UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Departamento de Electrónica

Polimorfismo en C++, ligado dinámico y Métodos Virtuales, Declaración incompleta de clases, Cambios de tipo

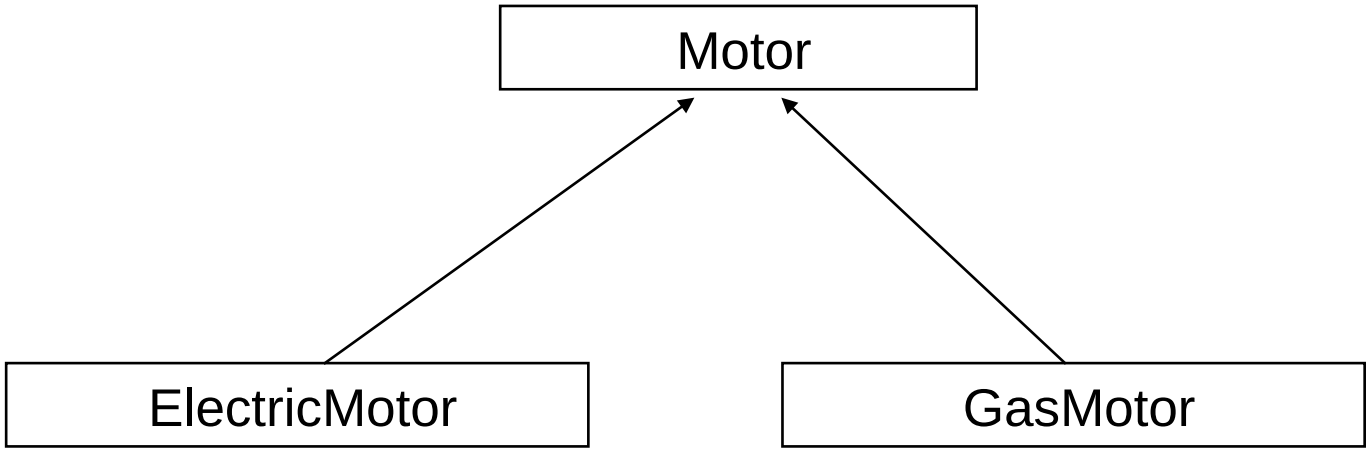
ELO329: Diseño y Programación Orientados a Objetos

Departamento de Electrónica

Universidad Técnica Federico Santa María

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 y referencias 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;
```

- No hay novedad hasta aquí. ¿Qué pasa en C++ cuando el puntero es definido como CMotor *pC; ?

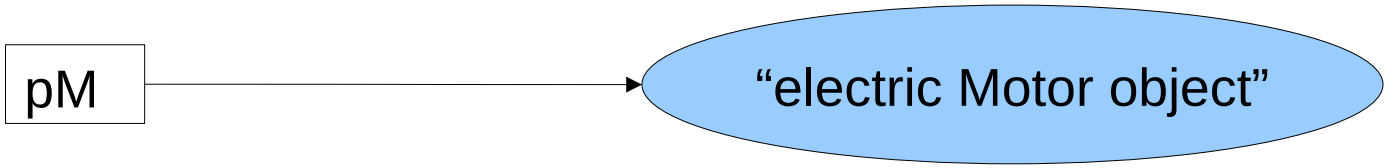
Polimorfismo

- Análogo a Java, en C++ 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, en una de sus formas, un mismo nombre puede referirse a objetos de clases diferentes que están relacionadas por una clase base común**.

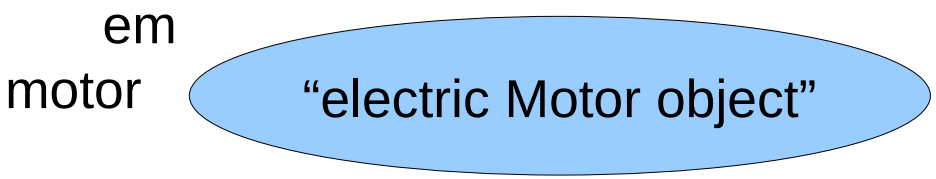
```
CMotor * pM;
pM = new CElectricMotor;  // puntero a motor eléctrico
                          // semántica similar a Java
CElectricMotor em;
CMotor & motor =  em;  // referencia a motor eléctrico
                       // esta opción no existe en Java
```

Polimorfismo

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




```
CElectricMotor em;
CMotor & motor =  em;  // referencia a motor eléctrico
    // dos nombres para un mismo objeto
```



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. Esto es Distinto a Java!

```
CMotor * pM;
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 única opción

// más...
```

Métodos Virtuales (Virtual)

- Si deseamos tener un comportamiento como 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";
}
```

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 lo previo 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
```

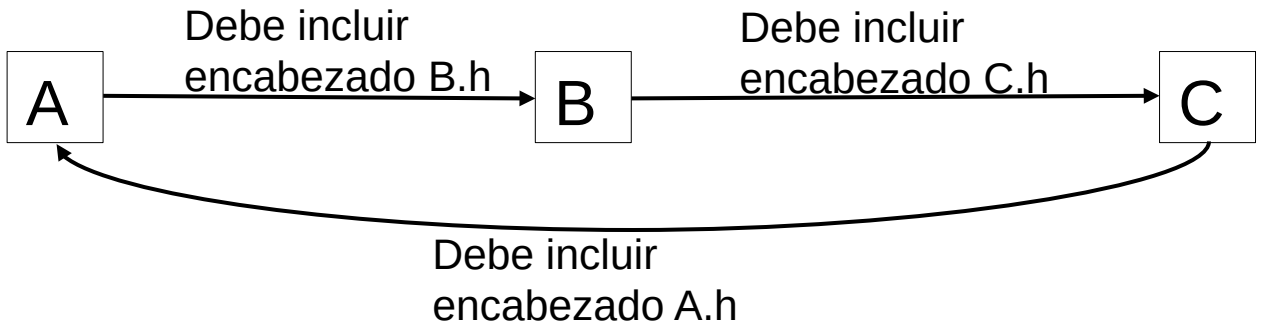
Declaración incompleta de clases (1/3)

- En ocasiones podemos tener este tipo de dependencias:

```
class A  {
  private: B b;
  //  …
};
```

```
class B  {
  private: C c;
  //  …
};
```

```
class C  {
  private: A &a;
  //  …
};
```



- ¿Cómo resolvemos esta dependencia circular? El preprocesador entraría en loop infinito.

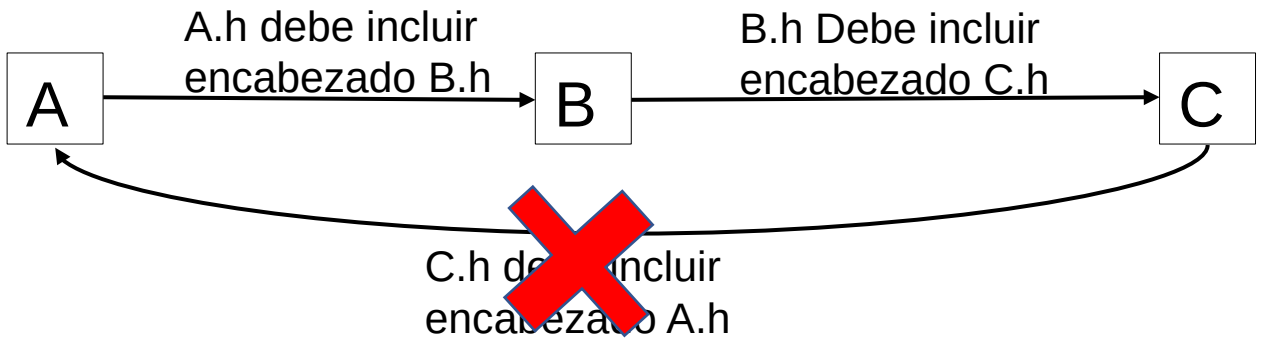
Declaración incompleta de clases (2/3)

- La dependencia cíclica la podemos eliminar usando un declaración incompleta de una clase.

```
#include “B.h”
class A  {
  private: B b;
  //  …
};
```

```
#include “C.h”
class B  {
  private: C c;
  //  …
};
```

```
class A; // declaración incompleta
class C  {
  private: A &a;
  //  …
};
```



Declaración incompleta de clases (3/3)

- Sólo la podemos usar en clases con atributos punteros o referencias a esa clase. En este caso a es una referencia a instancia de clase A.
- En class C **a** también pudo ser A *a;
- Cuando usamos referencias o punteros, el compilador reserva un espacio que no depende del contenido de la clase.

```
#include “B.h”
class A  {
  private: B b;
  //  …
};
```

```
#include “C.h”
class B  {
  private: C c;
  //  …
};
```

```
class A; // declaración incompleta
class C  {
  private: A &a;
  //  …
};
```

Conversiones Implícitas datos

- C++ maneja conversiones automáticamente en el caso de tipos numéricos intrínsecos (int, double, float)
- Mensajes de advertencia (warning) pueden aparecer cuando hay riesgo de pérdida de información (precisión).
  - Hay variaciones de un compilador a otro

- Ejemplos...

Ejemplos de Conversión

```
int n = 26;
double x = n;
double x = 36;
int b = x;      // posible warning, según compilador
bool isOK = 1; // posible warning, según compilador
int n = true;
char ch = 26;  // posible warning, según compilador
int w = 'A';
```

Operación cast

- Al igual que en Java, cuando queremos acceder a métodos sólo existentes en un objeto apuntado o referenciado, necesitamos accederlo con un puntero o referencia donde ese método esté definido.
- Una operación de “casteo” (cast) convierte explícitamente datos de un tipo a otro.
- Es usado en conversiones “seguras” que podrían ser hechas por el compilador.
- Para tipos básicos (no objetos), son usadas para evitar mensajes de advertencia (warning messages).
- El operador tradicional de C pone el nuevo tipo de dato entre paréntesis. C++ mejora esto con una operador cast tipo función.
- Ejemplos...

Ejemplos de cast

```
int n = (int)3.5; // tradicional proveniente de C
int w = int(3.5); // cast como función en C++
bool isOK = bool(15);
char ch = char(86);  // símbolo ascii
string st = string("123");

// situaciones sin conversión disonible, genera error
int x = int("123");  //error
string s = string(3.5); //error

// si usted necesita acceder avalores byte por byte
double x=3.1415;
char *p = (char*)&x; // permitiría analizar formato de números reales
```

“casteo” en C++: static_cast<> y dynamic_cast<>

Dynamic_cast<new_type> (expression)

- Éste asegura que el resultado de la conversión es un dato compatible. Sólo se aplica a punteros o referencias a objetos.
- Cuando no hay compatibilidad:
  - Retorna null cuando el new_type es puntero
  - Lanza una excepción cuando new_type es referencia

Static_cast <new_type> (expression)

- Éste reemplaza el estilo función de C++.
- No se garantiza compatibilidad entre los tipos de datos. Es más rápido que Dynamic_cast.
- Ejemplos...

En C++ no hay equivalente directo a instanceof de java

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
// si tipo es puntero, retorna null cuando hay incompatibilidad
// si tipo es referencia, se lanza una excepción ante falla
```

Otros tipos de cast en C+

reinterpret_cast <new_type> (“puntero”)

- Permite conversión de tipo de clases incluso no relacionadas.
- Simplemente se copia el el valor binario de un puntero al otro.
- No recomiendo su uso en este curso.

const_cast <new_type> (expression)

- Se usa, por ejemplo, para pasar punteros a valores constantes a funciones que no han declaro de ese modo su argumento.

  Ej. void foo(char * str) {….}. // str no es tratado como constante

  Luego tenemos: const char * phola =“Hola”; // *phola es constante

  foo(const_cast<char*> (phola));