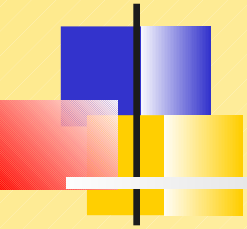
UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA



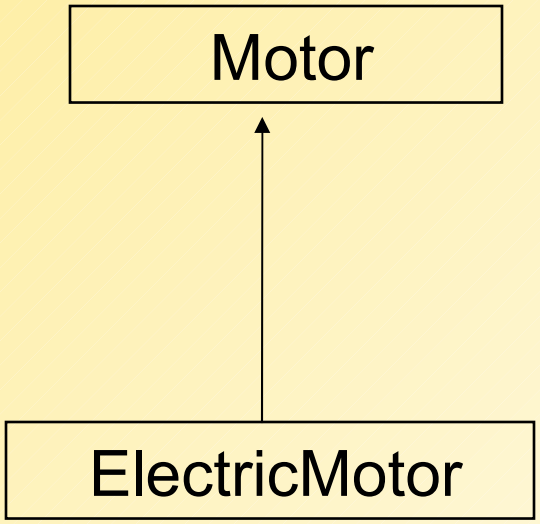
Herencia

Agustín J. González

ELO329

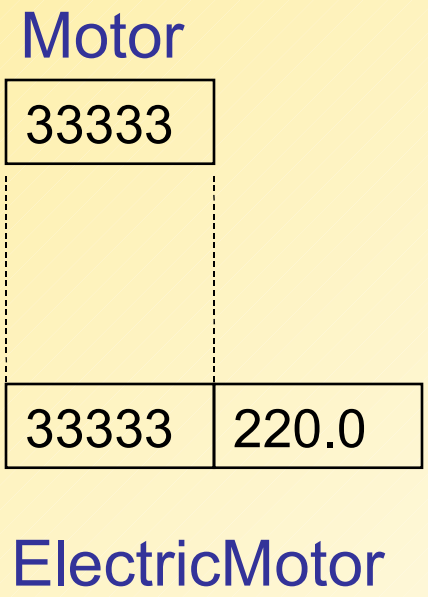
Motor y ElectricMotor

- 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.

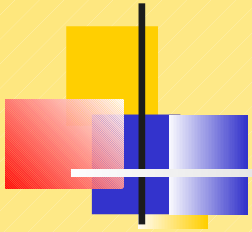


Clase CMotor

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);
}
```

Creación de Clases Derivadas

La clase base debe aparecer primero en las declaraciones. Cuando la clase derivada es declarada, ésta menciona a la clase base.

```
class base {
...
};

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

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:

```
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);
}
```

Llamado de Métodos de la clase Base

Esta es la función Input en la clase CMotor:

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

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 las Clases

Un programa de prueba puede crear instancias de ambas clases CMotor y CElectricMotor.

```
CMotor mot("33333");
mot.Display();
cout << endl;

CElectricMotor elec("40000",220.0);
elec.Display();
cout << endl;
```

Probando Clases

Cuando usamos instancias de clases derivadas de otra, podemos llamar a funciones de la clase base y la derivada. Igual que en Java.

```
CElectricMotor elec;        // CElectricMotor

elec.set_ID("40000");       // CMotor

elec.set_Voltage(220.0);    // CElectricMotor
```

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)

Clases con miembros Protected

Aquí hay un ejemplo que usa el calificador protected para limitar la visibilidad de get_ID y set_ID:

```
class CMotor {
public:
  CMotor() { }
  CMotor( const string & id );

protected:
  string get_ID() const;
  void set_ID(const string & s);

//...
```

Acceso a miembros Protected

El programa principal no puede llamar set_ID y get_ID porque ellos están protegidos:

```
CMotor M;

M.set_ID("12345");      // error

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

Acceso a miembros Protected

Pero funciones en CElectricMotor sí pueden acceder set_ID:

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

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 creemos una clase derivada.

Todos los métodos públicos en la clase base pasan a ser protegidos en la clase derivada.

```
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 Protegida

Puede ser que el autor de la clase CElectricMotor no quiera dar a conocer el número ID del motor.

Métodos en CElectricMotor sí pueden acceder métodos públicos en CMotor que serán privadas para otros. Un ejemplo:

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

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
  }


};
```