



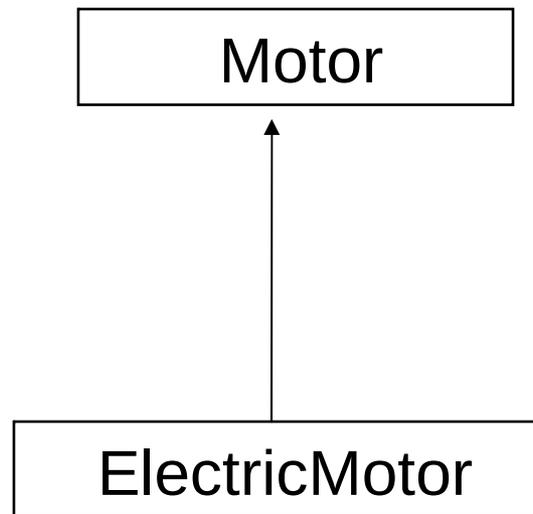
UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Herencia

Agustín J. González
ELO329

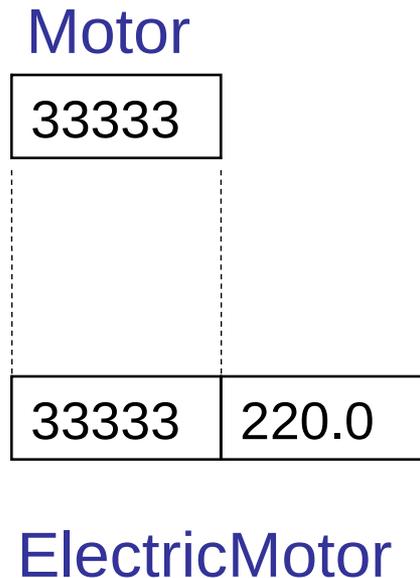
Motor y ElectricMotor

- Consideremos dos clases que tienen algo en común.



Motor y ElectricMotor

- Un objeto ElectricMotor contiene el mismo número de identificación (ID) como un Motor, más el voltaje.



Clase CMotor

- Definición de la clase CMotor:

```
class CMotor {  
public:  
    CMotor() { }  
    CMotor( const string & id );  
    string get_ID() const;  
    void set_ID(const string & s);  
    void Display() const;  
    void Input();  
  
private:  
    string m_sID;  
}; // más...
```

```

CMotor::CMotor( const string & id )
{ set_ID(id); }

string CMotor::get_ID() const
{ return m_sID; }

void CMotor::set_ID(const string & s)
{ m_sID = s; }

void CMotor::Display() const
{   cout << "[CMotor] ID=" << m_sID; }

void CMotor::Input()
{ string temp;
  cout << "Enter Motor ID: ";
  cin >> temp;
  set_ID(temp);
}

```

Diseño y Programación Orientados a Objetos

Creación de Clases Derivadas

```
class base {  
...  
};
```

La clase base debe aparecer primero en las declaraciones.

```
class derivada : public base {  
...  
};
```

Un clase puede derivar de más de una clase base
(ojo es una diferencia con JAVA)

Clase CElectricMotor

```
class CElectricMotor : public CMotor {  
public:  
    CElectricMotor();  
    CElectricMotor(const string & id, double volts);  
  
    void Display() const;  
    void Input();  
    void set_Voltage(double volts);  
    double get_Voltage() const;  
  
private:  
    double m_nVoltage;  
};
```

Inicializador de Clase Base

- Un inicializador de clase base llama al constructor de la clase base. En este ejemplo, el número ID del motor es pasado al constructor de Cmotor.

```
CElectricMotor::CElectricMotor(const string & id, double  
    volts) : CMotor(id)  
{  
    m_nVoltage = volts;  
}
```

Llamando a métodos de la clase base

```
void CElectricMotor::Input()
{
    CMotor::Input(); // llamamos a la clase base primero
                    // En java lo hacíamos con super.Input()

    double volts;
    cout << "Voltage: ";
    cin >> volts;
    set_Voltage(volts);
}
```

La función `Input` existe en ambas clases `CMotor` y `CElectricMotor`. En lugar de duplicar el código ya escrito, se llama al método correspondiente en la clase base:

Función o método de Despliegue

- La función Display funciona de la misma manera. Ésta llama a CMotor::Display primero.

```
void CElectricMotor::Display() const
{
    // call base class function first
    CMotor::Display();

    cout << " [CElectricMotor]"
         << " Voltage=" << m_nVoltage << endl;
}
```

Probando Clases

- Cuando el mismo nombre de método existe en ambas clases, **C++ llama al método implementado para la clase según la declaración del objeto**. Éste es el caso con los métodos Input y Display:

```
CElectricMotor elec;    // CelectricMotor
```

```
elec.Input();           // CElectricMotor
```

```
elec.Display();        // CElectricMotor
```

Asignación de Objetos

- Podemos asignar objetos de clases derivadas a un objeto de la clase base. Igual que en Java

```
CMotor mot;
```

```
CElectricMotor elec;
```

```
mot = elec; // se copian sólo los atributos de Motor
```

```
elec.get_Voltage(); // ok
```

```
mot.get_Voltage(); // error, no es motor eléctrico
```

Asignación de Objetos

- Pero no podemos asignar una instancia de una clase base a una instancia de una clase derivada (igual que en Java). Ello permitiría referencias a miembros no existentes.

```
CMotor mot;
```

```
CElectricMotor elec;
```

```
elec = mot;           // error
```

```
elec.set_Voltage( 220 ); // ???
```

Acceso a miembros Protected (Protegidos)

- Miembros de una clase designados como protected son visibles a ambos la clase actual y las clases derivadas. (y a clases amigas -friend- pero a nadie más). **Son análogos a Java.**

```
class CMotor {  
public:  
    CMotor() { }  
    CMotor( const string & id );  
protected:  
    string get_ID() const;  
    void set_ID(const string & s);  
//...  
}
```

Herencia Protegida

- Supongamos por un momento que CMotor usa miembros públicos para todos sus métodos:

```
class CMotor {  
public:  
    CMotor() { }  
    CMotor( const string & id );  
    string get_ID() const;  
    void set_ID(const string & s);  
  
    //...  
}
```

Herencia Protegida

- Podemos usar el calificador `protected` cuando creamos una clase derivada.
- **Todos los métodos públicos en la clase base pasan a ser protegidos en la clase derivada. Los restantes mantienen su nivel de restricción.**

```
class CElectricMotor : protected CMotor {
```

```
//...
```

```
};
```

Herencia Protegida

- Por ejemplo, el programa principal no puede llamar `set_ID` y `get_ID` en un motor eléctrico porque los métodos no son públicos en esta clase:

```
CElectricMotor EM;
```

```
EM.set_ID("12345");    // error
```

```
EM.get_ID();          // error
```

Herencia Privada

- La herencia privada causa que todos los métodos declarados en la clase base sean privados en la clase derivada.
- Pareciera que no hay diferencia con herencia protegida: Funciones en CElectricMotor pueden acceder funciones miembros en CMotor...

```
class CElectricMotor : private CMotor {
```

```
    //...
```

```
};
```

Herencia Privada

- **Pero** cuando derivamos una nueva clase (CPumpMotor) de CElectricMotor, la diferencia se hace notar: métodos en CPumpMotor no pueden acceder miembros públicos de Cmotor.

```
class CPumpMotor : public CElectricMotor {  
public:  
    void Display() {  
        CMotor::Display();    // not accessible!  
        CElectricMotor::Display(); // this is OK  
    }  
};
```

Miembros que no son heredados

- El constructor y destructor de la clase base no son heredados, deben ser definidos en la clase heredada.
- El constructor por omisión y el destructor sí son llamados cuando se crea o destruye una instancia de la clase derivada.
- Si la clase base tiene sobrecargado el operador =, éste no se hereda en la clase derivada.
- Las clases y funciones friend no son heredadas.