Herencia y Polimorfismo

Taller de Programación I - FIUBA

class y **struct**

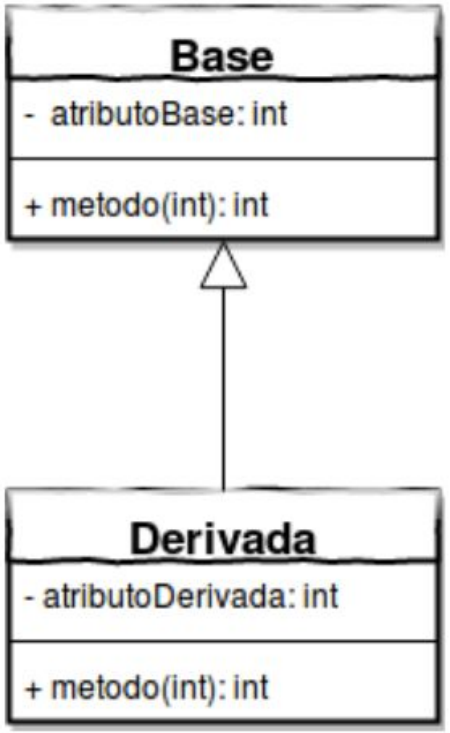
```
class Clase1 {

 ....

};

struct Clase2 {

  ....

};
```

En C++ **class** y **struct** son equivalentes. La única diferencia es que por default los atributos y métodos son *privados* en una clase (**class**) mientras que un una estructura (**struct**) son *públicos*.

Se recomienda usar siempre **class** pero para ahorrar espacio en estas diapositivas se usarán mayoritariamente **struct**.

Herencia

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



Herencia en C++

```
class Base {

 ....

};

class Derivada : public Base {

  ....

};
```

- La herencia **pública** es la más usada.
- No se heredan:
  - Los constructores
  - Los operadores
  - friend
- También se puede heredar en forma **protected** o **private**

Tipos de Herencia en C++

	Tipo de herencia de la clase Derivada		
Método de la clase Base	**Public**	**Protected**	**Private**
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	No accesible	No accesible	No accesible

Cómo llamar al constructor de la clase Base?

```
class Base {
  int entero;
public:
   Base(int entero): entero(entero) { }
};

class Derivada : public Base {
public:
   Derivada(int entero): Base(entero) { }
};
```

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

Cómo llamar a un método de la clase Base?

```
struct Base {
  void hola() {
     std::cout << “Hola base\n”;
   }
};

struct Derivada : public Base {
  void hola() {
     Base::hola();
     std::cout << “Hola derivada\n”;
   }
};
```

```
int main() {
    Derivada d;
    d.hola();   // Imprime “Hola base” y
                // luego imprime “Hola derivada”

}
```

Static linkage (early binding)

```
struct Base {
  void hola() {
    std::cout << “Hola base\n”;
  }
};

struct Derivada : public Base {
  void hola() {
    std::cout << “Hola derivada\n”;
  }
};
```

```
int main() {
    Derivada d;
    d.hola();   // Imprime “Hola derivada”

    Base *ptr = &d;  // Relacion “es un”
    ptr->hola();  // Imprime “Hola base”
}
```

- Por defecto, C++ resuelve a qué método llamar **en tiempo de compilación según la variable** (**ptr** es de tipo **Base)**

Polimorfismo (dynamic linkage o late binding)

```
struct Base {
  virtual void hola() {
     std::cout << “Hola base\n”;
   }
};

struct Derivada : public Base {
  virtual void hola() {
     std::cout << “Hola derivada\n”;
   }
};
```

```
int main() {
    Derivada d;
    d.hola();   // Imprime “Hola derivada”

    Base *ptr = &d;  // Relacion “es un”
    ptr->hola();  // Imprime “Hola derivada”
}
```

- Con **virtual**, C++ resuelve a qué método llamar **en tiempo de ejecución según el objeto en memoria** (**ptr apunta** a un tipo **Derivada**)

Destructores virtuales (bug)

```
struct Base {
  Base() { std::cout << “init B\n”;  }
  ~Base() { std::cout << “destr B\n”; }
};

struct Derivada : public Base {
  Derivada() { std::cout << “init D\n”;  }
  ~Derivada() { std::cout << “destr D\n”; }
};
```

```
int main() {
  // Imprime “init B” y luego
  // imprime “init D”
  Base *ptr = new Derivada();

  // Imprime solo “destr B”
  // posiblemente dejando leaks
  delete ptr;
}
```

Destructores virtuales (ok)

```
struct Base {
  Base() { std::cout << “init B\n”;  }
  virtual ~Base() { std::cout << “destr B\n”; }
};

struct Derivada : public Base {
  Derivada() { std::cout << “init D\n”;  }
  virtual ~Derivada() { std::cout << “destr D\n”; }
};
```

```
int main() {
   // Imprime “init B” y luego
   // imprime “init D”
   Base *ptr = new Derivada();

   // Imprime “destr D” y luego
   // imprime “destr B”
   delete ptr;
}
```

Clases abstractas (virtual puro)

```
struct Base {
  virtual void hola() = 0;
};

struct Derivada : public Base {
  virtual void hola() { std::cout << “hola\n”; }
};
```

```
int main() {
   // Esto no compila, no se puede
   // instanciar Base por que tiene
   // un método virtual puro (la clase es
   // abstracta)
   Base b;

   // Derivada sí se puede instanciar
   Derivada d;
}
```

Métodos de clase (métodos **static**)

```
struct Clase {
  static void hola() { std::cout << “hola\n”; }
};
```

```
int main() {
   Clase c;

   // Imprime “hola”
   c.hola();

   // También imprime “hola”
   Clase::hola();
}
```

Object slicing

```
struct Base {
  int a;
  Base(int a) : a(a) {}
  void print() { cout << “Base “ << a; }
};
struct Derivada : public Base {
  int b;
  Derivada(int a, int b) : Base(a), b(b) {}
  void print() { cout << “Derivada “ << a << b; }
};
```

```
int main() {
   Base b(1);
   b.print();  // Imprime “Base 1”

   Derivada d(2, 3);
   d.print();   // Imprime “Derivada 2 3”

   b = d;  // object sliced!
   b.print();  // Imprime “Base 2”
}
```

Herencia múltiple

```
struct Base1 {
};

struct Base2 {
};

struct Derivada : public Base1, public Base2 {
};
```