CS2013: Programación III

Teoría: Fundamentos de POO I

José Chávez

UTEC

Contenido

Contenido

¿Porque POO?

Entre las principales ventajas de utilizar *Programación Orientada a Objetos (POO)* tenemos las siguientes:

¿Porque POO?

Entre las principales ventajas de utilizar *Programación Orientada a Objetos (POO)* tenemos las siguientes:

- Modularidad. Dividir el problema en partes pequeñas, esto facilita y agiliza la resolución de problemas.

¿Porque POO?

Entre las principales ventajas de utilizar *Programación Orientada a Objetos (POO)* tenemos las siguientes:

- Modularidad. Dividir el problema en partes pequeñas, esto facilita y agiliza la resolución de problemas.
- Mantenimiento. Permite añadir, modificar o eliminar de una manera sencilla.

¿Porque POO?

Entre las principales ventajas de utilizar *Programación Orientada a Objetos (POO)* tenemos las siguientes:

- Modularidad. Dividir el problema en partes pequeñas, esto facilita y agiliza la resolución de problemas.
- Mantenimiento. Permite añadir, modificar o eliminar de una manera sencilla.
- Reutilización de código. La *Herencia* permite heredar características de una super-clase en una sub-clase.
- Flexibilidad. Utilizando *Polimorfismo*, podemos hacer que una función se adapte, y cambia, a cualquier clase.

Contenido

Abstracción

La *Abstracción* es una de las característica principales de POO. Esta técnica permite separar la interfaz de la implementación, ocultando los detalles irrelevantes para el usuario.

Abstracción

La *Abstracción* es una de las característica principales de POO. Esta técnica permite separar la interfaz de la implementación, ocultando los detalles irrelevantes para el usuario.

De este modo, el usuario solo requiere comprender la interfaz (comandos) y no cómo se representa/implementa la estructura interna de los datos.

Encapsulación

La *Encapsulación* es otro de los fundamentos de POO. Esta técnica permite ocultar datos y funcionalidades en una unidad (clase).

Encapsulación

La *Encapsulación* es otro de los fundamentos de POO. Esta técnica permite ocultar datos y funcionalidades en una unidad (clase).

En una clase encapsulada, los usuario/programadores pueden usar la interfaz, pero no tiene acceso a la implementación o datos *privados*. *Saben lo que hace, pero no saben como*.

Tipo de Dato Abstracto (TDA)

La clase que use abstracción y encapsulamiento define un *Tipo de Dato Abstracto (TDA)*.

Tipo de Dato Abstracto (TDA)

La clase que use abstracción y encapsulamiento define un *Tipo de Dato Abstracto (TDA)*.

En una TDA, el diseñador se preocupa de la implementación de la clase. Los usuarios/programadores que usen la clase no necesitan saber como funciona clase, solo tienen que preocuparse en lo que hace.

Contenido

Definiendo un Tipo de Dato Abstracto

La esto usaremos un ejemplo. Supongamos que queremos representar un Alumno con una clase. Para ello podemos definir algunas funcionalidades:

Definiendo un Tipo de Dato Abstracto

La esto usaremos un ejemplo. Supongamos que queremos representar un Alumno con una clase. Para ello podemos definir algunas funcionalidades:

- La función `ingresar_nota` para ingresar un nota del alumno.

Definiendo un Tipo de Dato Abstracto

La esto usaremos un ejemplo. Supongamos que queremos representar un Alumno con una clase. Para ello podemos definir algunas funcionalidades:

- La función `ingresar_nota` para ingresar un nota del alumno.
- La función `print` para imprimir el promedio notas de un alumno.

¿Como usaríamos la interfaz?

```
int main() {
  Alumno obj1;
  obj1.ingresar_nota(11);
  obj1.ingresar_nota(17);
  obj1.ingresar_nota(8);
  print(cout, obj1);
}
```

¿Como usaríamos la interfaz?

```
int main() {
  Alumno obj1;
  obj1.ingresar_nota(11);
  obj1.ingresar_nota(17);
  obj1.ingresar_nota(8);
  print(cout, obj1);
}
```

```
Nota Final:  12
```

Definiendo la clase `Alumno`

- La clase alumno tendrá un atributo llamado `notas` de tipo `vector<int>`, el cual almacenará todas las notas ingresadas.
- La clase también tendrá un método llamado `ingresar_nota` el cual debe permitir ingresar un número entero y almacenarlo en `notas`.
- Dado que no es método de la clase, la función auxiliar `print` debe ser definida fuera de la clase.

Definiendo los métodos y atributos

```
struct Alumno{
    // Metodos:
    void ingresar_nota(int nota);

    // Atributos
    vector<int> notas;
};

void Alumno::ingresar_nota(int nota){
    notas.push_back(nota);
}
```

Definiendo las funciones auxiliares

```
struct Alumno{
    ...
};

ostream &print(ostream& os, const Alumno& item){
    float promedio = 0.;
    for(const int nota: item.notas)
        promedio += nota;
    promedio = promedio/item.notas.size();
    os << "Promedio: " << promedio << endl;
    return os;
}
```

Constructores

Cada clase puede establecer como deben ser inicializados sus objetos. Las clases controlan este proceso a través de los *Constructores*.

```
struct Alumno{
    Alumno(string s, int n): nombre(s), edad(n){}

    // Atributos
    string nombre;
    int edad;
};
```

El constructor permite inicializar los atributos.

Constructores

Cada clase puede establecer como deben ser inicializados sus objetos. Las clases controlan este proceso a través de los *Constructores*.

```
int main(){
    Alumno obj1("Julian", 18);
    ...
}
```

El constructor es llamado cuando un objeto de la clase es creado.

Constructores

Cada clase puede establecer como deben ser inicializados sus objetos. Las clases controlan este proceso a través de los *Constructores*.

```
struct Alumno{
    Alumno() = default;
    Alumno(string s): nombre(s){}
    Alumno(int n): edad(n){}
    Alumno(string s, int n): nombre(s), edad(n){}
};
```

Los constructores debe llevar el mismo nombre que la clase.

Contenido

Nuestra clase no esta encapsulada

Hasta ahora hemos definido la interfaz de la clase `Alumno`; sin embargo, cualquier usuario podría modificar los atributos:

Nuestra clase no esta encapsulada

Hasta ahora hemos definido la interfaz de la clase `Alumno`; sin embargo, cualquier usuario podría modificar los atributos:

```
int main(){
    Alumno obj1("Julian");
    obj1.ingresar_nota(11);

    obj1.nombre = "Maria";
    obj1.notas[0] = 17;
    print(cout , obj1);
}
```

```
Alumno: Maria, Nota Final: 17
```

Especificadores `public` y `private`

En C++ se utiliza especificadores de acceso para hacer cumplir la encapsulación:

Especificadores `public` y `private`

En C++ se utiliza especificadores de acceso para hacer cumplir la encapsulación:

- `public`: Miembros públicos son accesibles para todas las partes del programa. Estos definen la interfaz de la clase.

Especificadores `public` y `private`

En C++ se utiliza especificadores de acceso para hacer cumplir la encapsulación:

- `public`: Miembros públicos son accesibles para todas las partes del programa. Estos definen la interfaz de la clase.
- `private`: Miembros privados son accesibles para todas los métodos de la clase, pero no para el código que usa la clase. Esta sección se encarga de encapsular la implementación.

¿struct o class?

Podemos utilizar cualquiera de estos dos *keywords*, la única diferencia radica en que:

¿`struct` o `class`?

Podemos utilizar cualquiera de estos dos *keywords*, la única diferencia radica en que:

- Al utilizar `struct`, los miembros son declarados públicos por defecto.

¿`struct` o `class`?

Podemos utilizar cualquiera de estos dos *keywords*, la única diferencia radica en que:

- Al utilizar `struct`, los miembros son declarados públicos por defecto.
- Al utilizar `class`, los miembros son declarados privados por defecto.

Encapsulando nuestra clase

Utilizando los especificadores, podemos encapsular nuestra clase

Encapsulando nuestra clase

Utilizando los especificadores, podemos encapsular nuestra clase

```
class Alumno{
public:
    Alumno(string s): nombre(s){}
    void ingresar_nota(int nota);
private:
    string nombre;
    int edad;
    vector<int> notas;
};
```

Encapsulando nuestra clase

Ahora nadie tiene acceso a los miembros privados:

Encapsulando nuestra clase

Ahora nadie tiene acceso a los miembros privados:

```
int main(){
    Alumno obj1("Julian");
    obj1.ingresar_nota (11);

    obj1.nombre = "Maria";
    obj1.notas[0] = 17;
    print(cout , obj1);
}
```

```
ERROR 1:  Alumno::nombre is private
ERROR 2:  Alumno::notas is private
```

Funciones `friend`

Note que ahora nuestra función `print` ya no compila debido a que los miembros `nombre` y `notas` son privados.

Funciones friend

Note que ahora nuestra función print ya no compila debido a que los miembros nombre y notas son privados.

```
ostream &print(ostream& os, const Alumno& item){
    float promedio = 0.;
    for(const int nota: item.notas)
        promedio += nota;
    promedio = promedio/item.notas.size();
    os << "Alumno: " << item.nombre << ", ";
    os << "Promedio: " << promedio << endl;
    return os;
}
```

Funciones `friend`

Una clase puede permitir a otra clase/función acceder a sus miembros privados haciendo la clase/función *amiga* o `friend`.

Funciones `friend`

Una clase puede permitir a otra clase/función acceder a sus miembros privados haciendo la clase/función *amiga* o `friend`.

```
class Alumno{
friend ostream &print(ostream&, const Alumno&);
public:
    Alumno(string s): nombre(s){}
    void ingresar_nota(int nota);
private:
    string nombre;
    int edad;
    vector<int> notas;
};
```

Contenido

El constructor de copia

Se define cuando el primer parámetro del constructor es una referencia al tipo de clase.

El constructor de copia

Se define cuando el primer parámetro del constructor es una referencia al tipo de clase.

```
class Alumno{
public:
    Alumno() = default;
    Alumno(const Alumno&);
    ...
};

Alumno::Alumno(const Alumno& item){
    nombre = item.nombre;
    notas = item.notas;
}
```

El Constructor de Copia

En nuestra interfaz el constructor de copia funcionaría de la siguiente manera:

El Constructor de Copia

En nuestra interfaz el constructor de copia funcionaría de la siguiente manera:

```
int main(){
    Alumno obj1("Julian");
    obj1.ingresar_nota(11);
    obj1.ingresar_nota(17);
    obj1.ingresar_nota(8);

    Alumno obj2 = obj1;
    print(cout, obj2);
}
```

```
Alumno: Julian, Promedio: 12
```

El Constructor de Copia

Al igual que el constructor por defecto, si no creamos un constructor de copia el compilador creara un constructor de copia sintético.

El Constructor de Copia

Al igual que el constructor por defecto, si no creamos un constructor de copia el compilador creara un constructor de copia sintético.

Es decir no es necesario definir el constructor de copia, a no ser que se quiera realizar una copia distinta.

El Constructor de Movimiento

Este constructor nos permite mover objetos en lugar de copiarlos. Las ventajas/desventajas:

El Constructor de Movimiento

Este constructor nos permite mover objetos en lugar de copiarlos. Las ventajas/desventajas:

- Evitar la copia innecesaria de grandes cantidades de datos: matrices, estructuras de datos, etc.

El Constructor de Movimiento

Este constructor nos permite mover objetos en lugar de copiarlos. Las ventajas/desventajas:

- Evitar la copia innecesaria de grandes cantidades de datos: matrices, estructuras de datos, etc.
- No existen beneficio alguno si la clase no hace uso de memoria dinámica.

El Constructor de Movimiento

Este constructor nos permite mover objetos en lugar de copiarlos. Las ventajas/desventajas:

- Evitar la copia innecesaria de grandes cantidades de datos: matrices, estructuras de datos, etc.
- No existen beneficio alguno si la clase no hace uso de memoria dinámica.
- Añade más código a la clase.

El Constructor de Movimiento

Este constructor nos permite mover objetos en lugar de copiarlos. Las ventajas/desventajas:

- Evitar la copia innecesaria de grandes cantidades de datos: matrices, estructuras de datos, etc.
- No existen beneficio alguno si la clase no hace uso de memoria dinámica.
- Añade más código a la clase.

Para implementar este constructor necesitamos saber que es el *Lvalue* y *Rvalue*.

Lvalue

Un Lvalue es un objeto que tiene una ubicación en memoria y es identificable. Por ejemplo:

```
int i = 7          // i es un Lvalue
int *pi = &i;      // i es indentificable,
                   // es decir, iene una direccion
i = 10;            // i puede ser modificado
```

Rvalue

Hablamos de un objeto Rvalue cuando este no es indentificable en memoria. Podemos identificar un Rvalue rápidamente cuando el objeto no tiene nombre. Por ejemplo:

```
int i = 7;           // i es un Lvalue, pero 7 es un Rvalue
int k = i+3;         // (i+3) es un Lvalue
int *pi = &(i + 3); // ERROR: (i+3) no es identificable
i + 3 = 10;          // ERROR: (i+3) no es modificable
3 = i;               // ERROR: no se puede asignar
                     //        un valor a un Rvalue
```

Referencias *Rvalue*

Una referencia de Rvalue es aquella que se vincula unicamente a un Rvalue, es decir a objetos que van a ser destruidos. Se obtiene utilizando &&, en lugar de &. Por ejemplo:

```
int i = 42;
int &&c1 = 8;              // OK
int &&d1 = i * 42;         // OK
const int &d2 = i * 42; // OK: Se puede vincular una
                        //     referencia Lvalue a un
                        //     rvalue usando 'const'
```

Referencias *Rvalue*

Una referencia de Rvalue es aquella que se vincula unicamente a un Rvalue, es decir a objetos que van a ser destruidos. Se obtiene utilizando &&, en lugar de &. Por ejemplo:

```
int i = 42;
int &b1 = i;        // OK: b1 es una referencia Lvalue
int &b2 = i * 7;    // ERROR: (i * 42) es un Rvalue
int &&c2 = i;       // ERROR: no se puede vincular
                    // una referencia Rvalue a un Lvalue
```

La función `move`

Es imposible vincular de manera directa una referencia `rvalue` a una variable definida como referencia `rvalue`:

```
int &&i = 7;
int &&j = i;
```

La función move

Es imposible vincular de manera directa una referencia rvalue a una variable definida como referencia rvalue:

```
int &&i = 7;
int &&j = i;
```

La función move permite realizar esta operación, al castear un Lvalue a su respectiva referencia Rvalue.

```
int &&i = 7;
int &&j = move(i);
```

Definiendo el Constructor de Movimiento

```
class Foo{
public:
    Foo(int x){ ptr = new int(x);}
    Foo(const Foo& item): ptr(item.ptr){
        cout << "Constructor de Copia" << endl;
    }
    Foo(Foo&& item): ptr(item.ptr){
        item.ptr = NULL;
        cout << "Constructor de Movimiento" << endl;
    }
    print(){cout << *ptr << endl;}
    ~Foo(){
        delete ptr;
        ptr=NULL;
    }
private:
    int* ptr;
```

Definiendo el Constructor de Movimiento

Luego en nuestra interfaz:

```
int main(){
    Foo p1(7);

    Foo p2 = p1;
    Foo p3 = move(p1);
}
```

```
Resultado:
Constructor de Copia
Constructor de Movimiento
```

Definiendo el Constructor de Movimiento

Despues de aplicar el Constructor de Movimiento, se destruye el objeto. Para nuestro ejemplo el objeto p1:

```
int main(){
    Foo p1(7);
    Foo p2 = p1;
    p2.print();
    p1.print();
    Foo p3 = move(p1);
    p3.print();
    p1.print(); // No visualizaremos el numero 7
}
```

Contenido

Resumen

- Una clase nos permite definir nuevos tipos de datos. Esto hace que nuestro programa sea más eficiente.

Resumen

- Una clase nos permite definir nuevos tipos de datos. Esto hace que nuestro programa sea más eficiente.
- Si la clase no declara un Constructor por defecto, Copia o Movimiento entonces el compilador las sintetiza.

Resumen

- Una clase nos permite definir nuevos tipos de datos. Esto hace que nuestro programa sea más eficiente.
- Si la clase no declara un Constructor por defecto, Copia o Movimiento entonces el compilador las sintetiza.
- Si una clase necesita de un Destructor, entonces también necesita de un Constructor de Copia y Movimiento.