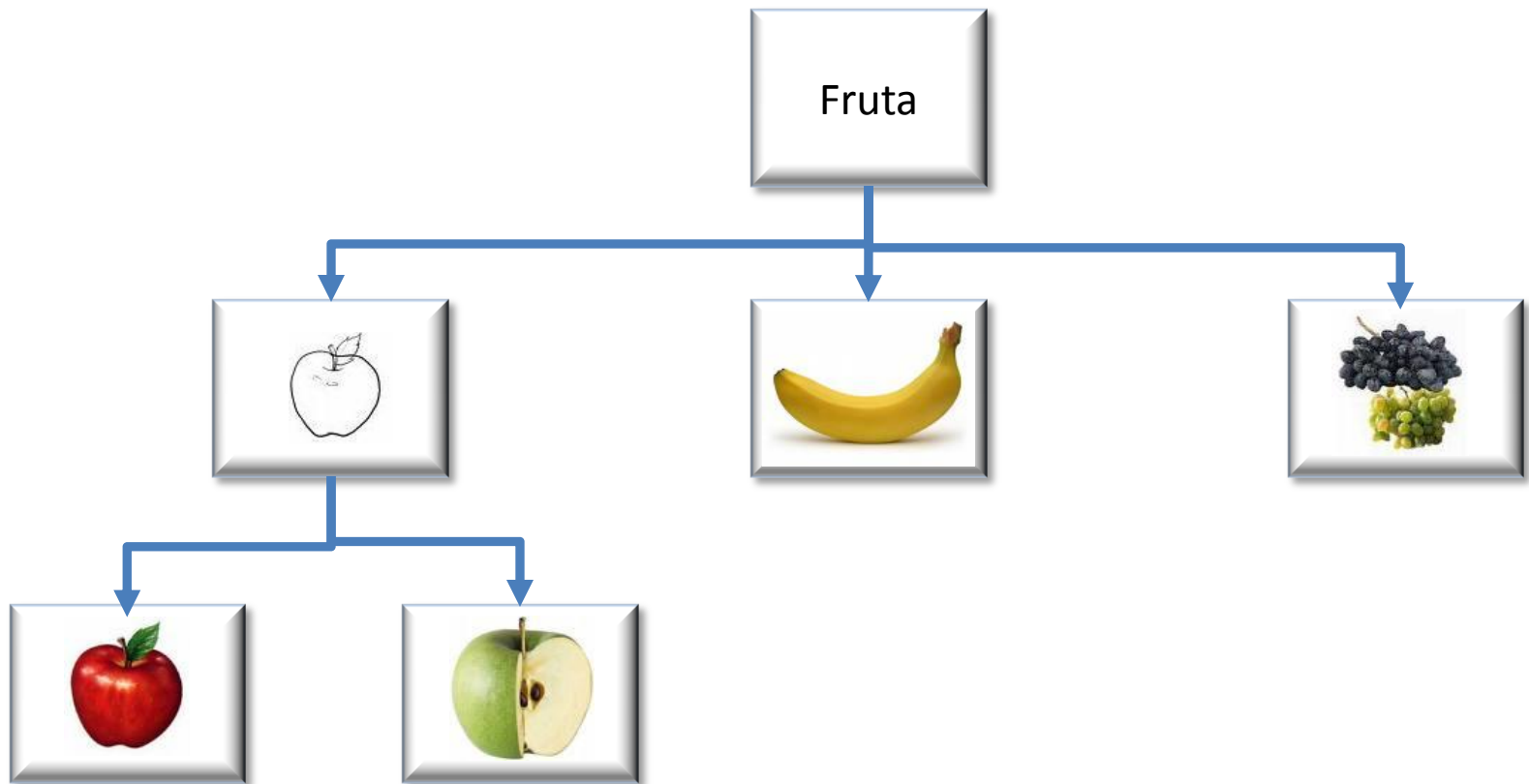


# C++

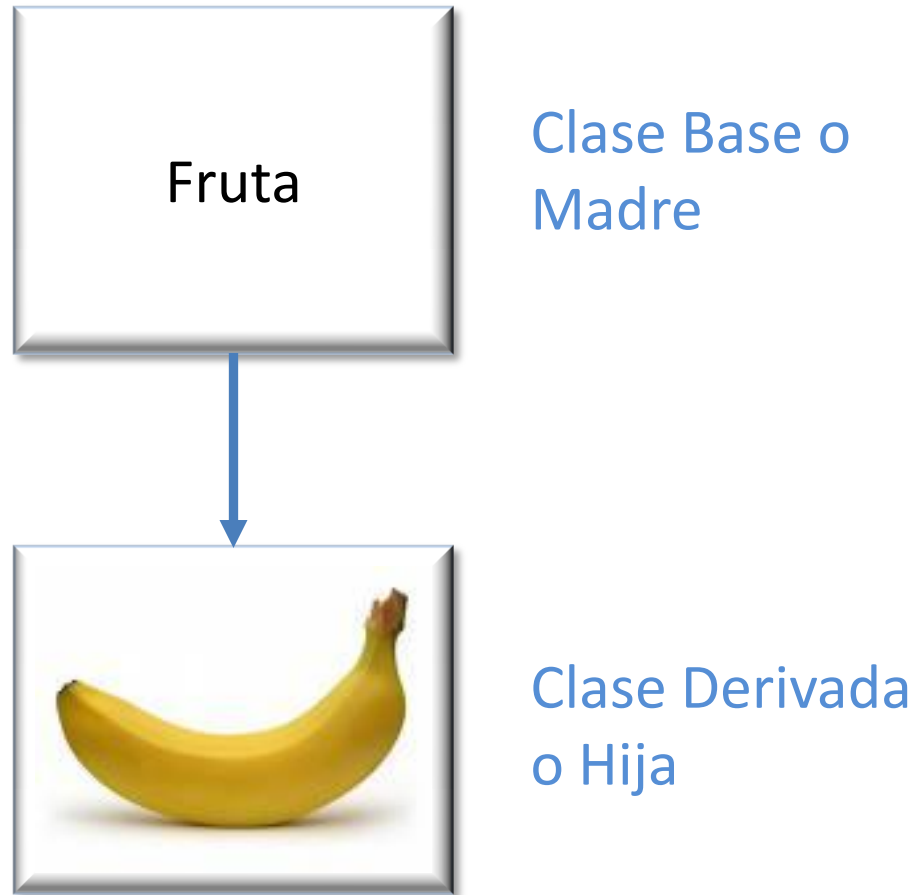
## Herencia y Polimorfismo

# Herencia

- Es el proceso mediante el cual un objeto puede adquirir las características de otro.



# Relaciones de Herencia



# Sintaxis Herencia

```
class nombre_clase_derivada : acceso nombre_clase_base
{
    // cuerpo de la clase derivada
};
```

- *nombre\_clase\_derivada*: Clase derivada o hija, la cual se está declarando.
- *acceso*: Una de las palabras clave **public**, **private** o **protected**.
- *nombre\_clase\_base*: Clase base o madre, la cual debe haber sido declarada.

# Acceso y Herencia

- **Dentro** de la clase derivada se puede acceder a todos los miembros de la clase base que NO sean **private**, siempre que esta NO sea heredada como **private**.
- **Fuera** de la clase derivada:

		Acceso Clase Base		
		public	protected	private
Acceso Miembros Clase Base	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private

# Sobre-escritura de Funciones

- Las clases derivadas pueden sobre-escribir funciones existentes en la clase base.
- La función original puede ser usada con el operador de alcance (::).

# Ejemplo

```
class lineal {
private:
double m;
public:
void set_pend(double pend) { m=pend; }
double f(double x) { return m*x; }
};

class lineal_afin: public lineal {
private:
double n;
public:
void set_corte(double corte) { n=corte; }
double f(double x) { return lineal::f(x)+n; }
// si m fuese public o protected podríamos acceder a ella (return m*x+n;)
};

int main()
{
lineal_afin la;
la.set_pend(1);
la.set_corte(2);
printf("%lf\n", la.f(3)); // muestra 5 (3*1+2)
printf("%lf\n", la.lineal::f(3)); // muestra 3 (3*1)
printf("%lf\n", ((lineal)la).f(3)); // muestra 3 (3*1)
}
```

# Constructores, Destruktores y Herencia

- Los **constructores** de las clases base son llamados **antes** que el de la clase derivada.
- Los **destruictores** de las clases base son llamados **después** que el de la clase derivada.



# Ejemplo

```
class base {  
public:  
base() { printf("Construyendo base\n"); }  
~base() { printf("Destruyendo base\n"); }  
};  
class derivada: public base {  
public:  
derivada() { printf("Construyendo derivada\n"); }  
~derivada() { printf("Destruyendo derivada\n"); }  
};  
int main()  
{  
derivada ob;  
return 0;  
}
```

Construyendo base  
Construyendo derivada  
Destruyendo derivada  
Destruyendo base

# Pasar Argumentos a Constructores

- En el constructor de una clase derivada, se puede pasar argumentos a los constructores de las clases base (y por lo tanto, seleccionar cuál constructor de la clase base se ejecutará).

# Sintaxis Paso Argumentos

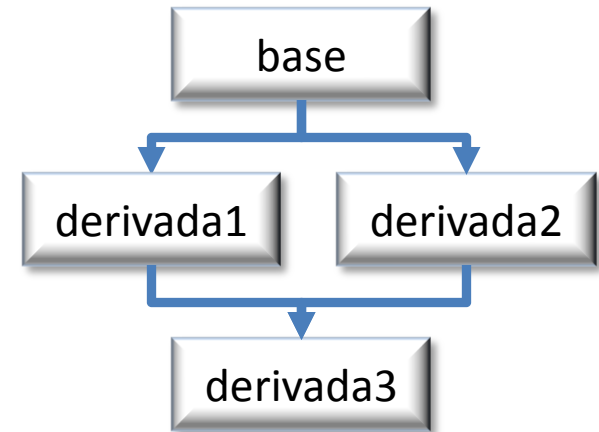
```
derivada::derivada(lista_arg)
: base1(lista_arg_1)
, base2(lista_arg_2)
// ...
, baseN(lista_arg_N)
{
// cuerpo del constructor de la clase derivada
}
```

- *derivada*: Clase derivada, de la cual se está definiendo un constructor.
- *base1,...,baseN*: Las clases de las cuales hereda derivada.
- *lista\_arg*: Lista de argumentos del constructor (de la clase derivada) que se está definiendo.
- *lista\_arg\_i*: Lista de argumentos pasados al constructor de la clase base i.

# Clases Base Virtuales

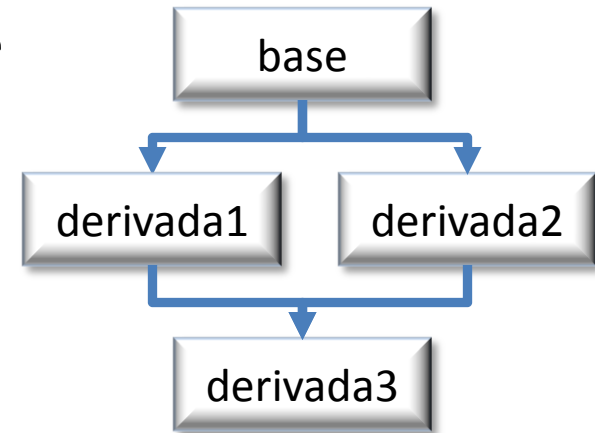
- Consideremos el siguiente ejemplo:

```
class base {  
public:  
int i;  
};  
// derivada1 y derivada2 heredan de base.  
class derivada1: public base {  
public:  
int j;  
};  
class derivada2 : public base {  
public:  
int k;  
};  
// derivada3 hereda de derivada1 y de derivada2.  
class derivada3 : public derivada1, public derivada2 {  
public:  
int suma;  
};
```



# Clases Base Virtuales(2)

- En el ejemplo anterior se crean **2 copias** de la clase base en la clase derivada3, por lo tanto al usar cualquier miembro de base, se produce una ambigüedad.
- La ambigüedad puede ser resuelta explícitamente:

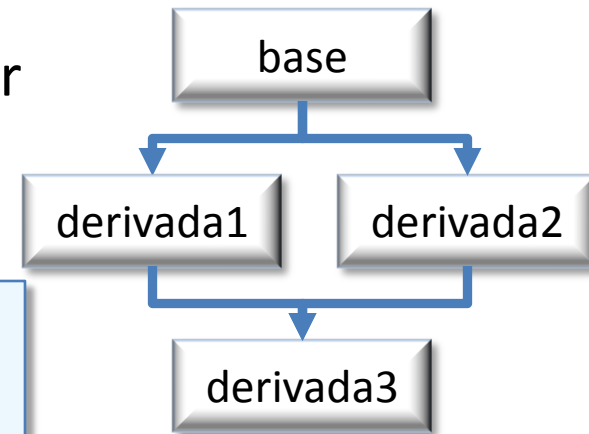


```
int main()
{
    derivada3 ob;
    ob.derivada1::i = 10;
    ob.j = 20;
    ob.k = 30;
    ob.sum = ob.derivada1::i + ob.j + ob.k;
    return 0;
}
```

# Clases Base Virtuales(3)

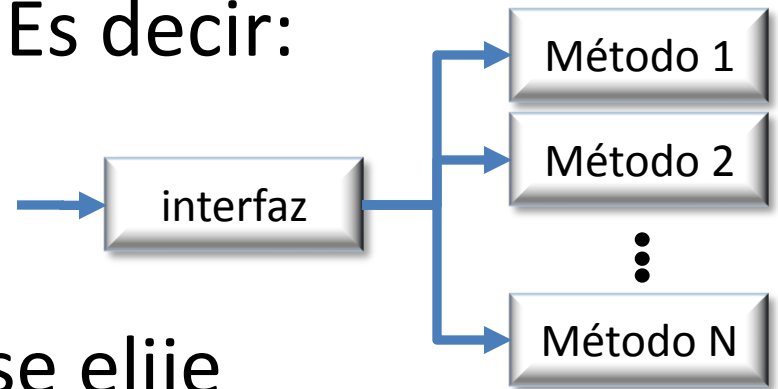
- Sin embargo, muchas veces deseamos tener sólo **una copia** de la clase base. Entonces debemos usar **clases base virtuales**.

```
class base {  
public:  
    int i;  
};  
// derivada1 y derivada2 heredan de base (como virtual).  
class derivada1: virtual public base {  
public:  
    int j;  
};  
class derivada2 : virtual public base {  
public:  
    int k;  
};  
// derivada3 hereda de derivada1 y de derivada2.  
class derivada3 : public derivada1, public derivada2 {  
public:  
    int suma;  
};
```



# Polimorfismo

- Es el atributo que permite a una interfaz controlar el acceso a un tipo general de acciones. Es decir:
  - Una interfaz
  - Múltiples métodos
- La acción que se ejecuta se elige de acuerdo a la situación.
- Es una característica fundamental de la programación orientada a objetos (OOP).



# Polimorfismo en tiempos de compilación y ejecución

- El polimorfismo en **tiempo de compilación** se logra a través de la **sobre-escritura** de funciones y operadores.
- El polimorfismo en **tiempo de ejecución** se logra a través de la **herencia** y de las **funciones virtuales**.



# Funciones Virtuales

- Es una función miembro de una clase base que puede ser sobre-escrita por las clases derivadas de forma más poderosa de lo habitual.
- Su verdadero potencial se expresa cuando usamos punteros a la clase base.

# Declaración de Funciones Virtuales

```
class base {
public:
virtual void funcion_virt() {
printf("Esta es funcion_virt() de base.\n");
}
};

class derivada1 : public base {
public:
void funcion_virt() {
printf("Esta es funcion_virt() de derivada1.\n");
}
};

class derivada2 : public base {
public:
void funcion_virt() {
printf("Esta es funcion_virt() de derivada2.\n");
}
};
```

# Ejemplo

```
int main()
{
base *p, b;
derivada1 d1;
derivada2 d2;
// apunta a base
p = &b;
p->funcion_virt(); // accede a funcion_virt de base
// apunta a derivada1
p = &d1;
p->funcion_virt(); // accede a funcion_virt de derivada1
// apunta a derivada2
p = &d2;
p->funcion_virt(); // accede a funcion_virt de derivada2
return 0;
}
```

Esta es funcion\_virt() de base.

Esta es funcion\_virt() de derivada1.

Esta es funcion\_virt() de derivada2.

# Funciones Virtuales y Referencias

- Cuando una función virtual se llama a través de una referencia a una clase base, se usará la versión de la función determinada por la clase del objeto referenciado.

# Mismo Ejemplo Versión 2

```
// Recibe como parámetro una referencia a base.
void f(base &r) {
    r.funcion_virt();
}
int main()
{
    base b;
    derivada1 d1;
    derivada2 d2;
    f(b); // pasa un objeto base a f()
    f(d1); // pasa un objeto derivada1 a f()
    f(d2); // pasa un objeto derivada2 a f()
    return 0;
}
```

Esta es funcion\_virt() de base.

Esta es funcion\_virt() de derivada1.

Esta es funcion\_virt() de derivada2.

# Herencia de Funciones Virtuales

- Si una función virtual no es sobre-escrita:
  - Se hereda la de la clase base.
  - Se hereda el atributo de virtual.

# Funciones Virtuales Puras

- Cuando se tiene una clase base que:
  - No tiene suficiente información para implementar una función, pero
  - Se desea obligar a todas las derivadas a implementarla.
- Se puede usar una **función virtual pura**.

# Ejemplo Funciones Virtuales Puras

```
class numero {
protected:
int val;
public:
void setvalor(int i) { val = i; }
//mostrar() es una función virtual pura
virtual void mostrar() = 0;
};

class numero_hex : public numero {
public:
void mostrar() {
printf("%x\n",i);
}
};

class numero_dec : public numero {
public:
void mostrar() {
printf("%d\n",i);
}
};
```

```
int main()
{
numero_dec d;
numero_hex h;
d.setval(20);
// muestra 20 (decimal)
d.mostrar();
h.setval(20);
// muestra 14 (hexadecimal)
h.mostrar();
// numero n
// error! número es abstracta
return 0;
}
```



# Clases Abstractas

- Una clase que tiene al menos una función virtual pura es una **clase abstracta**.
- No se pueden declarar instancias de una clase abstracta, ya que no está totalmente implementada.

# Utilidad de Clases Abstractas

- La utilidad de las clases abstractas se reduce a ser clases base de otras clases.
- Las clases derivadas de una clase abstracta deben implementar al menos sus funciones virtuales puras.
- Si bien no se pueden crear objetos de una clase abstracta, sí se pueden crear punteros a una clase abstracta, lo cual permite usar polimorfismo en tiempo de ejecución.