Programación Orientada a Objetos

U4: Polimorfismo

LI. Jaime Jesús Delgado Meraz
j2deme+poo@gmail.com

Instituto Tecnológico de Ciudad Valles

Enero – Junio 2013

Introducción

- La programación orientada a objetos provee de múltiples beneficios comparada con la programación procedural tradicional.
- Conceptos como la herencia y el polimorfismo, hacen la tarea del programador mucho más fácil.
- El uso de la sobrecarga de métodos dada por la herencia, es una de las primera implementaciones del polimorfismo.
- Sin embargo el polimorfismo también incluye el uso de "funciones plantilla" y "clases contenedoras"
- Con la implementación de estas técnicas se promueve la reusabilidad de código.

- Los conceptos relacionados al uso de variables son básicos de todos los lenguajes de programación, y el uso de nombres de variables hace la programación más manejable.
- P.e. *Si un programa debe procesar datos de 1,000 empleados, no necesitamos 1,000 variables para cada salario, si tenemos una variable llamada* `salario`*, esta misma puede contener un número infinito de valores, uno a la vez, en cada ejecución del programa*.
- Igualmente, al generar funciones en los programas, es una característica de gran ayuda, pues permite que las funciones puedan operar sobre cualquiera valores que se pasen como parámetros (siempre y cuando sean del tipo correcto).

- En C++ podemos crear funciones con una variedad de tipos en sus listas de parámetros.
- No estamos limitados a pasar parámetros de tipos escalares (`int`, `double`, `char`), sino que también podemos usar objetos (tales como `Estudiante` o `Automovil`).
- En cualquier caso, el compilador de C++ determina los tipos de los parámetros de la función, cuando esta se crea y se compila, y una vez creada los tipos permanecen fijos.

- La **sobrecarga** es una forma de polimorfismo, using un mensaje consistente que actua apropiadamente para diferentes objetos.
- P.e. *Si tenemos una función llamada `revertir()`, esta función podría cambiar el signo de un valor númerico, revertir el orden de los caracteres de una cadena de texto, cambiar un código de una objeto de tipo `LlamadaTelefonica`, o procesar una devolución a un objeto `Cliente`.*
- Puesto que "revertir" tiene sentido para cada una de las instancias, y diversas tareas se necesitan para cada uno de los 4 casos de `revertir()`, la sobrecarga es una herramienta útil y adecuada.
- Cuando el nombre de función es el mismo, pero la lógica es diferente dependiendo de los argumentos, el sobrecargar funciones es un recurso de gran valor.

Funciones Plantilla

- Algunas veces, las tareas requeridas no son tan diversas, y cuando la lógica de programación es la misma, escribir múltiples funciones sobrecargadas se vuelve aburrido.
- P.e. *Asumiendo una función simple llamada `revertir()` que invierta el signo de un número, tendríamos que crear 3 funciones sobrecargadas, cada uno con diferentes parámetros para que pudiera funcionar con enteros, dobles y flotantes*.

```
1     int revertir(int x){
2         return -x;
3     }
4     double revertir(double x){
5         return -x;
6     }
7     float revertir(float x){
8         return -x;
9     }
```

- Podemos observar que la lógica interna de las 3 funciones es idéntica. Solo difieren en los tipos del parámetro y valor de retorno, por lo tanto sería conveniente escribir una sola función con un tipo de dato indicado con una variable.

```
1    tipoVariable revertir(tipoVariable x){
2        return -x;
3    }
```

- Aunque muestra una buena idea, la función anterior no funciona como tal en C++, pero nos da una idea de la necesidad de crear una **definición de plantilla** (template).

Sintaxis de una función plantilla

- En C++, se pueden crear funciones que utilicen tipos de datos variables. Estas **funciones plantilla** sirven como una *marco* o *patrón* para un grupo de funciones que difieren en el tipo de parámetros que utilizan.
- Una grupo de funciones que se genera de la misma familia es llamada comúnmente como una **familia de funciones**.
- En una función plantilla, por lo menos un parámetro debe ser genérico o parametrizado, es decir, que uno de sus parámetros pueda contener cualquier tipo de dato de C++.

- Si escribimos una función plantilla para `revertir()`, por ejemplo, un usuario podría invocar la función usando cualquier tipo de argumento, siempre y cuando la función unaria del signo menos ($-$) tenga sentido y este definida para ese tipo
- Por lo tanto, si se pasa como parámetro un entero a `revertir()`, nos devolvería un entero negativo, y si pasamos un valor doble positivo, nos devolvería un doble negativo.
- Si hubieramos sobrecargado el operador negativo (signo menos) para revertir los valores de un objeto de tipo `LlamadaTelefonica`, y pasaramos un objeto de tipo `LlamadaTelefonica` a `revertir()`, los costos se revertirían.

- Before you code a function template, you must include a template definition with the following information: ≫ the keyword template ≫ a left angle bracket (¡) ≫ a list of generic types, separated with commas if more than one type is needed ≫ a right angle bracket (¿) Each generic type in the list of generic types has two parts: ≫ the keyword class ≫ an identifier that represents the generic type