



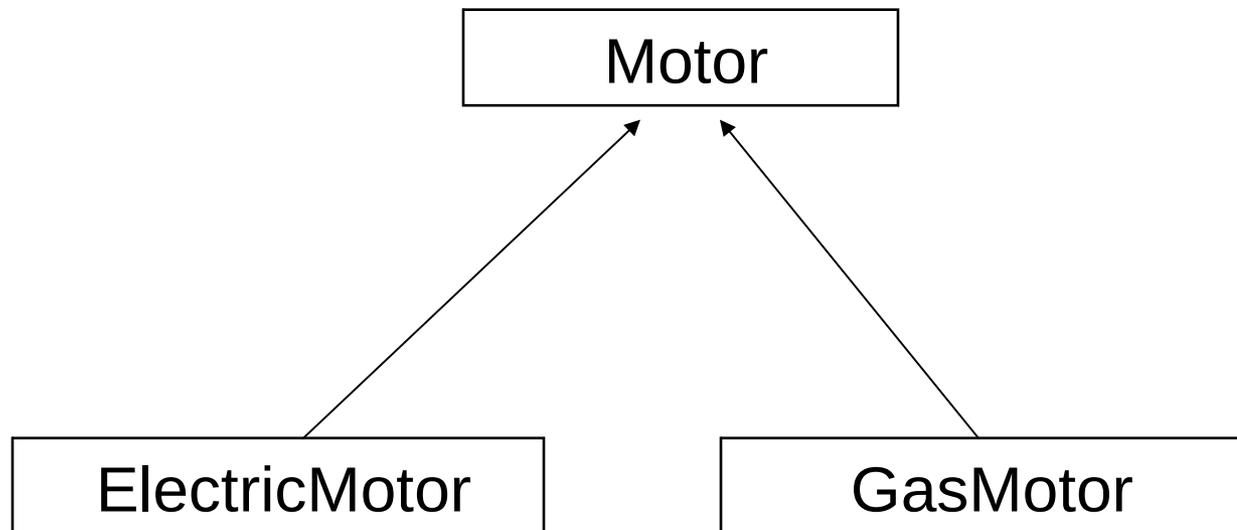
UNIVERSIDAD TÉCNICA  
FEDERICO SANTA MARÍA

# Polimorfismo en C++, ligado dinámico y Métodos Virtuales

Agustín J. González  
ELO329

# Jerarquía de clases Motor

- Recordemos la jerarquía de clases establecida para el estudio sobre Herencia:



# Clase CMotor

- La 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;  
};
```

# 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;  
};
```

```
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;  
};
```

# Clase CGasMotor

```
class CGasMotor :public CMotor {
public:
    CGasMotor();
    CGasMotor(const string & id, int cylinders);

    void Display() const;
    void Input();

private:
    int m_nCylinders;
};
```

```
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;
};
```

# Punteros a objetos de clases derivadas y referencias a objetos derivados

- Es fácil definir objetos dinámicos de una clase derivada usando un puntero de tipo específico:

```
CElectricMotor * pC = new CElectricMotor;
```

```
pC->set_ID("3099999");
```

```
pC->set_Voltage(110.5);
```

```
pC->Display();
```

```
:
```

```
delete pC;
```

No hay novedades en este código.

¿Qué pasa en C++ cuando el puntero es definido como:  
CMotor \*pC; ?

# Polimorfismo

- Podemos declarar punteros a una clase base, y luego asignarle la dirección de una instancia de una clase derivada. **Este caso es normal en Java**. Es el principio de sustitución en C++. Esta técnica es un tipo de polimorfismo.
- **Recordar: Polimorfismo es un concepto donde un mismo nombre puede referirse a objetos de clases diferentes que están relacionadas por una clase base común.**

```
CMotor * pM;
```

```
pM = new CElectricMotor; // puntero a motor eléctrico  
// semántica similar a Java
```

```
CElectricMotor em;
```

```
CMotor & motor = em; // referencia a motor eléctrico  
// esta opción no existe en Java
```

# Ligado dinámico

- En C++ la opción por **omisión es llamar el método definido por el tipo del puntero o referencia**, no el tipo del objeto apuntado. **Distinto a Java!**

```
CMotor * pM;           // base pointer type
```

```
pM = new CElectricMotor;
```

```
pM->Input();           // llama a CMotor::Input()
```

```
pM->Display();         // llama a CMotor::Display()
```

```
    // esta es una gran diferencia con Java. En Java
```

```
    // el ligado dinámico es la opción por omisión
```

```
// más...
```

# Métodos Virtuales (Virtual)

- Si deseamos tener un comportamiento como el de Java debemos declarar los métodos Input y Display como **virtuales** en la clase base.
- El calificador virtual le dice al compilador que genere código que mire al tipo del objeto apuntado (no del puntero) en tiempo de ejecución y use esta información para seleccionar la versión apropiada del método.
- El ligado dinámico aplica cuando usamos punteros o referencias a objetos.

```
class CMotor {  
    ...  
    virtual void Display() const; // el calificador virtual cambia  
    virtual void Input();        // la forma de definir el código  
                                // a ser finalmente invocado.  
    ...  
};
```

# Métodos Virtuales (cont.)

- Es recomendable definir también como virtuales los métodos en la clase derivada, en este caso en las clases CGasMotor y CElectricMotor.

```
class CGasMotor :public CMotor {  
public:  
  
...  
    virtual void Display() const;  
    virtual void Input();  
  
...  
};
```

- De esta forma la semántica del método se mantiene entre clases heredadas.

# Métodos Virtuales. Ejemplo

- Ahora los métodos Display e Input son llamados usando **ligado dinámico** desde la clase CElectricMotor:

```
CMotor * pM;
```

```
pM = new CElectricMotor;
```

```
pM->Input(); // CElectricMotor::Input()
```

```
pM->Display(); // CElectricMotor::Display()
```

# Métodos Virtuales

- A menudo, un puntero será pasado como argumento a una función que espera un puntero a objeto de la clase base.
- Cuando el método es llamado, podemos pasar cualquier puntero como parámetro actual, siempre y cuando éste apunte a una instancia derivada de la clase base (“subtipo”).

```
void GetAndShowMotor(CMotor * pC )
{
    pC->Input();
    cout << "Here's what you entered:\n";
    pC->Display();
    cout << "\n\n";
}
```

```
void GetAndShowMotor(CMotor & m )
{
    m.Input();
    cout << "Here's what you entered:\n";
    m.Display();
    cout << "\n\n";
}
```

Lo mismo vale para referencias

# Métodos Virtuales

- Ejemplo de llamados a GetAndShowMotor con diferentes tipos de punteros.

```
CGasMotor * pG = new CGasMotor;  
GetAndShowMotor( pG );
```

```
CElectricMotor * pE = new CElectricMotor;  
GetAndShowMotor( pE );
```

```
CMotor * pM = new CGasMotor;  
GetAndShowMotor( pM );
```

```
// la salida sería como ...
```

```
CGasMotor gm;  
GetAndShowMotor(gm);
```

```
CElectricMotor em;  
GetAndShowMotor(em);
```

```
CMotor m;  
GetAndShowMotor(m);
```

```
// la salida sería como ...
```

# (Salida de la lámina previa)

**[GasMotor]: Enter the Motor ID: 234323**

**Enter the number of cylinders: 3**

**Here's what you entered:**

**[GasMotor] ID=234323, Cylinders=3**

**[ElectricMotor]: Enter the Motor ID: 234324**

**Voltage: 220**

**Here's what you entered:**

**[ElectricMotor] ID=234324, Voltage=220**

**[GasMotor]: Enter the Motor ID: 44444**

**Enter the number of cylinders: 5**

**Here's what you entered:**

**[GasMotor] ID=44444, Cylinders=5**

# Creación de un vector de Motores

- Un vector de punteros CMotor puede contener punteros a cualquiera tipo de objeto derivado de CMotor.

```
vector<CMotor*> vMotors;
```

```
CMotor * pMotor;
```

```
pMotor = new CElectricMotor("10000",110);
```

```
vMotors.push_back(pMotor);
```

```
pMotor = new CGasMotor("20000",4);
```

```
vMotors.push_back(pMotor);
```

```
pMotor = new CElectricMotor("30000",220);
```

```
vMotors.push_back(pMotor);
```

```
pMotor = new CGasMotor("40000",2);
```

```
vMotors.push_back(pMotor);
```

# Acceso a Vector de punteros

- La función que despliega tales vectores no necesita saber exactamente qué tipo de puntero están en el vector mientras se llame a métodos virtuales.

```
void ShowVector( const vector<CMotor*> & vMotors ) {  
    cout << "---- Vector of Motor Pointers ----\n";  
    for(int i=0; i < vMotors.size(); i++) {  
        cout << (i+1) << ": ";  
        vMotors[i]->Display();    // virtual  
    }  
}
```

# Salida de la función ShowVector

- La función ShowVector llama a la versión apropiada del método virtual Display() para cada puntero en el vector.

----- Vector of Motor Pointers -----

1: [ElectricMotor] ID=10000, Voltage=110

2: [GasMotor] ID=20000, Cylinders=4

3: [ElectricMotor] ID=30000, Voltage=220

4: [GasMotor] ID=40000, Cylinders=2

# Liberación de almacenamiento

- Debemos liberar el almacenamiento usado por cada objeto motor. Este bucle remueve los punteros uno por uno.

```
for(int i=0; i < vMotors.size(); i++) {  
    delete vMotors[i]; // delete each motor  
}
```

- El operador delete accede a información que le permite saber exactamente cuánto almacenamiento liberar por cada puntero (aún cuando los motores ocupan distintos tamaños).
- Saber distinguir de:  
 CMotor \* motor = new CMotor [40];  
 delete [] motor; // aquí el arreglo está en el heap.

# Métodos Virtuales Puros

- Un método virtual puro no tiene implementación. Esto es identificado en C++ con un "= 0" al final de la declaración.
- Un método virtual puro **requiere** que la función sea implementada en la clase derivada.
- Es **equivalente a los métodos abstractos en Java**

```
class CMotor {
```

```
public:
```

```
//...
```

```
virtual void Display() const = 0; // => no está implementado
```

```
virtual void Input() = 0; // => no está implementado
```

```
//...
```

```
}
```

# Clases Abstractas (Abstract Classes)

- Una clase que contiene uno o más métodos **virtuales** puros pasa a ser una clase **abstracta**.
- Por igual razón que en Java, NO es posible crear instancias de una clase abstracta, pero en **C++ no requiere calificador “abstract”**.

Con la declaración previa para CMotor:

```
CMotor M;           // error
```

```
CMotor * pM = new CMotor; // error
```

# Nueva forma de hacer “casteo” en C++: `static_cast<>` y `dynamic_cast<>`

- El operador `static_cast<>` es la forma preferida para hacer conversiones “seguras” en C++.
- Éste reemplaza al operador tradicional de C y el estilo función de C++.
- Existe también el `dynamic_cast<>`, éste asegura que el resultado de la conversión es un dato compatible. Sólo se aplica a punteros a objetos.
- `dynamic_cast<>` lanza error cuando el resultado de la conversión no es un objeto completo de la clase requerida.
- Ejemplos...

# Ejemplos de `static_` y `dynamic_cast`

## `static_cast`

```
int w = static_cast<int>(3.5);  
bool isOK = static_cast<bool>(1);  
char ch = static_cast<char>(86);
```

## `dynamic_cast`

```
class CBase { };  
class CDerived: public CBase { };  
CBase b; CBase* pb;  
CDerived d; CDerived* pd;  
  
pb = dynamic_cast<CBase*>(&d); // ok: derived-to-base  
pd = dynamic_cast<CDerived*>(&b); // wrong: base-to-derived  
// cuando falla, se retorna null  
// si el casteo es a una referencia, se lanza una excepción
```