base y también de sus derivadas.

```
empleado e;
empleado_producción ep;
empleado_comerciales ec;

Despedir (e);
Despedir (ep);
Despedir (ec);
```

Herencia de métodos y variables.



- Las clases derivadas reciben las variables y métodos de la clase base.

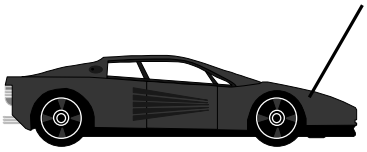
```
Empleado_comercial ec;
Empleado_produccion ep;

ec.Cambiar_Sueldo();
ep.Cambiar_Sueldo();
```

Composición:

- **Relación tener-un**

  Un coche tiene un tipo de motor

- **Composición significa contener un objeto.**



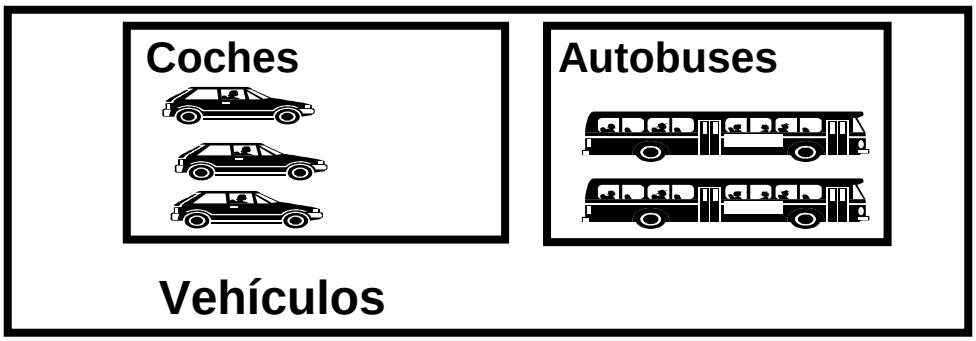

```
class coche
{ ...
private:
    Motor _motor;
};
```

Herencia:

- **Relación ser-un**

  Un coche es un vehículo

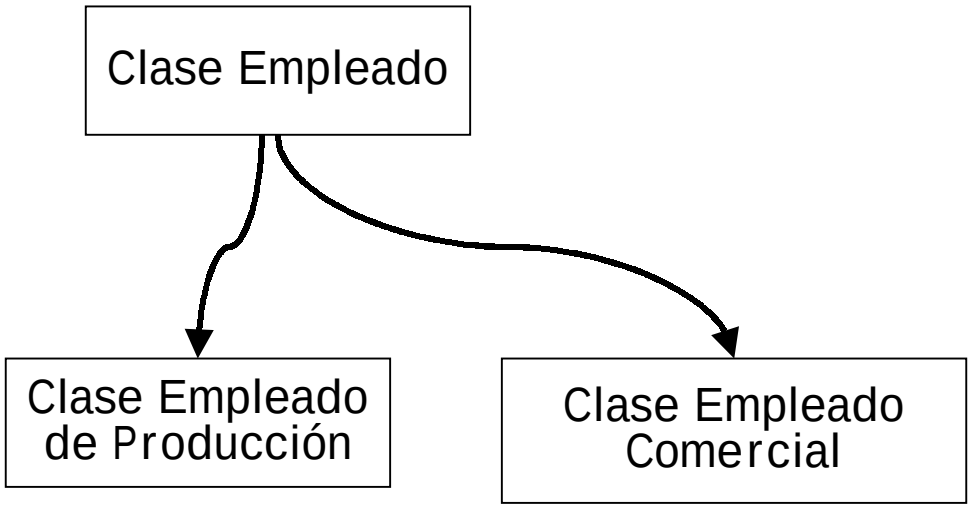
- **Herencia significa contener una clase.**




```
class coche: vehiculo
```

Ejemplos de composición y herencia

- Clase persona y clase empleado.
  - Herencia: un empleado **es** una persona.
- Clase persona y clase domicilio.
  - Composición: una persona **tiene** un domicilio.
- Clase lista y clase nodo de la lista:
  - Composición: una lista **tiene** un puntero de tipo nodo al nodo que está en cabeza de la lista (tener-un).
- Clase empresa, clase empleado y clase jefe de grupo de empleados.
  - Herencia entre empleado y jefe: Un jefe **es** un empleado.
  - Composición entre empresa y empleado o jefe.
    - Una empresa puede **tener** una lista de empleados y otra de jefes.
    - Por el principio de los subtipos, una empresa puede **tener** una única lista donde aparezcan tanto jefes como empleados.




```
class empleado
{
  char nombre[30];
  char DNI[9];
  long int sueldo;
public:
  void CambiarSueldo(long int v);
  long int SueldoBase(void)
                    { return sueldo; }
};
```

```
class empleado_produccion: public empleado
{
  int num_piezas;
  int ptas_pieza;

public:
  long int CalcularSueldo(void);
};
```

```
class empleado_comercial: public empleado
{
  int ptas_vendidas;

public:
  long int CalcularSueldo(void);
};
```

```
long int
  empleado_produccion::CalcularSueldo(void)
{
return( SueldoBase() +
                num_piezas * ptas_pieza );
}
```

Equivale a:
`empleado::SueldoBase()`

- Los miembros de la clase empleado se pueden utilizar en las clases derivadas tal y como si hubiesen sido definidos en éstas

Método de la clase base
`empleado::CambiarSueldo()`

```
void main(void)
{ empleado_produccion ep;

 ep.CambiarSueldo(100000);

  cout << "El sueldo es "
       << ep.CalcularSueldo();
}
```

- Modos de acceso
  - private
    - Lo que es private en la clase base no es accesible en la clase derivada.
  - public
    - Lo que definimos como public es accesible desde cualquier parte.
  - protected
    - Lo que definimos como protected en la clase base:
      - es accesible en la clases derivadas,
      - pero no es accesible desde fuera de las clases derivadas o base.

	public	**protected**	**private**
Clase derivada	Accesible	Accesible	No Accesible
Fuera	Accesible	No Accesible	No Accesible

- Tipos de herencia
  - `class <clase_derivada>:<tipo> <clase_base>`
  - public:
    - los modos de acceso a los miembros de la clase base se quedan igual en la clase derivada.
  - protected:
    - Los miembros “public” de la clase base pasan a ser “protected”.
    - El resto se queda igual.
  - private:
    - Todos los miembros de la clase base pasan a ser “private” en la derivada.

```
class padre
{ int x;
public:
  void Sx(int v) { x=v; }
};
class hija: protected padre
{ int h;
public:
  void SetX(int v)
            { // x=v; no permitido
              Sx(v); }
};
```

```
void main(void)
{ hija h;

// h.Sx(5);
// no permitido

h.SetX(5);
}
```

- A veces, interesa cambiar en la subclase la definición de algo que está en la clase base.

```
class base
{
protected:
  int v;
public:
  void Sv(int val) { v=val; }
};

class derivada: public base
{ char v[30]; // re-definición v
public:
  void Sv(int val) // re-definición
  Sv()
     { itoa(val,10,v); }
};
```

- Lo redefinido no desaparece.

```
derivada d;
d.base::Sv(5);
```

- Ejemplo re-definición de métodos

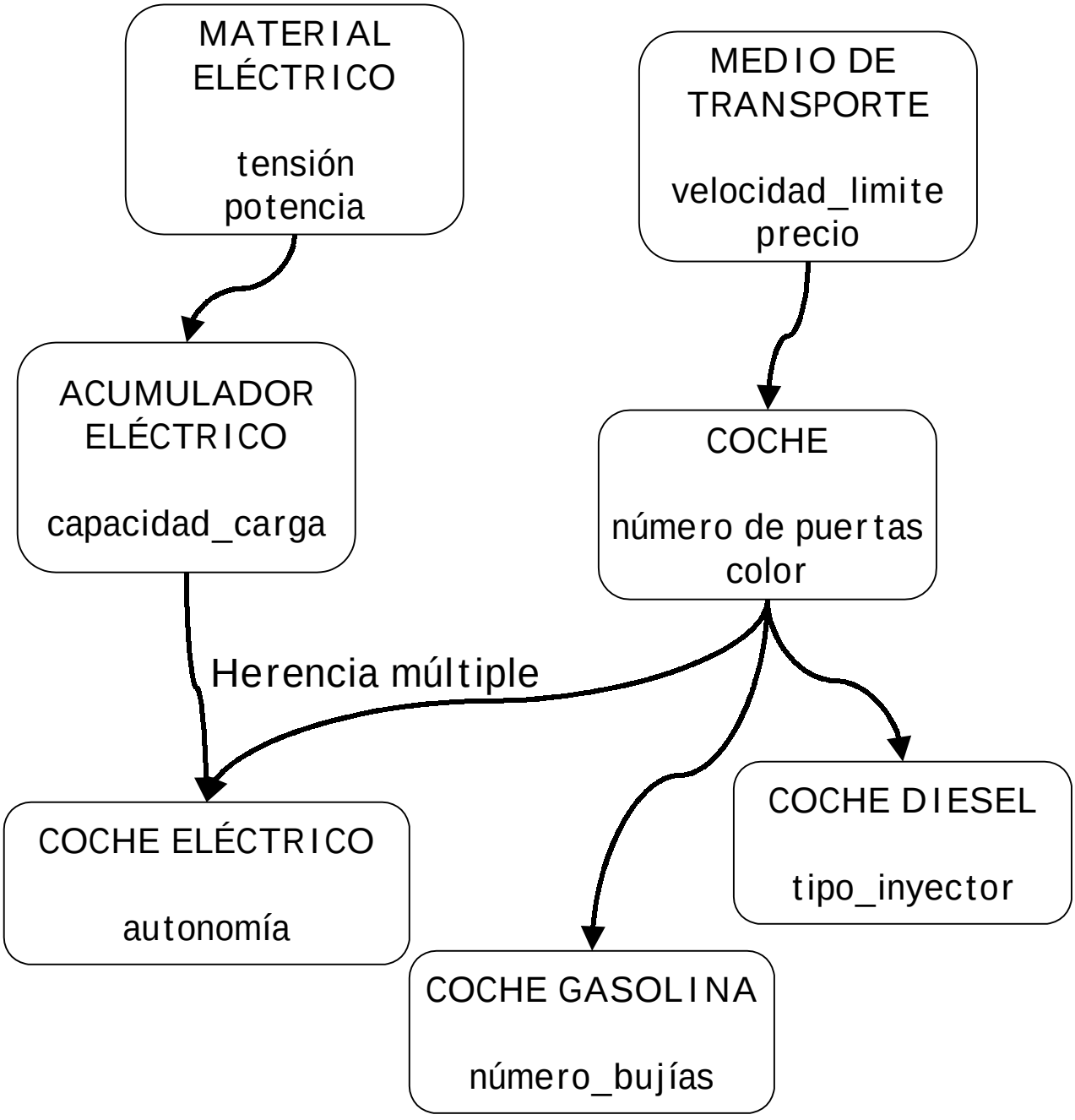
```
class empleado
{ char nombre[30];
  int ptas_hora;
  int num_horas;
public:
  void AsignarNombre(char *nom) {...}
  void AsignarHorasPtas(int ptas,
                        int horas) {...}
  int Sueldo(void)
     { return ( ptas_hora * num_horas );
     }
};

class empleado_comercial: public empleado
{ int ptas_km;
  int num_km;
public:
  void DietasPtas(int ptas,int km) {...}
  int Sueldo(void)
     { return ( empleado::Sueldo() +
                        ptas_km * num_km );
     }
}
```

Clase Empleado

Clase Empleado Comercial

Sino, sería una llamada recursiva a empleado_comercial::Sueldo()

– Una clase derivada hereda las características de más de una clase base.



Herencia múltiple en C++

```
class empleado
{ ...

public:
  ...
  int Sueldo(void);
};
```

```
class comercial
{
  int ptas_km;
  int num_km;

public:
 ...
 int Sueldo(void)
   { return (ptas_km
            * num_km);
   }
}
```

Herencia múltiple

```
class empleado_comercial: public empleado,
                          public comercial
{
public:
  int Sueldo(void)
   { return ( empleado::Sueldo() +
              comercial::Sueldo() );
   }
}
```

- Constructores y destructores en la herencia:
  - Construcción objeto clase derivada:
    - Primero se construye la parte heredada de la clase(s) base.
      - Se ejecutan constructores de las clases base.
    - Por último se ejecuta el código del constructor de la clase derivada.
  - Destrucción objeto clase derivada:
    - el proceso es a la inversa que en la construcción.
    - Se ejecuta primero el destructor de la clase derivada,
    - y a continuación los de las clases base.

– Ejemplo (Constructores y destructores en la herencia):

```
class base1
{ public:
    base1(void) { cout << “base1”; }
    ~base1(void) { cout << “base1 D”; }
}

class base2
{ public:
    base2(void) { cout << “base2”; }
    ~base2(void) { cout << “base2 D”; }
}

class derivada: public base1, public base2
{ public:
    derivada(void) { cout << “derivada”; }
    ~derivada(void) { cout<< “derivada D”;}
}

void main(void)
{ derivada d;
}
```

```
base1
base2
derivada
derivada D
base2 D
base1 D
```

- Constructores y destructores en clases compuestas y derivadas:

```
class base
{ public:
    base(void) { cout << “base”; }
    ~base(void) { cout << “base D”; }
};

class miembro
{ public:
    miembro(void) { cout << “miembro”; }
    ~miembro(void) { cout << “miembro D”; }
};

class derivada: public base
{ miembro m;
public:
    derivada(void) { cout << “derivada”; }
    ~derivada(void) { cout<< “derivada D”;}
};

void main(void)
{ derivada d;
}
```

```
base
miembro
derivada
derivada D
miembro D
base D
```

- Llamadas a los constructores de las clases base:

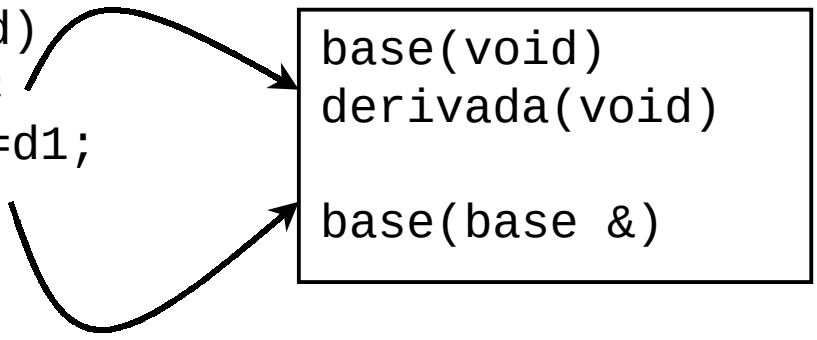
```
class base
{ public:
    base(void) { cout << “base(void)”; }
    base(int a) { cout << “base(int)”; }
};

class derivada: public base
{ public:
   derivada(void) {cout <<“derivada(void)”;}

   derivada(int v): base(v)
       { cout << “derivada(int)”; }
};

void main(void)
{ derivada d1;
  derivada d2=5;
}
```

Llamada al constructor base

```
base(void)
derivada(void)

base(int)
derivada(int)
```

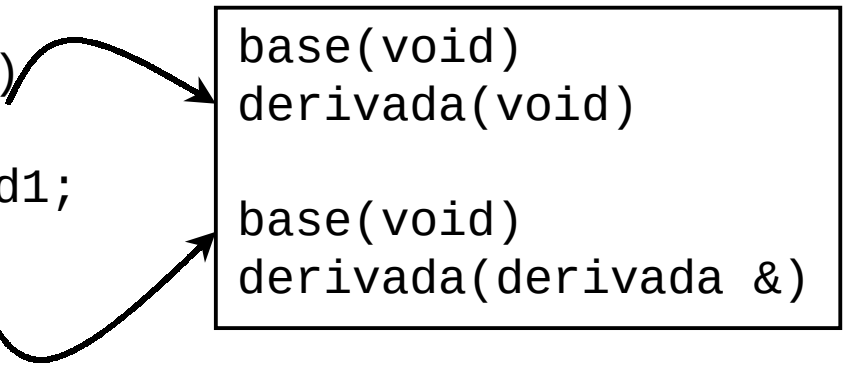
- El constructor copia en la herencia:

```
class base
{ public:
    base(void) { cout << “base(void)”; }
    base(base &b) {cout << “base(abase &)”;
  }
};


class derivada: public base
{ public:
   derivada(void) {cout <<“derivada(void)”;}
};


void main(void)
{ derivada d1;
  derivada d2=d1;
}
```



– Si definimos constructor copia en la clase derivada:

```
class base
{ public:
    base(void) { cout << “base(void)”; }
    base(base &b) {cout << “base(abase &)”;
   }
};

class derivada: public base
{ public:
   derivada(void) {cout <<“derivada(void)”;}
   derivada(derivada &d)
       { cout << “derivada(derivada &)”; }

};

void main(void)
{ derivada d1;
  derivada d2=d1;
}
```

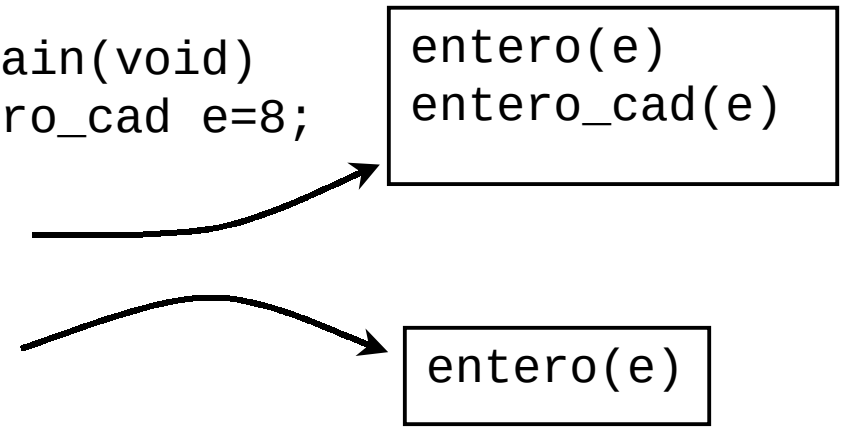


- Ejemplo de subtipado y constructor copia en herencia:

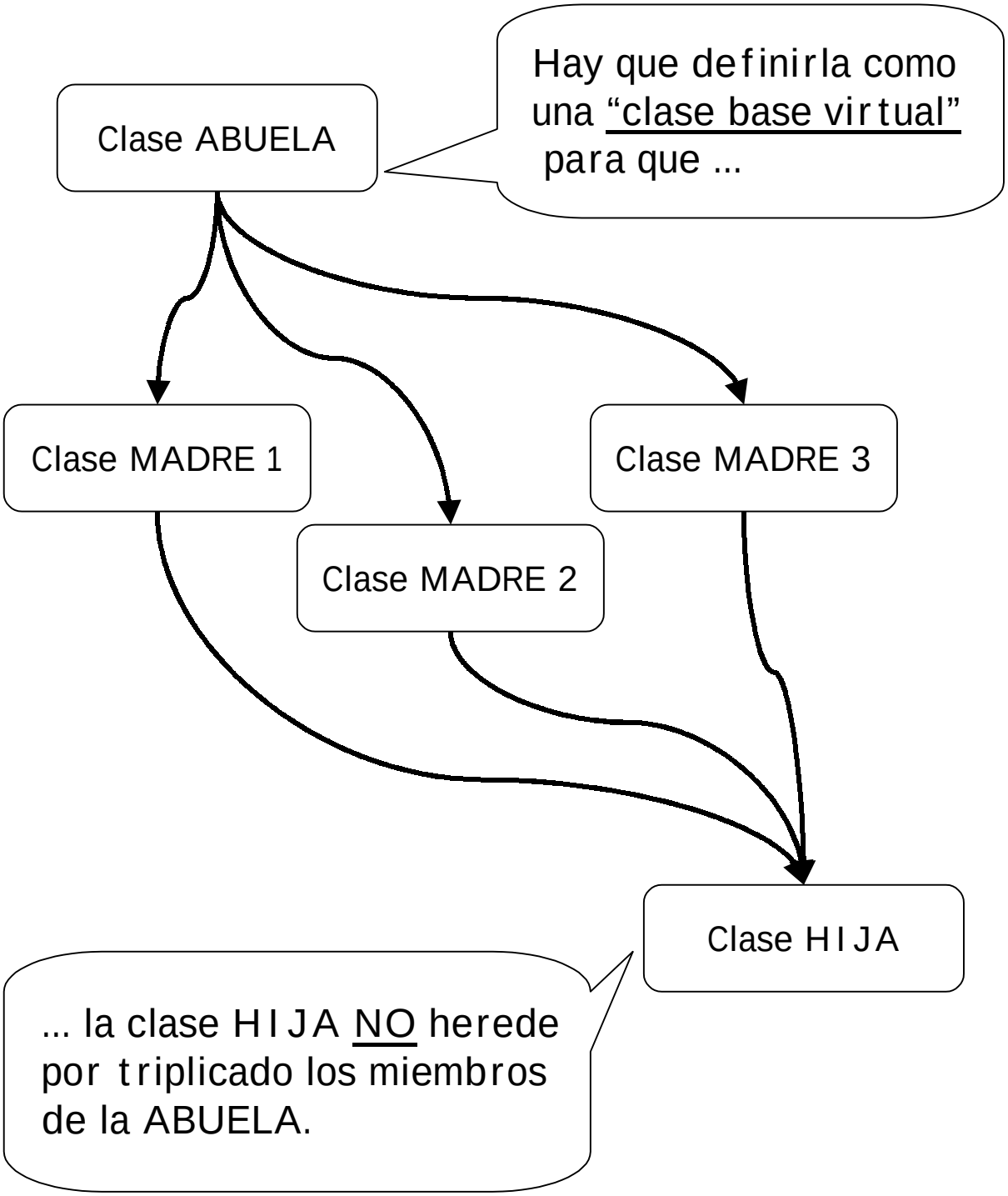
```
class entero
{ int v;

public:
  entero(int val=0) { v=val; }
  void Sv(int val) {v=val; }
  int Gv(void) { return v; }
};

class entero_cad : public entero
{ char *cad;

public:
  entero_cad(int val=0): entero(val)
   { char aux[30];

    itoa(val,10,aux);
    cad=new char[strlen(aux)+1];
    strcpy(cad,aux);
   }

  entero_cad(entero_cad &e);
};
```

```
// constructor copia
entero_cad ::
  entero_cad(entero_cad &e) : entero(e)
{ char aux[30];

itoa(e.v,10,aux);
cad=new char[strlen(aux)+1];
strcpy(cad,aux);
}

void f1(entero_cad c)
{ cout << "valor" << c.Gv();
}

void f2(entero t)
{ cout << "valor" << t.Gv();
  cout << "suma" << t.Gv()+50;
}

void main(void)
{ entero_cad e=8;

f1(e);

f2(e);
}
```

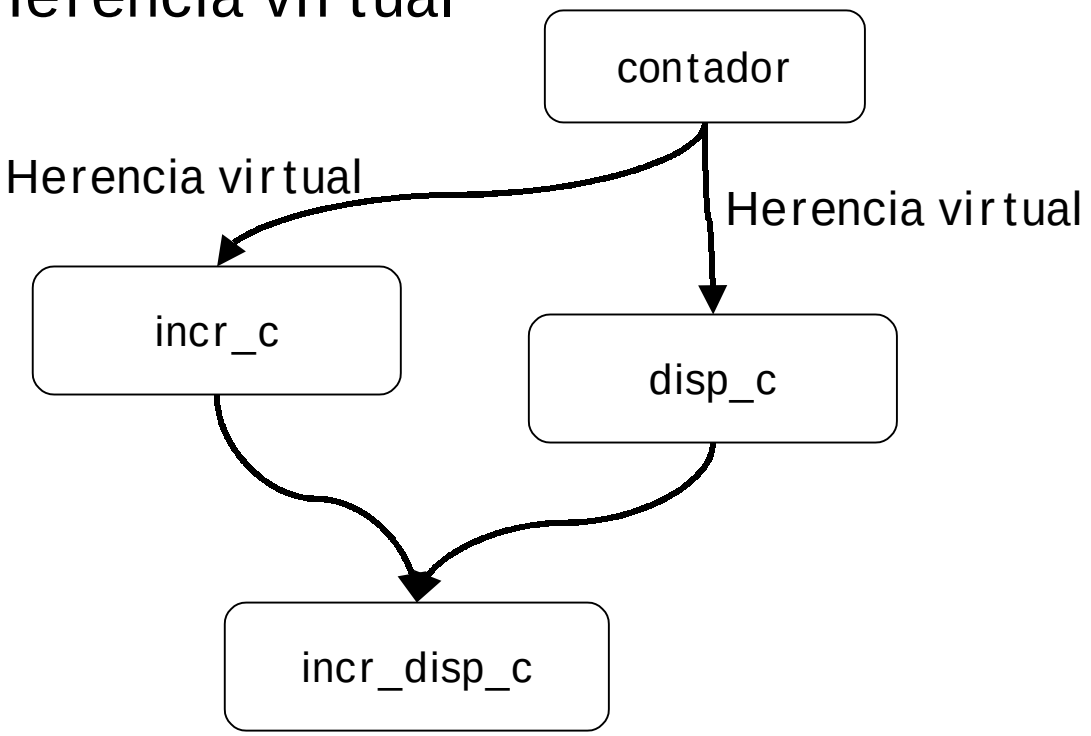
Llamada al constructor copia. En este caso, al que hay definido por defecto en la clase entero.



- Clase base virtual



- Herencia virtual



- Si una clase base virtual define constructores, debe proporcionar:
  - un constructor sin parámetros o
  - un constructor que admita valores por defecto para todos los parámetros.

- Las clases derivadas de una clase base virtual tienen que:
  - ser definidas como herencia virtual.

```
class contador  //clase base virtual
{ protected:
    int cont;
  public:
    contador(int c=0) { cont=c; }
    void Reset(int c=0) { cont=c; }
};

class incr_c: virtual public contador
{ public:
    incr_c(): contador(100) {}
    void Increment() { cont++; }
};

class disp_c: virtual public contador
{ public:
    disp_c(): contador(200) {}
    void mostrar() { cout << cont; }
};

class incr_disp_c: public incr_c,
                   public disp_c
{ public:
    incr_disp_c(): contador(300) {}
};
```

- Punteros a clases derivadas

```
class A
{ protected:
    int v;
  public:
    void Sv(int x) { v=x; }
    int Gv(void) { return v; }
};

class B: public A
{ public:
    B(void) { v=0; }
    void Sv(int x) { v+=x; }
};

void main(void)
{ B vb[10];
  A *pa;

for(int i=0; pa=vb; i<10; i++, pa++)
  { int a;
    cin >> a;
    cout << pa->Gv(); //de la clase A
    pa->Sv(a); //se ejecuta A::Sv(), aunque
               //el objeto sea de la clase B
  }
}
```