Herencia y Polimorfismo

Werner Ezequiel Maximiliano
eze210 <at> gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

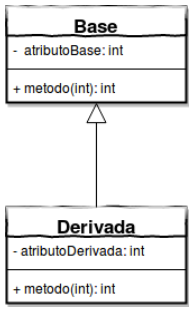
Contenidos

Subsection 1

¿Qué es la Herencia?

Herencia

1. Es un concepto de la OOP
2. Establece una relación **"es un"**
3. Se dice que:
   - 3.1 Una clase *derivada* **extiende** a una clase *base*.
   - 3.2 Una clase *base* **generaliza** varias clases *derivadas*.



Subsection 2

Herencia en C++

Herencia en C++

En C++ heredamos indicando un tipo de herencia (usaremos public por default)

```
1  class Base {
2    ...
3  };
4
5  class Derivada: public Base {
6    ...
7  };
```

1. No se heredan:
   - 1.1 Los constructores
   - 1.2 Los operadores
   - 1.3 friend
2. También se puede heredar en forma protected o private.

¿Cómo llamamos al constructor de Base?

¡No podemos acceder a los miembros private de la clase base!

```
1  class Base {
2  private:
3    int entero;
4
5  public:
6    Base(int entero): entero(entero) {
7    }
8  };
9
10 class Derivada: public Base {
11 public:
12   Derivada(int entero): Base(entero) {
13   }
14 };
```

1. Para delegar el constructor usamos la **member initializer list**.
2. En C++ el constructor de Base se llama antes que el constructor de Derivada.
3. Con los destructores es al revés (como una pila).

Subsection 1

¿Qué es el Polimorfismo?

Polimorfismo

1. Tener varias formas.
2. En C++, significa que la misma llamada a función tiene distintos comportamientos dependiendo del tipo del **objeto**.
3. No queremos que dependa del tipo de la **variable**.

Subsection 2

Polimorfismo en C++

Polimorfismo en C++

¿Cuál es la salida de este código?

```
1  #include <iostream>
2
3  class Base {
4  public:
5    void metodo() {
6      std::cout << "Base\n";
7    }
8  };
9
10 class Derivada1: public Base {
11 public:
12   void metodo() {
13     std::cout << "Derivada_1\n";
14   }
15 };
16
17 class Derivada2: public Base {
18 public:
19   void metodo() {
20     std::cout << "Derivada_2\n";
21   }
```

```
25 int main(void) {
26   Base *ptr;
27   Derivada1 d1;
28   Derivada2 d2;
29
30   ptr = &d1;
31   ptr->metodo();
32
33   ptr = &d2;
34   ptr->metodo();
35
36   return 0;
37 }
```

La salida será:

```
Base
Base
```

- Por defecto, C++ resuelve a qué método llamar en tiempo de compilación.
- A esto se lo conoce como **static linkage, static resolution, o early binding**.
- Para que esto se resuelva en tiempo de ejecución debemos usar el modificador **virtual**.
- En Java todo es virtual por defecto, y para que no sea virtual hay que usar **final**.
- Usar virtual crea **VTable**s, que sirven para la resolución dinámica, lo cual empeora la performance.
- A esto se lo conoce como **dynamic linkage, o late binding**.

Polimorfismo en C++

Usando el modificador **virtual**.

```cpp
#include <iostream>

class Base {
public:
  virtual void metodo() {
    std::cout << "Base\n";
  }
};

class Derivada1: public Base {
public:
  virtual void metodo() {
    std::cout << "Derivada 1\n";
  }
};

class Derivada2: public Base {
public:
  virtual void metodo() override {
    std::cout << "Derivada 2\n";
  }
```

```cpp
int main(void) {
  Base *ptr;
  Derivada1 d1;
  Derivada2 d2;

  ptr = &d1;
  ptr->metodo();

  ptr = &d2;
  ptr->metodo();

  return 0;
}
```

La salida será:

```
Derivada 1
Derivada 2
```

- Notar que la variable **ptr** siempre es de tipo **Base***, lo que cambia es el tipo del **objeto**.
- Esto nos permite, por ejemplo, tener una lista de punteros a **Base**, e iterarla llamando siempre a **metodo()**, sin depender del tipo del objeto en tiempo de compilación.
- El identificador **override** (C++11) obliga a que la firma coincida con la del método de la clase base.

¿Cómo llamamos un método de Base desde Derivada?

```cpp
#include <iostream>

class Base {
public:
  virtual void metodo() {
    std::cout << "Base\n";
  }
};

class Derivada: public Base {
public:
  virtual void metodo() {
    std::cout << "Derivada\n";
    Base::metodo();
  }
};
```

```cpp
int main(void) {
  Derivada d;
  d.metodo();

  return 0;
}
```

La salida será:

```
Derivada
Base
```

Subsection 3

Clases abstractas e Interfaces

Clases abstractas

Las clases abstractas son las que tienen un **método virtual puro**.

```cpp
#include <iostream>

class Base {
public:
  virtual void metodo() = 0;
};

class Derivada1: public Base {
public:
  virtual void metodo() {
    std::cout << "Derivada 1\n";
  }
};

class Derivada2: public Base {
public:
  virtual void metodo() {
    std::cout << "Derivada 2\n";
  }
};
```

```cpp
int main(void) {
  // no compila!
  Base base;
  Derivada1 d1;
  Derivada2 d2;

  return 0;
}
```

- No se puede instanciar una clase abstracta.
- En C++ no existe un modificador *abstract*, una clase abstracta es la que tiene un método virtual puro.
- La única manera de hacer interfaces en C++ es usando clases abstractas sin atributos.

Subsection 4

Destructores virtuales

Destructores virtuales

```
1  class Base {
2  public:
3    Base() {
4      std::cout << "Base\n";
5    }
6
7    ~Base() {
8      std::cout << "Destr Base\n";
9    }
10 };
11
12 class Derivada: public Base {
13 public:
14   Derivada() {
15     std::cout << "Derivada\n";
16   }
17
18   ~Derivada() {
19     std::cout << "Destr Derivada\n";
20   }
21 };
```

```
23 int main(void) {
24   Base *b = new Derivada();
25   delete b;
26   return 0;
27 }
```

La salida será:

```
Derivada
Base
Destr Base
```

- En la línea 25 se llama al destructor de Base, pero no al de Derivada (porque b es de tipo Base* para el compilador!).
- Para evitar esto, el destructor de Base debe ser virtual (de esta manera demoramos la decisión de qué método llamar hasta la ejecución).

Destructores virtuales

```
1  class Base {
2  public:
3    Base() {
4      std::cout << "Base\n";
5    }
6
7    virtual ~Base() {
8      std::cout << "Destr Base\n";
9    }
10 };
11
12 class Derivada: public Base {
13 public:
14   Derivada() {
15     std::cout << "Derivada\n";
16   }
17
18   ~Derivada() {
19     std::cout << "Destr Derivada\n";
20   }
21 };
```

```
23 int main(void) {
24   Base *b = new Derivada();
25   delete b;
26   return 0;
27 }
```

- Ahora el compilador no decide a qué método llamar sino que se decide en tiempo de ejecución.
- Entonces, como el objeto es de tipo Derivada (aunque la variable sea un Base*) se llamará a:
  1. Constructor de Base
  2. Constructor de Derivada
  3. Destructor de Derivada
  4. Destructor de Base

Subsection 5

Métodos de Clase

Métodos de Clase

Los métodos de Clase se marcan con el modificador **static**.

```
1  class Clase {
2  public:
3    static void metodoDeClase() {
4      std::cout << "Metodo de clase" <<
5        std::endl;
6    }
7
8    void metodoDeInstancia() {
9      std::cout << "Metodo de instancia" <<
10       std::endl;
11   }
12 };

14 int main(void) {
15   Clase::metodoDeClase();
16
17   Clase c;
18   c.metodoDeInstancia();
19
20   return 0;
21 }
```

Subsection 1

Object Slicing

Object Slicing

```
1  class Base {
2  protected:
3    int a;
4  public:
5    Base(int a) : a(a) {}
6    virtual void print() {
7      std::cout << "Base, a=" <<
8        a << std::endl;
9    }
10 };

26 int main(void) {
27   Derivada d(1, 2);
28   d.print();
29   Base b(3);
30   b.print();
31   b = d;
32   b.print();
33
34   return 0;
35 }
```

```
12 class Derivada: public Base {
13 private:
14   int b;
15 public:
16   Derivada(int a, int b) :
17     Base(a),
18     b(b) {}
19
20   void print() {
21     std::cout << "Derivada, a=" <<
22       a << ", b=" << b << std::endl
23   }
24 };
```

La salida será:

```
Derivada, a=1, b=2
Base, a=3
Base, a=1
```

Object Slicing - Pasaje por valor

El object slicing también se da cuando pasamos objetos por valor.

```
26 void print(Base obj) {
27   obj.print();
28 }
29
30 int main(void) {
31   Derivada d(1, 2);
32   print(d); // pasaje por valor
33   return 0;
34 }
```

La salida será:

```
Base, a=1
```

Object Slicing - Pasaje por referencia

Pasar por referencia (o por puntero) es una solución al object slicing.

```
26 void print(Base &obj) {
27   obj.print();
28 }
29
30 void printPtr(Base *obj) {
31   obj->print();
32 }
33
34 int main(void) {
35   Derivada d(1, 2);
36   print(d); // pasaje por referencia
37   printPtr(&d); // pasaje por puntero
38   return 0;
39 }
```

La salida será:

```
Derivada, a=1, b=2
Derivada, a=1, b=2
```

Object Slicing - Veamos qué pasa con este código

```
1  class A {
2      int a;
3  public:
4      A(int a) : a(a) {}
5      void print(){
6          std::cout << "A. a:" <<
7            a << std::endl;
8      }
9      int getA(){
10         return a;
11     }
12 };
13
14 class B : public A {
15     int b;
16 public:
17     B(int a, int b): A(a), b(b) {}
18     void print(){
19         std::cout << "B. a:" <<
20           getA() << ". b:" <<
21           b << std::endl;
22     }
```

```
24 int main() {
25     A var1(1);
26     var1.print();
27
28     B var2(2, 3);
29     var2.print();
30
31     A& var3 = var2;
32     var3.print();
33
34     var3 = var1;
35     var3.print();
36     var2.print();
37     return 0;
38 }
```

- En la línea 26, var1 es de tipo A, entonces se mostrará el mensaje "A. a:1".
- Luego se instancia var2, de tipo B, y en la línea 29 se mostrará el mensaje "B. a:2. b:3".
- En la línea 31 vemos que var3 es una referencia a A, y como print no es un método virtual, en la línea 32 se va a tratar a var2 como un A, y se va a imprimir el mensaje "A. a:2".
- Otro punto importante a tener en cuenta en la línea 31 es que a partir de esa inicialización de var3, var3 será un alias de var2, y siempre que usemos var3, vamos a estar haciendo referencia a var2.
- En la asignación de la línea 34, estamos pisando el valor del atributo a de var3 (¡y también de var2!) con el valor del atributo a de var1.
- En la línea 35 vamos a imprimir var3 (var2 tratada como un A), y entonces se mostrará el mensaje "A. a:1".
- Pero en la línea 36, tenemos a var2 (ahora tratada como un B), y entonces se mostrará el mensaje "B. a:1. b:3". ¡Porque en la 34 pisamos el a de var2!
- Como ejercicio, ver qué cambiaría si print fuese declarado como virtual en A.

Subsection 2

Herencia múltiple - Problema del Diamante

Herencia múltiple

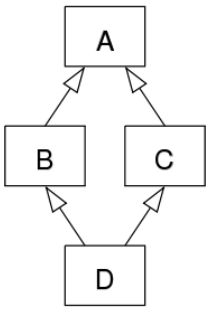
C++ permite herencia múltiple.

```
1  class Base1 {
2    ...
3  };
4
5  class Base2 {
6    ...
7  };
8
9  class Derivada: public Base1, public Base2 {
10   ...
11 };
```

1. Para usar herencia múltiple debemos indicar de qué clases queremos heredar, separándolas con comas.
2. No se recomienda, pero para implementar varias interfaces no hay otra alternativa.

Problema del diamante

La herencia múltiple nos permite codificar un esquema como el siguiente



Problema del diamante

Qué pasa con el siguiente código?

```
1  class A {
2    int getEntero();
3  };
4
5  class B: public A {};
6  class C: public A {};
7  class D: public B, public C {};
8
9  int main(void) {
10   D d;
11   d.getEntero();
12   return 0;
13 }
```

La línea 11 no compila porque tanto B como C tienen su copia interna de los miembros de A, y por ende la llamada a **getEntero()** es ambigua.

Problema del diamante - Solución

Hacer que B y C hereden virtualmente de A.

```
1  class A {
2    int getEntero();
3  };
4
5  class B: virtual public A {};
6  class C: virtual public A {};
7  class D: public B, public C {};
8
9  int main(void) {
10   D d;
11   d.getEntero();
12   return 0;
13 }
```

Ahora D tendrá una sola copia interna de A.

Subsection 1

Referencias

Referencias I


Bjarne Stroustrup.
The C++ Programming Language.
Addison Wesley, Fourth Edition.