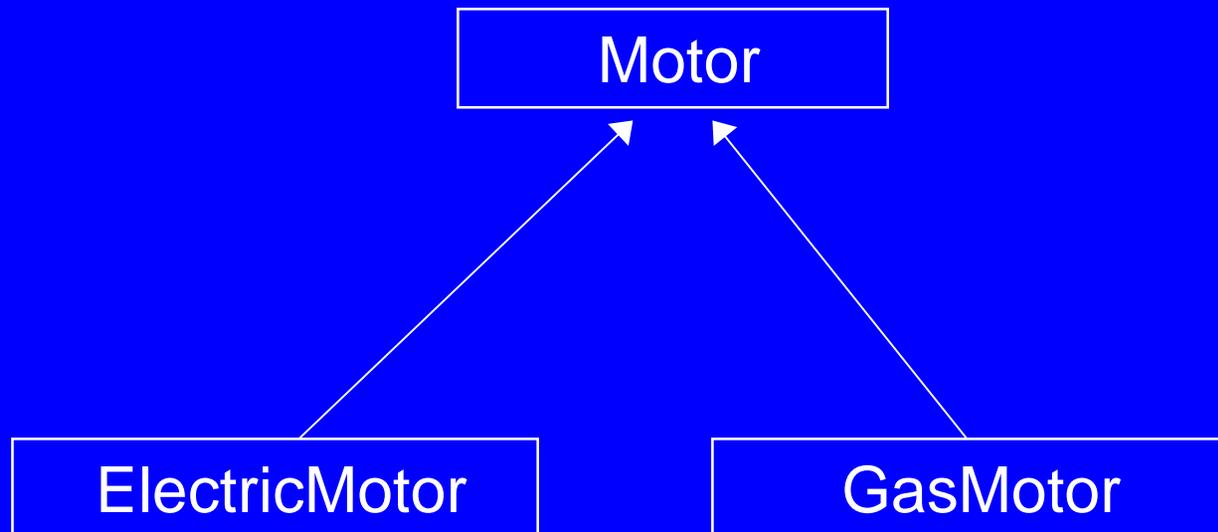


# Polimorfismo y Funciones Virtuales

Agustín J. González  
Versión original de Kip Irvine  
ELO329

# Jerarquía de clases Motor

Consideremos la jerarquía de clases establecida en la sesión 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;
};
```

# Class 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;
};
```

# Clase CGasMotor

```
class CGasMotor :public CMotor {
public:
    CGasMotor();

    CGasMotor(const string & id, int cylinders);

    void Display() const;
    void Input();

private:
    int m_nCylinders;
};
```

# Punteros a objetos de clases derivadas

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;
```

# Polimorfismo

También podemos declarar punteros a una clase base, y luego asignarle la dirección de un objeto de una clase derivada. Esta técnica es llamada polimorfismo.

**Polimorfismo** es un concepto donde un mismo nombre puede denotar objetos de clases diferentes que están relacionadas por una clase base común.

```
CMotor * pM;  
pM = new CElectricMotor;
```

Este tipo de asignación es llamada “ligado” dinámico “**dynamic binding**” porque el tipo exacto asignado al puntero es desconocido hasta el momento de la ejecución.

# Problema de ligado dinámico

Sin embargo, hay un pequeño problema cuando usamos ligado dinámico. Si no somos cuidadosos, el compilador C++ puede llamar las funciones Input and Display de las clase CMotor:

```
CMotor * pM;                // base pointer type
pM = new CElectricMotor;

pM->Input();                 // calls CMotor::Input()
pM->Display();               // calls CMotor::Display()

// more...
```

# Funciones Virtuales (Virtual)

La solución al problema de la transparencia previa es fácil de solucionar declarando las funciones Input and Display como virtuales.

El calificativo virtual le dice al compilador que genere código que mire al tipo del objeto en tiempo de ejecución y use esta información para seleccionar la versión apropiada de la función.

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

# Funciones Virtuales

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

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

# Funciones Virtuales

Ahora las funciones Display e Input son correctamente llamadas desde la clase CElectricMotor:

```
CMotor * pM;  
pM = new CElectricMotor;  
  
pM->Input();          // CElectricMotor::Input()  
pM->Display();        // CElectricMotor::Display()
```

# Funciones Virtuales

A menudo, un puntero será pasado a una función que pide un puntero a la clase base. Cuando la función es llamada, podemos pasar cualquier puntero como parámetro actual, siempre y cuando este sea derivado de la clase base.

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

# Funciones 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 );  
  
// view output...
```

(Salida de la diapositiva 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);
```

# Despliegue de Vectores

La función que despliega tales vectores no necesita saber exactamente qué tipo de puntero están en el vector mientras se llame a funciones 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 de la función 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 cuanto almacenamiento liberar por cada puntero (aun cuando los motores ocupan distintos tamaños).

# Funciones Virtuales Puras

Una función virtual pura no tiene implementación. Esto es identificado con un "**= 0**" al final de la declaración.

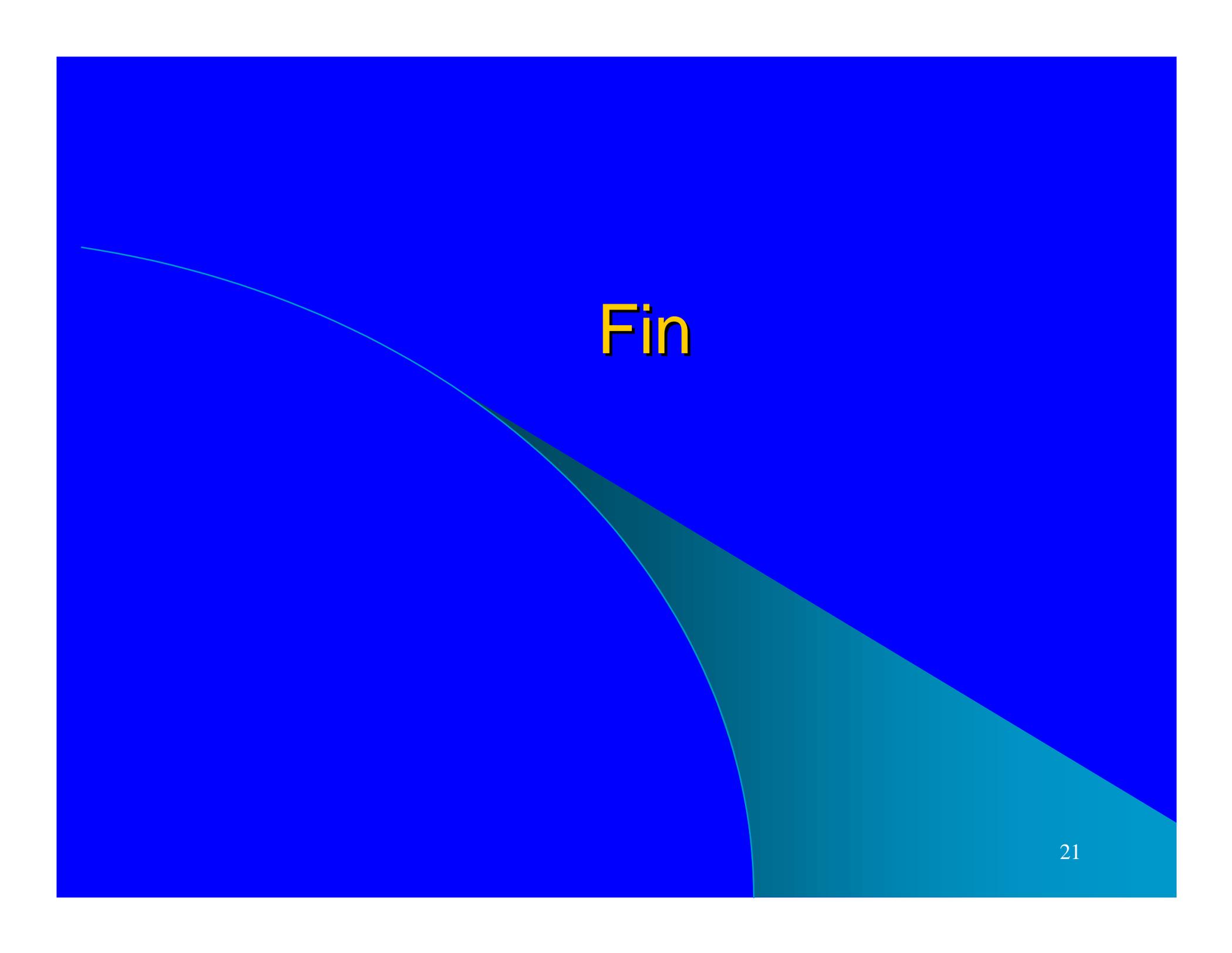
Una función virtual pura requiere que la función sea implementada en la clase derivada.

```
class CMotor {  
public:  
    //...  
  
    virtual void Display() const = 0;  
  
    virtual void Input() = 0;  
  
    //...
```

# Clases Abstractas (Abstract Classes)

Una clase que contiene una o más funciones virtuales puras es llamada **clase abstracta**. NO es posible crear instancias de una clase abstracta.

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

The image features a solid blue background. In the bottom right corner, there is a decorative shape with a teal-to-blue gradient, resembling a stylized fin or a curved wedge. The word "Fin" is written in a bold, yellow, sans-serif font, centered horizontally in the upper half of the image.

Fin