INFORMÁTICA INDUSTRIAL

Polimorfismo en C++.

M. Abderrahim, A. Castro, J. C. Castillo
Departamento de Ingeniería de Sistemas y Automática

uc3m | Universidad Carlos III de Madrid

AGENDA

- Superposición y Sobrecarga
- Polimorfismo
- Métodos Virtuales
- Virtualidad en Destructores y Constructores
- Funciones Virtuales Puras y Clases Abstractas

Superposición y Sobrecarga

- En una clase derivada se puede definir una función que ya existía en la clase base, esto se conoce como "overriding", o superposición de una función.

- La definición de la función en la clase derivada oculta la definición previa en la clase base.

- En caso necesario, es posible acceder a la función oculta de la clase base mediante su nombre completo:

```
<objeto>.<nombre clase base>::<método>;
```

Ejemplo de sobrecarga

```
#include <iostream>
using namespace std;
class ClaseA {
  public:
   ClaseA() : datoA(10) {}
   int LeerA() const { return datoA; }
   void Mostrar() {
      cout << "a = " << datoA << endl;
   }
  protected:
   int datoA;
};
class ClaseB : public ClaseA {
  public:
   ClaseB() : datoB(20) {}
   int LeerB() const { return datoB; }
   void Mostrar() {
      cout << "a = " << datoA << ", b = "
           << datoB << endl;
   }
  protected:
   int datoB;
};
```

```
int main() {
   ClaseB objeto;

   objeto.Mostrar();
   objeto.ClaseA::Mostrar();

   return 0;
}
```

```
WJRG-MacBookPro:Clase10 wladimir$ g++ ClaseA6.cpp -o ClaseA6
WJRG-MacBookPro:Clase10 wladimir$ ./ClaseA6
a = 10, b = 20
a = 10
```

Polimorfismo

- ***El polimorfismo*** es un concepto muy importante en la programación orientada a objetos.
- En lo que concierne a clases, el polimorfismo en C++, llega a su máxima expresión cuando las usamos junto con punteros o con referencias.
- C++ nos permite acceder a objetos de una clase derivada usando un puntero a la clase base. En esa capacidad es posible el polimorfismo.
- Por supuesto, sólo podremos acceder a datos y funciones que existan en la clase base, los datos y funciones propias de los objetos de clases derivadas serán inaccesibles.

Ejemplo de polimorfismo

```
#include <iostream>
#include <cstring>
using namespace std;

class Persona {
  public:
   Persona(const  char  *n) { strcpy(nombre,
    n); }
   void VerNombre() { cout << nombre << endl; }
  protected:
   char nombre[30];
};

class Empleado : public Persona {
  public:
   Empleado(const  char  *n) : Persona(n) {}
   void VerNombre() {
      cout << "Emp: " << nombre << endl;
   }
};
```

```
class Estudiante : public Persona {
  public:
   Estudiante(const  char  *n) : Persona(n)
{}
   void VerNombre() {
      cout << "Est: " << nombre << endl;
   }
};

int main() {
   Persona *Pepito = new Estudiante("Jose");
   Persona *Carlos = new Empleado("Carlos");

   Carlos->VerNombre();
   Pepito->VerNombre();
   delete Pepito;
   delete Carlos;

   return 0;
}
```

```
WJRG-MacBookPro:Clase10 wladimir$ g++ Persona1.cpp -o Persona1
WJRG-MacBookPro:Clase10 wladimir$ ./Persona1
Carlos
Jose
```

Métodos Virtuales

- Un método virtual es un método de una clase base que puede ser redefinido en cada una de las clases derivadas de esta.
- Y que una vez redefinido puede ser accedido por medio de un puntero o referencia a la clase base.
- Resolviéndose entonces la llamada en función del objeto referido en vez de en función de con qué se hace la referencia.

Métodos Virtuales

- Significando que si en una clase base definimos un método como virtual, si este método es superpuesto por una clase derivada, al invocarlo usando una referencia o puntero a la clase base, *se ejecutará el método de la clase derivada*.
- Cuando una clase tiene un método virtual; bien directamente o por herencia; se dice que la clase es polimórfica.
- Sintaxis:

```
virtual <tipo> <nombre función>(<lista parámetros>) [{}];
```

Ejemplo de métodos virtuales

```
// Un ejemplo corto que usa funciones virtuales.
#include <iostream>
using namespace std;

class base {
public:
 virtual void quien() { // especificar una clase virtual
cout << "Base" << endl;}
};

class primera_d : public base {
Public:
 // redefinir quien() relativa a primera_d
void quien() {cout << "Primera derivacion" << endl;
}
 };

 class segunda_d : public base {
public:
// redefinir quien relativa a segunda_d
void quien() {cout << "Segunda derivacion" << endl;
}
 };
```

```
int main() {

base obj_base;
base *p;

primera_d obj_primera;
segunda_d obj_segundo;

p = &obj_base;
p->quien(); // acceder a quien() en base

p = &obj_primera;
p->quien(); // acceder a quien() en primera_d

p = &obj_segunda;
p->quien();   // acceder a quien() en segunda_d

return 0;
}
```

```
Base
Primera derivacion
Segunda derivacion
```

Ejemplo de métodos virtuales (usando punteros)

```
#include <iostream>
#include <cstring>
using namespace std;

class Persona {
  public:
   Persona(const  char  *n) { strcpy(nombre,
    n); }
   virtual void VerNombre() { cout << nombre <<
    endl; }
  protected:
   char nombre[30];
};

class Empleado : public Persona {
  public:
   Empleado(const  char  *n) : Persona(n) {}
   void VerNombre() {
      cout << "Emp: " << nombre << endl;
   }
};
```

```
class Estudiante : public Persona {
  public:
   Estudiante(const  char  *n) : Persona(n)
{}
   void VerNombre() {
      cout << "Est: " << nombre << endl;
   }
};

int main() {
   Persona *Pepito = new Estudiante("Jose");
   Persona *Carlos = new Empleado("Carlos");

   Carlos->VerNombre();
   Pepito->VerNombre();
   delete Pepito;
   delete Carlos;

   return 0;
}
```

```
WJRG-MacBookPro:Clase10 wladimir$ g++ Persona2.cpp -o Persona2
WJRG-MacBookPro:Clase10 wladimir$ ./Persona2
Emp: Carlos
Est: Jose
```

Ejemplo de métodos virtuales (usando referencias)

```
#include <iostream>
#include <cstring>
using namespace std;

class Persona {
  public:
   Persona(const  char  *n) { strcpy(nombre,
    n); }
   virtual void VerNombre() { cout << nombre <<
    endl; }
  protected:
   char nombre[30];
};

class Empleado : public Persona {
  public:
   Empleado(const  char  *n) : Persona(n) {}
   void VerNombre() {
      cout << "Emp: " << nombre << endl;
   }
};
```

```
class Estudiante : public Persona {
  public:
   Estudiante(const  char  *n) : Persona(n)
{}
   void VerNombre() {
      cout << "Est: " << nombre << endl;
   }
};

int main() {
   Estudiante Pepito("Jose");
   Empleado Carlos("Carlos");
   Persona &rPepito = Pepito; // Referencia
                    // como Persona
   Persona &rCarlos = Carlos; // Referencia
                     // como Persona
   rCarlos.VerNombre();
   rPepito.VerNombre();

   return 0;
}
```

```
WJRG-MacBookPro:Clase10 wladimir$ g++ Persona3.cpp -o Persona3
WJRG-MacBookPro:Clase10 wladimir$ ./Persona3
Emp: Carlos
Est: Jose
```

Características del mecanismo de Virtualidad

- Una vez que una función es declarada como virtual, lo seguirá siendo en las clases derivadas, es decir, **la propiedad virtual se hereda**.
- Si la función virtual no se define exactamente con **el mismo tipo de valor de retorno y el mismo número y tipo de parámetros** que en la clase base, no se considerará como la misma función, sino como una función superpuesta. → El valor de retorno tiene que coincidir.
- El nivel de acceso no afecta a la virtualidad de las funciones. Es decir, una función virtual puede declararse como privada en las clases derivadas aun siendo pública en la clase base, pudiendo por tanto ejecutarse ese método privado desde fuera por medio de un puntero a la clase base.

Características del mecanismo de Virtualidad

- Una llamada a un método virtual se resuelve siempre en función del tipo del objeto referenciado.
- Una llamada a un método normal se resuelve siempre en función del tipo de la referencia o puntero utilizado.
- Una llamada a un método virtual especificando la clase, exige la utilización del operador de resolución de ámbito ::, lo que suprime el mecanismo de virtualidad. Evidentemente este mecanismo solo podrá utilizarse para el método de la misma clase del contenedor o de clases base del mismo.
- Por su modo de funcionamiento interno (es decir, por el modo en que realmente trabaja el ordenador) las funciones virtuales son un poco menos eficientes que las funciones normales.

Destructores Virtuales

- Supongamos que tenemos una estructura de clases en la que en alguna de las clases derivadas exista un destructor. Un destructor es una función como las demás, por lo tanto, si destruimos un objeto referenciado mediante un puntero a la clase base, y el destructor no es virtual, estaremos llamando al destructor de la clase base. Esto puede ser desastroso, ya que nuestra clase derivada puede tener más tareas que realizar en su destructor que la clase base de la que procede.

- Por lo tanto debemos respetar siempre ésta regla: *si en una clase existen funciones virtuales, el destructor debe ser virtual*.

Ejemplo de destructores virtuales

```
class B {   // Superclase (polimórfica)
  ...
  virtual ~B(); // Destructor virtual
};

class D : public B {    // Subclase (deriva de B)
  ...
  ~D(); // destructor también virtual
};

void func() {
  B* ptr = new D;    // puntero a superclase asignado a objeto de subclase
  ...
  delete ptr;        // Ok: delete es necesario siempre que se usa new
}
```

Constructores Virtuales

- Los constructores no pueden ser virtuales. Esto puede ser un problema en ciertas ocasiones.
- Por ejemplo, el constructor copia no hará siempre aquello que esperamos que haga.
- En general no debemos usar el constructor copia cuando usemos punteros a clases base.
- Para solucionar este inconveniente se suele crear una función virtual "clonar" en la clase base que se superpondrá para cada clase derivada.

Ejemplo de constructores

```
#include <iostream>
#include <cstring>
using namespace std;
class Persona {
  public:
   Persona(const  char  *n) { strcpy(nombre,
    n); }
   Persona(const Persona &p);
   virtual void VerNombre() {
      cout << nombre << endl;
   }
   virtual Persona* Clonar() { return new
    Persona(*this); }
  protected:
   char nombre[30];
};
Persona::Persona(const Persona &p) {
   strcpy(nombre, p.nombre);
   cout << "Per: constructor copia." << endl;
}
```

```
class Empleado : public Persona {
  public:
   Empleado(const  char  *n) : Persona(n) {}
   Empleado(const Empleado &e);
   void VerNombre() {
      cout << "Emp: " << nombre << endl;
   }
   virtual Persona* Clonar() { return new
Empleado(*this); }
};

Empleado::Empleado(const Empleado &e) :
Persona(e) {
   cout << "Emp: constructor copia." <<
endl;
}

class Estudiante : public Persona {
  public:
   Estudiante(const  char  *n) : Persona(n)
{}
   Estudiante(const Estudiante &e);
   void VerNombre() {
      cout << "Est: " << nombre << endl;
   }
   virtual Persona* Clonar() {
      return new Estudiante(*this);
   }
};
```

Ejemplo de constructores

```
Estudiante::Estudiante(const Estudiante &e) : Persona(e) {
   cout << "Est: constructor copia." << endl;
}
int main() {
   Persona *Pepito = new Estudiante("Jose");
   Persona *Carlos = new Empleado("Carlos");
   Persona *Gente[2];
   Carlos->VerNombre();
   Pepito->VerNombre();

   Gente[0] = Carlos->Clonar();
   Gente[0]->VerNombre();

   Gente[1] = Pepito->Clonar();
   Gente[1]->VerNombre();

   delete Pepito;
   delete Carlos;
   delete Gente[0];
   delete Gente[1];
   return 0;
}
```

```
WJRG-MacBookPro:Clase10 wladimir$ g++ Persona4.cpp -o Persona4
WJRG-MacBookPro:Clase10 wladimir$ ./Persona4
Emp: Carlos
Est: Jose
Per: constructor copia.
Emp: constructor copia.
Emp: Carlos
Per: constructor copia.
Est: constructor copia.
Est: Jose
```

Funciones Virtuales Puras

- Una función virtual pura es aquella que no necesita ser definida. En ocasiones esto puede ser útil, como se verá en el siguiente punto.
- El modo de declarar una función virtual pura es asignándole el valor cero.
- Sintaxis:

```
virtual <tipo> <nombre_función>(<lista_parámetros>) = 0;
```

Clases Abstractas

- Una clase abstracta es aquella que posee al menos una función virtual pura.
- No es posible crear objetos de una clase abstracta, estas clases sólo se usan como clases base para la declaración de clases derivadas.
- Las funciones virtuales puras serán aquellas que siempre se definirán en las clases derivadas, de modo que no será necesario definirlas en la clase base.

Clases Abstractas

- A menudo se mencionan las clases abstractas como tipos de datos abstractos, en inglés: *Abstract Data Type*, o resumido ADT.

- Hay varias reglas a tener en cuenta con las clases abstractas:
  - No está permitido crear objetos de una clase abstracta.
  - Siempre hay que definir todas las funciones virtuales de una clase abstracta en sus clases derivadas, no hacerlo así implica que la nueva clase derivada será también abstracta.

Ejemplo de Clases Abstractas

```
#include <iostream>
#include <cstring>
using namespace std;
class Persona {
   public:
     Persona(const char *n) { strcpy(nombre, n); }
     virtual void Mostrar() const = 0;
   protected:
     char nombre[30];
};
class Empleado : public Persona {
   public:
     Empleado(const char *n, int s) : Persona(n), salario(s) {}
     void Mostrar() const;
     int LeeSalario() const { return salario; }
     void ModificaSalario(int s) { salario = s; }
   protected:
     int salario;
};
void Empleado::Mostrar() const {
    cout << "Empleado: " << nombre
         << ", Salario: " << salario
         << endl;
}
```

Ejemplo de Clases Abstractas

```
class Estudiante : public Persona {
   public:
     Estudiante(const char *n, float no) : Persona(n), nota(no) {}
     void Mostrar() const;
     float LeeNota() const { return nota; }
     void ModificaNota(float no) { nota = no; }
   protected:
     float nota;
};
void Estudiante::Mostrar() const {
    cout << "Estudiante: " << nombre
         << ", Nota: " << nota << endl;
}
int main() {
   Persona *Pepito = new Empleado("Jose", 1000);    // (1)
   Persona *Pablito = new Estudiante("Pablo", 7.56);
   char n[30];
   Pepito->Mostrar();
   Pablito->Mostrar();
   delete Pepito;
   delete Pablito;
   return 0;
}
```

```
WJRG-MacBookPro:Clase10 wladimir$ g++ Persona5.cpp -o Persona5
WJRG-MacBookPro:Clase10 wladimir$ ./Persona5
Empleado: Jose, Salario: 1000
Estudiante: Pablo, Nota: 7.56
```

uc3m | Universidad Carlos III de Madrid