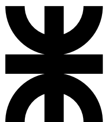
Informática II

Herencia en C++

Gonzalo F. Pérez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2018 –

Introducción

Herencia

- Es una de las principales características para la reutilización de software
- Las nuevas clases se crean a partir de clases existentes, utilizando sus atributos y comportamiento, y agregando nuevas capacidades

Introducción

Herencia

- Es una de las principales características para la reutilización de software
- Las nuevas clases se crean a partir de clases existentes, utilizando sus atributos y comportamiento, y agregando nuevas capacidades

- Cuando se crea una nueva clase, esta puede *heredar* los datos miembros y funciones miembros de una *clase base* definida previamente

Introducción

Herencia

- Es una de las principales características para la reutilización de software
- Las nuevas clases se crean a partir de clases existentes, utilizando sus atributos y comportamiento, y agregando nuevas capacidades

- Cuando se crea una nueva clase, esta puede *heredar* los datos miembros y funciones miembros de una *clase base* definida previamente
- A esta nueva clase se la conoce como *clase derivada*

Introducción

Herencia

- Es una de las principales características para la reutilización de software
- Las nuevas clases se crean a partir de clases existentes, utilizando sus atributos y comportamiento, y agregando nuevas capacidades

- Cuando se crea una nueva clase, esta puede *heredar* los datos miembros y funciones miembros de una *clase base* definida previamente
- A esta nueva clase se la conoce como *clase derivada*
- Cada clase derivada puede a su vez ser clase base de otras clases

Introducción

Herencia

- Es una de las principales características para la reutilización de software
- Las nuevas clases se crean a partir de clases existentes, utilizando sus atributos y comportamiento, y agregando nuevas capacidades

- Cuando se crea una nueva clase, esta puede *heredar* los datos miembros y funciones miembros de una *clase base* definida previamente
- A esta nueva clase se la conoce como *clase derivada*
- Cada clase derivada puede a su vez ser clase base de otras clases
- En la *herencia simple* una clase deriva de una única clase base

Introducción

Herencia

- Es una de las principales características para la reutilización de software
- Las nuevas clases se crean a partir de clases existentes, utilizando sus atributos y comportamiento, y agregando nuevas capacidades

- Cuando se crea una nueva clase, esta puede *heredar* los datos miembros y funciones miembros de una *clase base* definida previamente
- A esta nueva clase se la conoce como *clase derivada*
- Cada clase derivada puede a su vez ser clase base de otras clases
- En la *herencia simple* una clase deriva de una única clase base
- En la *herencia múltiple* una clase se deriva de varias clases bases (pudiendo no estar relacionadas entre sí)

Introducción

Herencia

- Es una de las principales características para la reutilización de software
- Las nuevas clases se crean a partir de clases existentes, utilizando sus atributos y comportamiento, y agregando nuevas capacidades

- Cuando se crea una nueva clase, esta puede *heredar* los datos miembros y funciones miembros de una *clase base* definida previamente
- A esta nueva clase se la conoce como *clase derivada*
- Cada clase derivada puede a su vez ser clase base de otras clases
- En la *herencia simple* una clase deriva de una única clase base
- En la *herencia múltiple* una clase se deriva de varias clases bases (pudiendo no estar relacionadas entre sí)

- Una clase derivada puede agregar nuevos datos y funciones miembros

Introducción

Herencia

- Es una de las principales características para la reutilización de software
- Las nuevas clases se crean a partir de clases existentes, utilizando sus atributos y comportamiento, y agregando nuevas capacidades

- Cuando se crea una nueva clase, esta puede *heredar* los datos miembros y funciones miembros de una *clase base* definida previamente
- A esta nueva clase se la conoce como *clase derivada*
- Cada clase derivada puede a su vez ser clase base de otras clases
- En la *herencia simple* una clase deriva de una única clase base
- En la *herencia múltiple* una clase se deriva de varias clases bases (pudiendo no estar relacionadas entre sí)

- Una clase derivada puede agregar nuevos datos y funciones miembros
- Por lo que, en general, una clase derivada es "más grande" que su clase base

Introducción

Herencia

- Es una de las principales características para la reutilización de software
- Las nuevas clases se crean a partir de clases existentes, utilizando sus atributos y comportamiento, y agregando nuevas capacidades

- Cuando se crea una nueva clase, esta puede *heredar* los datos miembros y funciones miembros de una *clase base* definida previamente
- A esta nueva clase se la conoce como *clase derivada*
- Cada clase derivada puede a su vez ser clase base de otras clases
- En la *herencia simple* una clase deriva de una única clase base
- En la *herencia múltiple* una clase se deriva de varias clases bases (pudiendo no estar relacionadas entre sí)

- Una clase derivada puede agregar nuevos datos y funciones miembros
- Por lo que, en general, una clase derivada es "más grande" que su clase base
- Una clase derivada es más específica que su clase base y representa a un grupo más pequeño de objetos

Introducción

En C++ se puede definir tres tipos de herencias: *pública*, *protegida* y *privada*

Introducción

En C++ se puede definir tres tipos de herencias: *pública*, *protegida* y *privada*

- Con la herencia pública, cada objeto de la clase derivada es también un objeto de la clase base
- Sin embargo, lo inverso no es verdad, los objetos de la clase base no son objetos de la clase derivada

Introducción

En C++ se puede definir tres tipos de herencias: *pública*, *protegida* y *privada*

- Con la herencia pública, cada objeto de la clase derivada es también un objeto de la clase base
- Sin embargo, lo inverso no es verdad, los objetos de la clase base no son objetos de la clase derivada

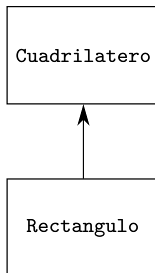
Por ejemplo: Un rectángulo *es un* cuadrilátero (como lo es un cuadradro, un paralelogramo o un trapezoide), pero no se puede decir que un cuadrilátero *es un* rectángulo

Introducción

En C++ se puede definir tres tipos de herencias: *pública*, *protegida* y *privada*

- Con la herencia pública, cada objeto de la clase derivada es también un objeto de la clase base
- Sin embargo, lo inverso no es verdad, los objetos de la clase base no son objetos de la clase derivada

Por ejemplo: Un rectángulo *es un* cuadrilátero (como lo es un cuadradro, un paralelogramo o un trapezoide), pero no se puede decir que un cuadrilátero *es un* rectángulo

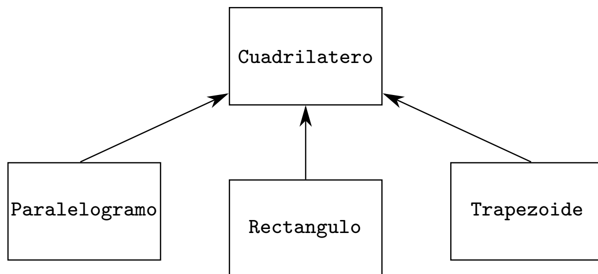


Introducción

En C++ se puede definir tres tipos de herencias: *pública*, *protegida* y *privada*

- Con la herencia pública, cada objeto de la clase derivada es también un objeto de la clase base
- Sin embargo, lo inverso no es verdad, los objetos de la clase base no son objetos de la clase derivada

Por ejemplo: Un rectángulo *es un* cuadrilátero (como lo es un cuadradro, un paralelogramo o un trapezoide), pero no se puede decir que un cuadrilátero *es un* rectángulo



Introducción

En C++ se puede definir tres tipos de herencias: *pública*, *protegida* y *privada*

- Con la herencia pública, cada objeto de la clase derivada es también un objeto de la clase base
- Sin embargo, lo inverso no es verdad, los objetos de la clase base no son objetos de la clase derivada

Por ejemplo: Un rectángulo *es un* cuadrilátero (como lo es un cuadradro, un paralelogramo o un trapezoide), pero no se puede decir que un cuadrilátero *es un* rectángulo

Ejemplos de herencia

`Estudiante`: `EstudianteUniversitario`, `EstudianteTitulado`

`Figura`: `Circulo`, `Triangulo`, `Rectangulo`

`Prestamo`: `PrestamoAutomovil`, `PrestamoMejorCasa`, `PrestamoHipotecario`

`Empleado`: `EmpleadoDocente`, `EmpleadoAdministrativo`

`Cuenta`: `CuentaCheques`, `CuentaAhorros`

Introducción

En C++ se puede definir tres tipos de herencias: *pública*, *protegida* y *privada*

- Con la herencia pública, cada objeto de la clase derivada es también un objeto de la clase base
- Sin embargo, lo inverso no es verdad, los objetos de la clase base no son objetos de la clase derivada

Por ejemplo: Un rectángulo *es un* cuadrilátero (como lo es un cuadradro, un paralelogramo o un trapezoide), pero no se puede decir que un cuadrilátero *es un* rectángulo

Ejemplos de herencia

`Estudiante`: `EstudianteUniversitario`, `EstudianteTitulado`

`Figura`: `Circulo`, `Triangulo`, `Rectangulo`

`Prestamo`: `PrestamoAutomovil`, `PrestamoMejorCasa`, `PrestamoHipotecario`

`Empleado`: `EmpleadoDocente`, `EmpleadoAdministrativo`

`Cuenta`: `CuentaCheques`, `CuentaAhorros`

Se utiliza una nueva forma de control de acceso a miembro, el acceso protegido –`protected` (además de `private` y `public`)

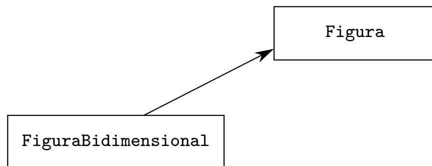
Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

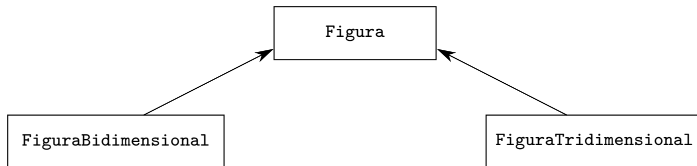
Ejemplo: jerarquía de clases `Figura`



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

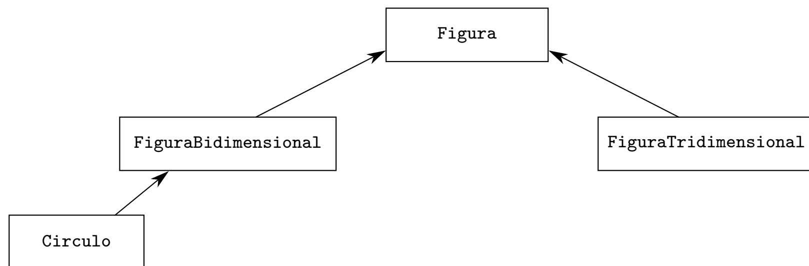
Ejemplo: jerarquía de clases `Figura`



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

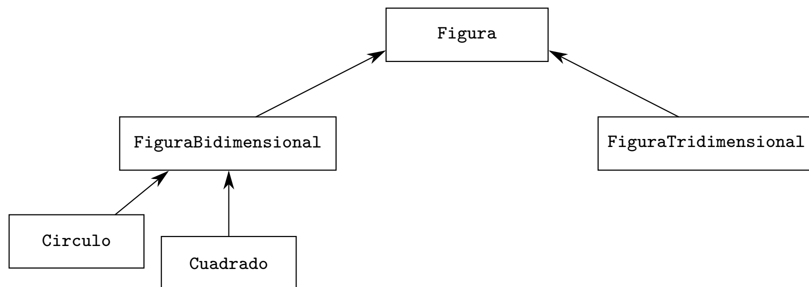
Ejemplo: jerarquía de clases `Figura`



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

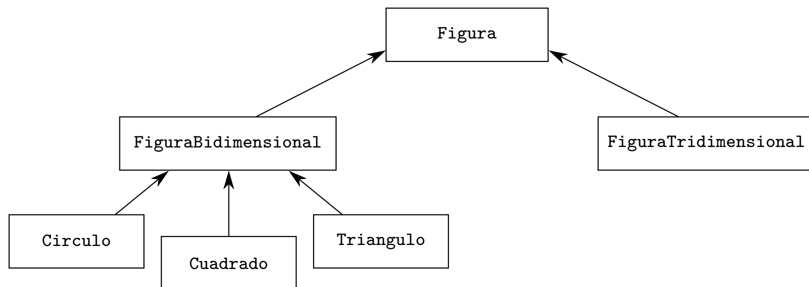
Ejemplo: jerarquía de clases `Figura`



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

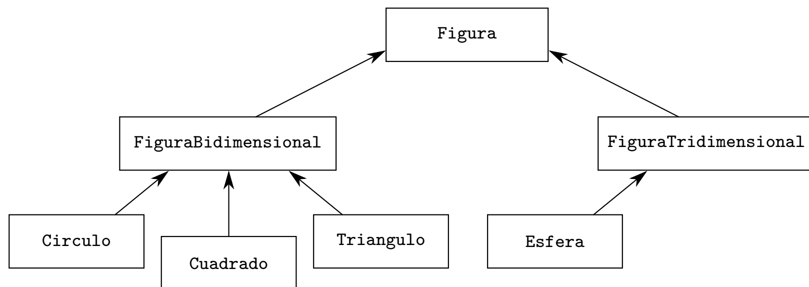
Ejemplo: jerarquía de clases `Figura`



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

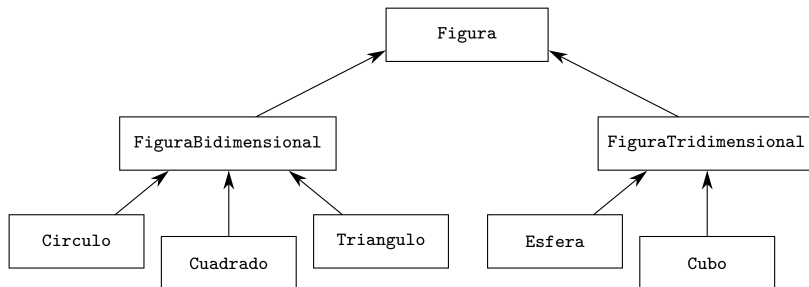
Ejemplo: jerarquía de clases `Figura`



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

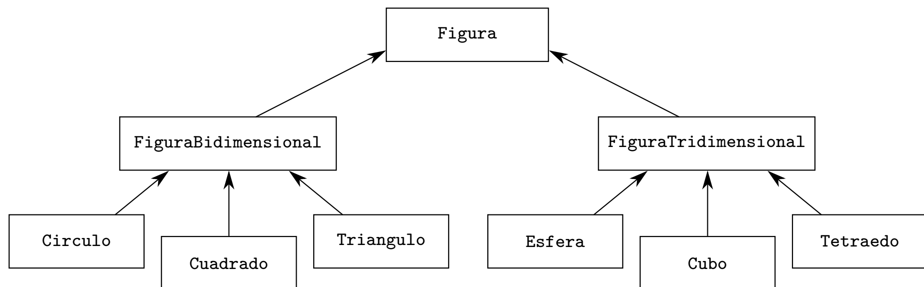
Ejemplo: jerarquía de clases `Figura`



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

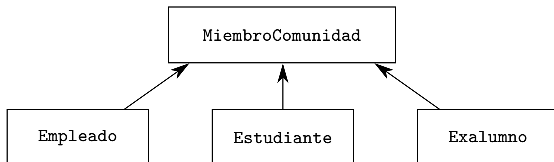
Ejemplo: jerarquía de clases `Figura`



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

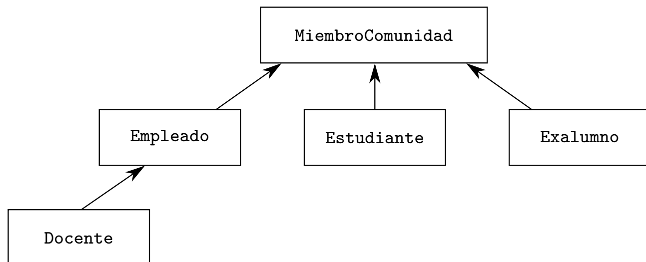
Ejemplo: comunidad universitaria



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

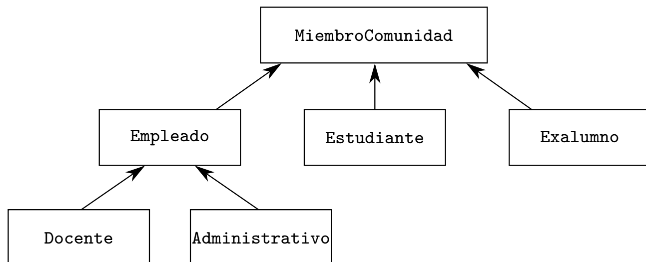
Ejemplo: comunidad universitaria



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

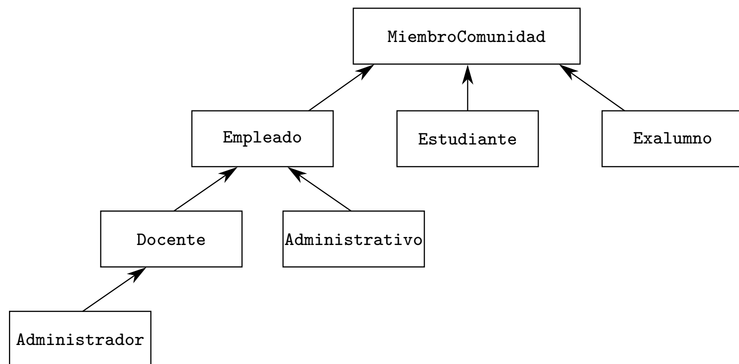
Ejemplo: comunidad universitaria



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

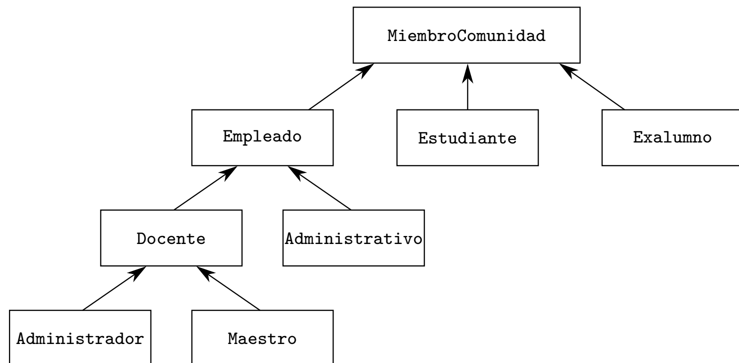
Ejemplo: comunidad universitaria



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

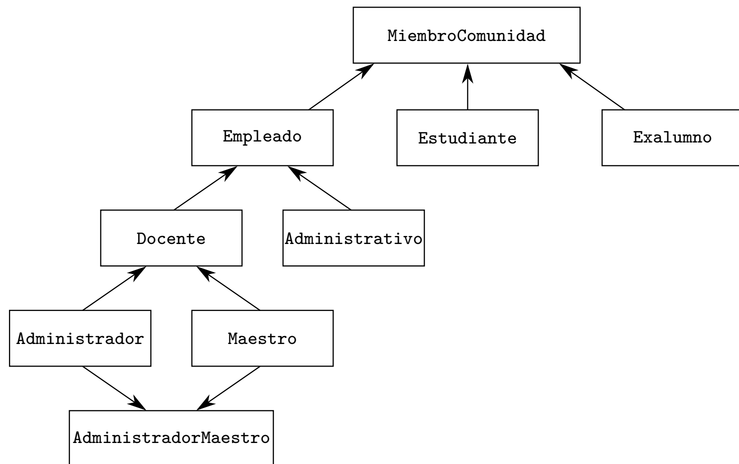
Ejemplo: comunidad universitaria



Herencia – Ejemplos

La herencia forma una estructura jerárquica en forma de árbol

Ejemplo: comunidad universitaria



Herencia – Definición

Para indicar que la clase `TrabajadorComision` deriva de la clase `Empleado`, la clase `TrabajadorComision` se define como

Herencia – Definición

Para indicar que la clase **TrabajadorComision** deriva de la clase **Empleado**, la clase **TrabajadorComision** se define como

```
class TrabajadorComision : public Empleado {
  . . .
};
```

Herencia – Definición

Para indicar que la clase **TrabajadorComision** deriva de la clase **Empleado**, la clase **TrabajadorComision** se define como

```
class TrabajadorComision : public Empleado {
  . . .
};
```

- Esto se conoce como herencia pública (la más utilizada)
- Los miembros públicos y protegidos se heredan como miembros públicos y protegidos, respectivamente
- Las funciones amigas no se heredan

Miembros `protected`

- El acceso `protected` es un nivel intermedio de protección entre acceso público y privado
- Los miembros `protected` de la clase base se acceden mediante miembros y amigas de la clase base, y por medio de miembros y amigas de la clase derivada
- La clase derivada puede acceder a los miembros públicos y protegidos de la clase base utilizando su nombre

Miembros `protected`

- El acceso `protected` es un nivel intermedio de protección entre acceso público y privado
- Los miembros `protected` de la clase base se acceden mediante miembros y amigas de la clase base, y por medio de miembros y amigas de la clase derivada
- La clase derivada puede acceder a los miembros públicos y protegidos de la clase base utilizando su nombre

Recordar que

- Se puede acceder a los miembros `public` de la clase base desde todas las funciones del programa
- Los miembros `private` son accesible solo de las funciones miembros y `friend` de la clase base

Ejemplo de herencia – Clase Punto y Circulo

```
1  class Punto {
2
3
4    public:
5      Punto(int = 0, int = 0); // constructor predeterminado
6      void establecePunto(int , int ); // establece coordenadas
7      int obtienex() const { return x; } // obtiene la coordenada x
8      int obtieneY() const { return y; } // obtiene la coordenada y
9
10
11
12 };
```

Ejemplo de herencia – Clase Punto y Circulo

```
1  class Punto {
2     friend ostream &operator<<( ostream & , const Punto & ); // cout << p;
3
4    public:
5     Punto(int = 0, int = 0); // constructor predeterminado
6     void establecePunto(int , int ); // establece coordenadas
7     int obtienex() const { return x; } // obtiene la coordenada x
8     int obtieneY() const { return y; } // obtiene la coordenada y
9
10
11
12 };
```

Ejemplo de herencia – Clase Punto y Circulo

```
1  class Punto {
2     friend ostream &operator<<( ostream & , const Punto & ); // cout << p;
3
4   public:
5     Punto(int = 0, int = 0); // constructor predeterminado
6     void establecePunto(int , int ); // establece coordenadas
7     int obtienex() const { return x; } // obtiene la coordenada x
8     int obtieneY() const { return y; } // obtiene la coordenada y
9
10  protected: // accesible para las clase derivadas
11    int x, y; // las coordenadas x e y de Punto
12 };
```

Ejemplo de herencia – Clase Punto y Circulo

```
1  class Punto {
2     friend ostream &operator<<( ostream & , const Punto & ); // cout << p;
3
4    public:
5     Punto(int = 0, int = 0); // constructor predeterminado
6     void establecePunto(int , int ); // establece coordenadas
7     int obtienex() const { return x; } // obtiene la coordenada x
8     int obtieneY() const { return y; } // obtiene la coordenada y
9
10   protected: // accesible para las clase derivadas
11    int x, y; // las coordenadas x e y de Punto
12 };
```

```
1  class Circulo : public Punto { // Circulo hereda de Punto
2
3
4
5
6
7
8
9
10
11
12
13 };
```

Ejemplo de herencia – Clase Punto y Circulo

```
1  class Punto {
2     friend ostream &operator<<( ostream & , const Punto & ); // cout << p;
3
4    public:
5     Punto(int = 0, int = 0); // constructor predeterminado
6     void establecePunto(int , int ); // establece coordenadas
7     int obtienex() const { return x; } // obtiene la coordenada x
8     int obtieneY() const { return y; } // obtiene la coordenada y
9
10   protected: // accesible para las clase derivadas
11    int x, y; // las coordenadas x e y de Punto
12 };
```

```
1  class Circulo : public Punto { // Circulo hereda de Punto
2     friend ostream &operator<<( ostream & , const Circulo & ); // cout << c;
3
4    public:
5     // constructor predeterminado
6     Circulo(double r = 0.0, int x = 0, int y = 0);
7     void estableceRadio(double ); // establece el radio
8     double obtieneRadio() const; // devuelve el radio
9     double area() const; // calcula el área
10
11
12
13 };
```

Ejemplo de herencia – Clase Punto y Circulo

```
1  class Punto {
2     friend ostream &operator<<( ostream & , const Punto & ); // cout << p;
3
4   public:
5     Punto(int = 0, int = 0); // constructor predeterminado
6     void establecePunto(int , int ); // establece coordenadas
7     int obtienex() const { return x; } // obtiene la coordenada x
8     int obtieneY() const { return y; } // obtiene la coordenada y
9
10  protected: // accesible para las clase derivadas
11    int x, y; // las coordenadas x e y de Punto
12 };
```

```
1  class Circulo : public Punto { // Circulo hereda de Punto
2     friend ostream &operator<<( ostream & , const Circulo & ); // cout << c;
3
4   public:
5     // constructor predeterminado
6     Circulo(double r = 0.0, int x = 0, int y = 0);
7     void estableceRadio(double ); // establece el radio
8     double obtieneRadio() const; // devuelve el radio
9     double area() const; // calcula el área
10
11  protected:
12    double radio;
13 };
```

Ejemplo de herencia – Clase Punto y Circulo

```
1  #include <iostream>
2  using namespace std;
3
4  #include "punto.h"
5  #include "circulo.h"
6
7  int main() {
8    Punto p1(10, 20);
9    Circulo c1(2.5, 2, 2);
10
11   cout << "Punto␣p1:␣" << p1 << endl;
12   cout << "Circulo␣c1:␣" << c1 << endl;
13
14   c1.establecePunto(10, 10);
15   cout << "Circulo␣c1:␣" << c1 << endl;
16   cout << "Área:␣" << c1.area() << endl;
17
18   return 0;
19 }
```

Ejemplo de herencia – Clase Punto y Circulo

```
 1 #include <iostream>
 2 using namespace std;
 3 
 4 #include "punto.h"
 5 #include "circulo.h"
 6 
 7 int main() {
 8   Punto p1(10, 20);
 9   Circulo c1(2.5, 2, 2);
10 
11   cout << "Punto p1: " << p1 << endl;
12   cout << "Circulo c1: " << c1 << endl;
13 
14   c1.establecePunto(10, 10);
15   cout << "Circulo c1: " << c1 << endl;
16   cout << "Área: " << c1.area() << endl;
17 
18   return 0;
19 }
```

```
Punto p1: [10, 20]
Circulo c1: Centro = [2, 2]; Radio = 2.50
Circulo c1: Centro = [10, 10]; Radio = 2.50
Area: 19.63
```

Conversión de puntero: de clase base a derivada

- Un objeto de una clase derivada puede tratarse como un objeto de su clase base

Conversión de puntero: de clase base a derivada

- Un objeto de una clase derivada puede tratarse como un objeto de su clase base
- Un objeto de una clase base no siempre es un objeto de la clase derivada

Conversión de puntero: de clase base a derivada

- Un objeto de una clase derivada puede tratarse como un objeto de su clase base
- Un objeto de una clase base no siempre es un objeto de la clase derivada
- Se puede utilizar una conversión de tipo explícita para convertir un puntero de una clase base a uno de clase derivada

Conversión de puntero: de clase base a derivada

- Un objeto de una clase derivada puede tratarse como un objeto de su clase base
- Un objeto de una clase base no siempre es un objeto de la clase derivada
- Se puede utilizar una conversión de tipo explícita para convertir un puntero de una clase base a uno de clase derivada

Ver código fuente ejemplo `fig19_04.cpp` (D&D 4° ed) e implementación de las clases

Conversión de puntero: de clase base a derivada

- Un objeto de una clase derivada puede tratarse como un objeto de su clase base
- Un objeto de una clase base no siempre es un objeto de la clase derivada
- Se puede utilizar una conversión de tipo explícita para convertir un puntero de una clase base a uno de clase derivada

Ver código fuente ejemplo `fig19_04.cpp` (D&D 4° ed) e implementación de las clases

- `p` es un objeto `Punto`.

Conversión de puntero: de clase base a derivada

- Un objeto de una clase derivada puede tratarse como un objeto de su clase base
- Un objeto de una clase base no siempre es un objeto de la clase derivada
- Se puede utilizar una conversión de tipo explícita para convertir un puntero de una clase base a uno de clase derivada

Ver código fuente ejemplo `fig19_04.cpp` (D&D 4° ed) e implementación de las clases

- `p` es un objeto `Punto`. `ptrPunto` es un puntero a objeto `Punto`

Conversión de puntero: de clase base a derivada

- Un objeto de una clase derivada puede tratarse como un objeto de su clase base
- Un objeto de una clase base no siempre es un objeto de la clase derivada
- Se puede utilizar una conversión de tipo explícita para convertir un puntero de una clase base a uno de clase derivada

Ver código fuente ejemplo `fig19_04.cpp` (D&D 4° ed) e implementación de las clases

- `p` es un objeto `Punto`. `ptrPunto` es un puntero a objeto `Punto`
- `c` es un objeto `Circulo`.

Conversión de puntero: de clase base a derivada

- Un objeto de una clase derivada puede tratarse como un objeto de su clase base
- Un objeto de una clase base no siempre es un objeto de la clase derivada
- Se puede utilizar una conversión de tipo explícita para convertir un puntero de una clase base a uno de clase derivada

Ver código fuente ejemplo `fig19_04.cpp` (D&D 4° ed) e implementación de las clases

- `p` es un objeto `Punto`. `ptrPunto` es un puntero a objeto `Punto`
- `c` es un objeto `Circulo`. `ptrCirculo` es un puntero a objeto `Circulo`

Conversión de puntero: de clase base a derivada

- Un objeto de una clase derivada puede tratarse como un objeto de su clase base
- Un objeto de una clase base no siempre es un objeto de la clase derivada
- Se puede utilizar una conversión de tipo explícita para convertir un puntero de una clase base a uno de clase derivada

Ver código fuente ejemplo `fig19_04.cpp` (D&D 4° ed) e implementación de las clases

- `p` es un objeto `Punto`. `ptrPunto` es un puntero a objeto `Punto`
- `c` es un objeto `Circulo`. `ptrCirculo` es un puntero a objeto `Circulo`
- Con la herencia pública siempre es válido asignar un puntero de una clase derivada a un puntero de la clase base, debido a que un objeto de la clase derivada *es un* objeto de la clase base.

```
ptrPunto = &c // ptrPunto apunta al objeto c
```

Conversión de puntero: de clase base a derivada

- Un objeto de una clase derivada puede tratarse como un objeto de su clase base
- Un objeto de una clase base no siempre es un objeto de la clase derivada
- Se puede utilizar una conversión de tipo explícita para convertir un puntero de una clase base a uno de clase derivada

Ver código fuente ejemplo `fig19_04.cpp` (D&D 4° ed) e implementación de las clases

- `p` es un objeto `Punto`. `ptrPunto` es un puntero a objeto `Punto`
- `c` es un objeto `Circulo`. `ptrCirculo` es un puntero a objeto `Circulo`
- Con la herencia pública siempre es válido asignar un puntero de una clase derivada a un puntero de la clase base, debido a que un objeto de la clase derivada *es un* objeto de la clase base.

```
ptrPunto = &c // ptrPunto apunta al objeto c
```

- Conversión explícita de punteros

```
ptrCirculo = static_cast< Circulo * >(ptrPunto);
```

- Al mostrar un `Punto` como un `Circulo` resulta en un valor indefinido (en este caso es cero) para el `radio` (un objeto `Punto` no tiene un miembro `radio`)

Redefinición de funciones miembros

- Una clase derivada puede redefinir una función miembro de la clase base, generando así una nueva versión con la misma firma (si la firma fuera diferente sería una sobrecarga de función y no una redefinición)

Redefinición de funciones miembros

- Una clase derivada puede redefinir una función miembro de la clase base, generando así una nueva versión con la misma firma (si la firma fuera diferente sería una sobrecarga de función y no una redefinición)
- Cuando se llama a esta función en la clase derivada se invoca a la de dicha clase

Redefinición de funciones miembros

- Una clase derivada puede redefinir una función miembro de la clase base, generando así una nueva versión con la misma firma (si la firma fuera diferente sería una sobrecarga de función y no una redefinición)
- Cuando se llama a esta función en la clase derivada se invoca a la de dicha clase
- Se puede utilizar el operador de resolución de alcance (`::`) para tener acceso a la versión de la clase base

Redefinición de funciones miembros

- Una clase derivada puede redefinir una función miembro de la clase base, generando así una nueva versión con la misma firma (si la firma fuera diferente sería una sobrecarga de función y no una redefinición)
- Cuando se llama a esta función en la clase derivada se invoca a la de dicha clase
- Se puede utilizar el operador de resolución de alcance (`::`) para tener acceso a la versión de la clase base

Por ejemplo: suponer una clase **`Empleado`** tiene datos miembros **`nombre`** y **`apellido`**, de la cual se derivan:

1. **`EmpleadoXHora`**: obtiene el pago por hora

Redefinición de funciones miembros

- Una clase derivada puede redefinir una función miembro de la clase base, generando así una nueva versión con la misma firma (si la firma fuera diferente sería una sobrecarga de función y no una redefinición)
- Cuando se llama a esta función en la clase derivada se invoca a la de dicha clase
- Se puede utilizar el operador de resolución de alcance (`::`) para tener acceso a la versión de la clase base

Por ejemplo: suponer una clase **`Empleado`** tiene datos miembros **`nombre`** y **`apellido`**, de la cual se derivan:

1. **`EmpleadoXHora`**: obtiene el pago por hora
2. **`EmpleadoXPieza`**: obtiene el pago por pieza producida

Redefinición de funciones miembros

- Una clase derivada puede redefinir una función miembro de la clase base, generando así una nueva versión con la misma firma (si la firma fuera diferente sería una sobrecarga de función y no una redefinición)
- Cuando se llama a esta función en la clase derivada se invoca a la de dicha clase
- Se puede utilizar el operador de resolución de alcance (`::`) para tener acceso a la versión de la clase base

Por ejemplo: suponer una clase **`Empleado`** tiene datos miembros **`nombre`** y **`apellido`**, de la cual se derivan:

1. **`EmpleadoXHora`**: obtiene el pago por hora
2. **`EmpleadoXPieza`**: obtiene el pago por pieza producida
3. **`Jefe`**: obtiene un salario fijo por semana

Redefinición de funciones miembros

- Una clase derivada puede redefinir una función miembro de la clase base, generando así una nueva versión con la misma firma (si la firma fuera diferente sería una sobrecarga de función y no una redefinición)
- Cuando se llama a esta función en la clase derivada se invoca a la de dicha clase
- Se puede utilizar el operador de resolución de alcance (`::`) para tener acceso a la versión de la clase base

Por ejemplo: suponer una clase **`Empleado`** tiene datos miembros **`nombre`** y **`apellido`**, de la cual se derivan:

1. **`EmpleadoXHora`**: obtiene el pago por hora
2. **`EmpleadoXPieza`**: obtiene el pago por pieza producida
3. **`Jefe`**: obtiene un salario fijo por semana
4. **`EmpleadoXComision`**: salario fijo por semana más un porcentaje fijo de sus ventas por semana

Redefinición de funciones miembros

- Una clase derivada puede redefinir una función miembro de la clase base, generando así una nueva versión con la misma firma (si la firma fuera diferente sería una sobrecarga de función y no una redefinición)
- Cuando se llama a esta función en la clase derivada se invoca a la de dicha clase
- Se puede utilizar el operador de resolución de alcance (`::`) para tener acceso a la versión de la clase base

Por ejemplo: suponer una clase `Empleado` tiene datos miembros `nombre` y `apellido`, de la cual se derivan:

1. `EmpleadoXHora`: obtiene el pago por hora
2. `EmpleadoXPieza`: obtiene el pago por pieza producida
3. `Jefe`: obtiene un salario fijo por semana
4. `EmpleadoXComision`: salario fijo por semana más un porcentaje fijo de sus ventas por semana

Ver ejemplo de definición de la clase `Empleado` (`empleado.h`) y `EmpleadoXHora` (`porhora.h`). Código fuente `fig19_05.cpp` (D&D 4° ed.)
[redefine función miembro `imprime()`]

Ejemplo: `Punto`, `Circulo` y `Cilindro`

- Ver `punto3.h`, `punto3.cpp` y `fig19_08.cpp`
- Ver `circulo3.h`, `circulo3.cpp` y `fig19_09.cpp`
- Ver `cilindro3.h`, `cilindro3.cpp` y `fig19_10.cpp`