uc3m | Universidad Carlos III de Madrid

Informática Industrial I

Grado en Ingeniería en Electrónica Industrial y Automática

Álvaro Castro González

Nicolas Burus

Mohamed Abderrahim

José Carlos Castillo Montoya

Práctica 5

Clases abstractas y polimorfismo

A lo largo de los últimos años, han colaborado en impartir las prácticas presenciales, y por tanto proporcionar comentarios para mejorar algunos aspectos de este documento los profesores siguientes:

Ioannis Douratsos
Fares Abu-Dakka
Juan Carlos González Victores
Raúl Perula
Avinash Ranganath

Práctica 5 – Clases abstractas y polimorfismo.

Las **clases abstractas** están diseñadas para que sean clases padre y sean heredadas por clases hijas. Las clases abstractas presentan los métodos que se deben implementar en las clases hijas pero no se implementan, por lo tanto las clases abstractas no se pueden instanciar. Para instanciar una clase hija que hereda una abstracta, ésta debe implementar todos los métodos abstractos.

En C++ las clases abstractas se declaran utilizando funciones miembro virtuales puras mediante la palabra reservada `virtual`. Una función miembro **virtual pura** es aquella que tiene que ser implementada en una clase derivada. Una clase abstracta es aquella que tiene al menos una función miembro virtual pura.

```
class Operation { // abstract class
      public:
            virtual int method(int a, int b)=0;
};

class Addition: public Operation {
      public:
            int method(int a, int b) {return a+b;}
};

class Subtraction: public Operation {
      public:
            int method(int a, int b) {return a-b;}
};

int main() {
      int n1, n2;
      cout << "Introduce dos enteros:" << endl;
      cin >> n1 >> n2;

      Addition A;
      Subtraction S;
      cout << "Adding " << n1 << "+" << n2 << "=" << A.method(n1,n2) << endl;
      cout << "Subtracting " << n1 << "-" << n2 << "=" << S.method(n1,n2) << endl;

      return 0;
}
```

Como se ha visto, las funciones miembro virtuales puras se declaran como una función virtual **igualada a cero**.

Normalmente las funciones virtuales puras no se implementan en las clases abstractas pero podría hacerse si fuese estrictamente necesario. A continuación, se muestra cómo sería y cómo se realiza la llamada a las funciones miembro implementadas.

```
...

int Operation::method(int a, int b) {
      cout << "Operandos " << a << " y " << b << endl;
      return 0;
}

class Addition: public Operation {
      public:
            int method(int a, int b) {
                  Operation::method(a,b);
                  return a+b;
            }
};

...
```

Las funciones abstractas puras hacen que **no se puedan definir objetos de su clase**.

Definiendo los métodos de una clase base como virtual, permite que las clases derivadas sobrescriban los métodos de la clase base. Lo que hace es que la función que se llama se decida en tiempo de ejecución, esto es el **enlazado dinámico**.

Atendiendo al siguiente código, fijarse en el distinto comportamiento de las funciones `print` y `food`¿Qué salida se esperaría del programa? ¿Por qué?

```
#include <iostream>

class Animal {
      protected:
            int age;
      public:
            Animal(){this->age=-1;}
            Animal(int n){this->age=n;}
            virtual int getAge()=0;
            virtual void print(){cout << "An animal " <<endl;}
            void food() {cout << "not defined yet" << endl;};
};

class Aquatic : public Animal {
      public:
            Aquatic():Animal(0) {}
            Aquatic(int nn):Animal(nn) {}
            int getAge() {
                  return this->age;
            }
            void print() {
                  cout << "Aquatic Animal: " << this->age << " years old" << endl;
            }
            void food() {
                  cout << "little fishes and seaweeds" << endl;
            }
};
```

```
int main() {
      Aquatic*A = new Aquatic(11);
      cout << "Aquatic:" <<endl;
      A->print();
      A->food();

      Animal *animal = a;
      cout << "Animal:" <<endl;
      animal->print();
      animal->food();

      return 0;
}
```

El **polimorfismo** es el mecanismo por el cual un mismo método identificado por su nombre, se comporta de forma distinta dependiendo del tipo de objeto que lo invoque. Es decir, es la capacidad de un método de comportarse de diferente forma según el tipo de objeto que lo llama. En resumen, el polimorfismo permite que una serie de operaciones diferentes tengan el mismo nombre.

Se verán tres formas de realizar el polimorfismo:

1. Sobrecarga de métodos
2. Sobrecarga de operadores
3. Métodos virtuales

Los dos primeros casos corresponden al **polimorfismo estático**, es decir, se decide en tiempo de compilación qué función va a ser llamada. El último caso corresponde al **polimorfismo dinámico** donde es en tiempo de ejecución cuando se decide la función a ejecutar.

1. Sobrecarga de métodos

Una forma habitual de lograr el polimorfismo es a través de la **sobrecarga** de funciones y operadores. La sobrecarga consiste en declarar varias funciones con el mismo nombre pero con diferentes parámetros de entrada, ya sea en el númeroo en el tipo. El compilador utilizará la definición adecuada de la función dependiendo del tipo y del número de parámetros con los que se ha realizado la llamada a la función.

Un ejemplo de esto se puede observar en los constructores del ejemplo anterior de los animales. Para crear instancias de la clase `Aquatic` se podrá realizar la llamada de dos formas diferentes:

```
...

Aquatic*A1 = new Aquatic(11);
Aquatic*A2 = new Aquatic();

...
```

Ejercicio 1

Utilizando el primer ejemplo de los operadores aritméticos, modificarlo para que las clases para que permitantambién sumar o restar tres números utilizando la función `method`.

2. Sobrecarga de operadores

La sobrecarga de operadores proporciona funcionalidad adicional a operadores de C++ como +, *, >=, +=, etc., cuando se aplican a tipos de datos definidos por el usuario.

Siguiendo con el ejemplo de los animales, a continuación se presenta la sobrecarga de un operador unario y otro binario:

```
...

class Aquatic: public Animal {
...
    void operator ++(){this->age++;} //prefix
    void operator ++(int){this->age++;} //postfix
    bool operator>(Aquatic A) {return this->age > A.age;}
};

...

int main(){
      ...
      Aquatic A1(1);
      Aquatic A2(2);
      cout << "A1 age:" << A1.getAge() << endl;
      cout << "A2 age:" << A2.getAge() << endl;

      if(A1> A2) {
            cout << "A1>A2" << endl;
      }
      else {
            cout << "A1<=A2" << endl;
      }

      ++A1;
      A1++;

      cout << "A1 age:" << A1.getAge() << endl;
      cout << "A2 age:" << A2.getAge() << endl;

      if(A1>A2) {
            cout << "A1>A2 " << endl;
      }
      else {
            cout << "A1<=A2 " << endl;
      }
      ...
}
```

Sin embargo, no todos los operadores pueden ser sobrecargados. Los siguientes no lo permiten:

Categoría del operador	Operador
Acceso a miembro	`.` (operator punto)
Resolución de ámbito	`::` (acceso global)
Condicional	`?:` (sentencia condicional)
Puntero a miembro	`*`
Tamaño de tipo de dato	`sizeof(...)`

Ejercicio 2

Utilizando el ejemplo de los animales, implementa la sobrecarga del operador `>>` para la clase Animal de tal forma que nos permita asignarle la edad.

3. Métodos virtuales

La definición de clases abstractas a través de métodos miembro virtuales permite la última forma de polimorfismo que se verá. Esto se ilustra en el siguiente ejemplo donde se determina qué operación se va a realizar en tiempo de ejecución.

```
class Operation {
      virtual int method(int a, int b)=0;
};

class Addition: Operation {
      public:
            int method(int a, int b) {return a+b;}
};

class Subtraction: Operation {
      public:
            int method(int a, int b) {return a-b;}
};

class Multiplication: public Operation {
      public:
            int method(int a, int b) {return a*b;}
};
```

```
int main() {
	int n1, n2;
	cout << "Introduce dos enteros:" <<endl;
	cin >> n1 >> n2;

	srand(time(NULL));	/* initialize random seed: */
	Operation *O;
	switch(rand() % 3){
		case 0: O = new Addition;
			cout << "Operation ADDITION" <<endl;
			break;
		case 1: O = new Subtraction;
			cout << "Operation SUBTRACTION" <<endl;
			break;
		case 2:O = new Multiplication;
			cout << "Operation MULTIPLICATION" <<endl;
			break;
	}
	cout << "Result: " <<O->method(n1, n2) <<endl;
}
```

Ejercicios

- Continuando con la implementación del simulador de un ordenador que se empezó en la práctica anterior. Basándose en lo que ya se tenía, se van a realizar las pertinentes modificaciones para implementar el diagrama UML mostrado en el anexo junto a la descripción de cada una de las clases.
- El programa principal deberá ser capaz de ejecutar el siguiente programa:

```
int main() {
      LineKeyboard LK("logitech");
      Uppercase UP("intel");
      Display D("lg");
      Input *I = &LK;
      LK.connectTo(UP);
      I->connectTo(UP); // do same as before line
      UP.connectTo(D);
      LK.process();

      CharKeyboard CK("logitech char");
      Reverse R("intel reverse");
      Printer P("Ricoh");
      I = &CK;
      (*I) >> R>>P;
      I->process();
}
```