I102 - Paradigmas de Programación

OOP — SOBRECARGA DE OPERADORES

5

Prof. Mariano Scaramal

Ingeniería en Inteligencia Artificial

Esta Presentación Cubre…

- Conceptos:
  - Los calificadores *const* y *static* aplicados a clases.
  - Sobrecarga de operadores:
    - Ejemplo con operador lógico.
    - Ejemplo con operador aritmético.
  - Funciones y clases friend.
  - Copiar Objetos
    - Shallow Copy
    - Deep Copy

Métodos, Atributos y Objetos Tipo *const*

- Los **métodos** *const* no pueden realizar cambios en el objeto. Esto se logra:
  - No pudiendo cambiar valores de atributos dentro del método,
  - No pudiendo llamar métodos que no sean *const*, dado que estos sí podrían cambiar atributos del objeto.
- Los **atributos** o variables miembro *const*, deben seguir las siguientes reglas:
  - Deben ser inicializados cuando el objeto es creado,
  - No podrán volver a cambiar dado que son calificados como *inmutables*.
  - Sólo las variables ***static const*** pueden ser inicializadas (no las *static*).
- Los **objetos** *const* no pueden ser modificados:
  - Los atributos son declarados en el constructor,
  - Sólo se pueden llamar métodos tipo *const*, forzado por el compilador.

Métodos, Atributos y Objetos Tipo *const* (cont'd)

Como estas reglas pueden ser demasiado estrictas, existe el calificador *mutable* que permite, por ejemplo, modificar un atributo con un método constante.

Así, *counter* puede ser modificado por un método *const*.

Esto puede hacerse incluso habiendo definido el objeto como *const*.

```
int MyClass::sInt = 8;

int main() {
    const MyClass obj(10);     // Const object
    obj.incrementCounter();  // Permitido
    obj.incrementCounter();  // Permitido
    std::cout << "Counter: " << obj.getCounter() <<
    std::endl;  // Imprime: Counter: 2
    // obj.sInt  no puede ser accedido, es private
    return 0;
}
```

```
class MyClass {
private:
    mutable int counter;   // Atributo mutable
    const int cInt;        // Se debe inizialiar en el constructor
    const int cIInt = 3;   // Permitido
    static int sInt;       // Se debe definir fuera de la clase
    //static int sIInt = 5;    // No está permitido
    static const int y = 4;  // Permitido

public:
    MyClass() : counter(0) , cInt(3) {}
    MyClass(int val) : counter(val) , cInt(3) {}

    void incrementCounter() const {
        counter++;  // Permitido porque 'counter' es mutable
    }

    int getCounter() const { return counter; }
};
```

Atributos Tipo *static*

Los atributos *static* son variables que pueden ser compartidos entre todas las instancias de la clase (es decir, son atributos de clase).

La inicialización se hace fuera de la clase aunque son declarados dentro.

```
#include <iostream>

class MyClass {
private:
    static int counter;  // Variable de clase
    static const int PI = 3.14159; // Se debe definir

public:
    void incrementCounter() {
        counter++;
    }

    static int getCounter() {
        return counter;
    }
};
```

```
// Inicializacion del atributo static fuera de la clase
int MyClass::counter = 0;   // No es necesario
instanciar una clase para acceder la variable

int main() {
    MyClass obj; // Major definer static en constructor
    obj.incrementCounter(); obj.incrementCounter();
    std::cout << "Counter: " << obj.getCounter() <<
    std::endl;  // Output: Counter: 2

    MyClass obj2; obj2.incrementCounter();
    std::cout << "Counter: " << obj2.getCounter() <<
    std::endl;  // Imprime: Counter: 3

    return 0;
}
```

Métodos Tipo *static*

Los métodos *static* no pertenecen a una instancia de una clase (métodos de una clase). Debido a esto, sólo pueden utilizar atributos del tipo static.

Su uso incluye librerías de matemática, donde los valores son pasados para computar una operación, y donde las constantes son definidas como static.

```
#include <iostream>
#include <cmath>  // Para calcular la exponencial

class Math {
public:
   // Constante tipo static
   static const double SQRT_2PI;

   // Metodo static para  calcular la distribucion Gaussiana
   static double gaussian(double x, double mean, double stddev) {
      double exponent = -((x - mean) * (x - mean)) /
      (2 * stddev * stddev);
      return (1 / (stddev * SQRT_2PI)) * std::exp(exponent);
   }
};
```

```
const double Math::SQRT_2PI = 2.5066282746310002;

int main() {
   double x = 0.0;          // Punto a evaluar
   double mean = 0.0;       // Valor medio
   double stddev = 1.0;     // Desvio standard

   // Se usa el metodo estático para calcular el valor
   std::cout << "El valor de la gaussiana de x = " << x << " es "
         << Math::gaussian(x, mean, stddev)
         << " (o 1/SQRT_2PI)" << std::endl;
   // Valor de la gaussiana en x = 0 es 0.398942 (o 1/SQRT_2PI)
   return 0;
}
```

Sobrecarga de Operadores

En el caso de tener distintas instancias de una clase con distintos datos, surgen las pregunta:

1. Como comparar, sumar, concatenar o realizar alguna otra operación con estos objetos?
2. Tiene algún sentido comparar dos objetos? Como se sumarían 2 satélites?

Comencemos respondiendo la pregunta 2. La respuesta se puede obtener pensando en que sus nombre (tipo *string)* se concatenan mediante la operación “+”. Pero, en realidad, la operación puede ser definida según la necesidad del usuario.

La primer pregunta es más corta de responder: utilizando sobrecarga de operadores.

```
class X
{
    int i;
    X& operator+(const X &r)
    {
        i=i-r.i;
        return *this;
    }
};
```

Sobrecarga de Operadores

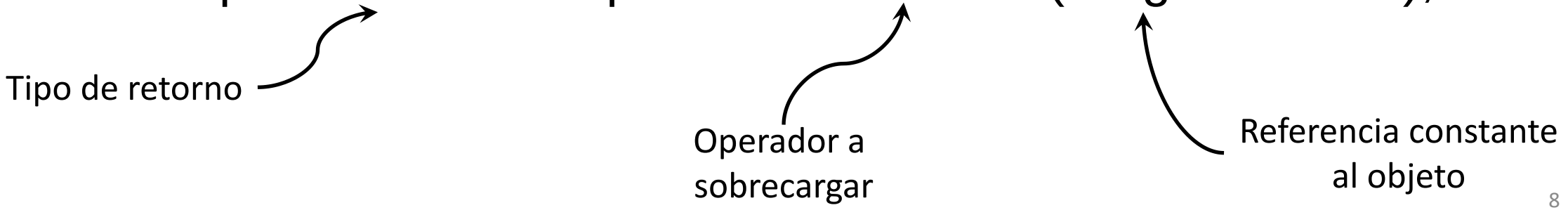
Como muchos otros programas, C++ permite definir un determinado comportamiento dado a una operación entre objetos, a esto se lo denomina, sobrecarga de operadores.

Usualmente, los operadores que se sobrecargan suelen ser:

==, !=, <, >, =, +

La sobrecarga de operadores se logra mediante la adición de un método, solo que su nombre es ***operator*** seguido del operador a sobrecargar. Por ejemplo:

<tipoDeRetorno> operator <símbolo> (<argumentos>);



Sobrecarga de Operador Lógico – Ejemplo 1

El método que sobrecarga el operador “>” tiene como parámetro el objeto de la derecha que interviene en la operación (*car2*), mientras que el de la izquierda es el propio objeto con puntero *“this”*.

```
class Car {
private:
    std::string maker;
    double length;  // Largo del auto en centímetros

public:
    Car(std::string maker, double length) : maker(maker),
length(length) {}

    // Sobrecargando el operador '>' para comparar por length
    bool operator>(const Car& other) const {
        return this->length > other.length;
    }
};
```

```
int main() {
    Car car1("Toyota", 450.0);
    Car car2("Ford", 400.0);

    if (car1 > car2)
        std::cout << "Car 1 es mas largo que car 2\n";
    else
        std::cout << "Car 1 no es mas largo que car 2\n";
    // Imprime: Car 1 es mas largo que car 2
    return 0;
}
```

Sobrecarga de Operador Lógico – Ejemplo 2

Aquí la comparación se hace de la misma manera.
Notar que los objetos iguales no son el mismo objeto, aquí es el desarrollador quien decide como dos objetos son iguales.

```
#include <iostream>
#include <string>

class Car {
private:
   std::string maker; int year;

public:
   Car(std::string maker, int year) : maker(maker), year(year) {}

   // Sobrecarga de '==' para ver si los autos son iguales
   bool operator==(const Car& other) const {
      return (this->maker == other.maker) &&
      (this->year == other.year);
   }
};
```

```
int main() {
   Car car1("Toyota", 2010); Car car2("Toyota", 2010);
   Car car3("Ford", 2010);

   // Comparar car1 y car2
   if (car1 == car2) // Imprime: Car1 y car 2 son iguales.
      std::cout << "Car 1 y car 2 son iguales.\n";
   else
      std::cout << "Car 1 y car 2 son diferentes.\n";
   // Comparar car1 y car3
   if (car1 == car3) // Imprime: Car1 y car 3 son diferentes.
      std::cout << "Car 1 y car 3 son iguales.\n";
   else
      std::cout << "Car 1 y car 3 son diferentes.\n";

   return 0;
}
```

Sobrecarga de Operador Aritmético – Ejemplo 1

Lo mismo puede ocurrir con el operador “+”, el programador puede estar interesado en conocer la longitud total que ocupa una flota de autos y por este motivo sobrecarga el operador para conocer la longitud total de los autos.

```
#include <iostream>
#include <string>

class Car {
private:
    std::string maker;
    double length;  // Largo del auto en centimetros

public:
    Car(std::string maker, double length) : maker(maker), length(length) {}

    // Sobrecarga de'+' para sumar las longitudes de los autos
    double operator+(const Car& other) const {
        return this->length + other.length;
    } // El segundo operando, car2, es pasado como parámetero
};
```

```
int main() {
    Car car1("Toyota", 450.0);
    Car car2("Ford", 400.0);

    // Imrpime la suma de las longitudes de los autos
    std::cout << “La longitud total de los autos es: " <<
     car1 + car2 << " cm\n";

    return 0;
}
```

Sobrecarga de Operador Aritmético – Ejemplo 2

```
#include <iostream>
#include <string>

class Car {
private:
   std::string maker; double length;  // Largo del auto

public:
   Car(std::string maker, double length) : maker(maker),
   length(length) {}

   // Sobrecarga '+=' para obtener un vehículo mas largo
   Car& operator+=(double extraLength) {
      this->length += extraLength;     // Agrega largo extra
      return *this;          // Devuelve el objeto modificado
   }

   void display() const { // Muestra la información
      std::cout << “Fabricante: " << make << ", Largo: " <<
      length << " cm\n";
   }
};
```

Se retorna *Car&*, porque se desea tener concatenación para esta operación.

En general, las operaciones que modifican el objeto (+, +=, *, -, etc), se retornan por referencia (&). Las operaciones que no modifican el objeto (==, >, !=, etc), se retorna por valor.

```
int main() {
   Car car("Toyota", 450.0);
   // Agrega 50 cm y luego 30 cm a la longitud original del auto
   (car += 50) += 30;
   car.display();  // Muestra que la longitud del auto es 530 cm

   return 0;
}
```

Funciones *friend*

Las funciones *friend* son funciones que no forman parte de un objeto (no son un método) pero que pueden tener acceso a miembros *public*, private y *protected* de la clase.

La declaración de la función *friend* se hace dentro de la clase, pero su declaración se realiza fuera de la misma.

La “amistad” no es mutua, es decir, si un objeto tiene una función *friend*, la función puede acceder los miembros *public*, *private* y *protected*, pero la clase no puede acceder a ninguna variable de la función.

Funciones *friend* - Ejemplo

- La declaración se hace dentro de la clase y su definición por fuera.
- La función tiene acceso a los atributos *private* oilStatus y *private waterStatus*.
- Tener precaución en como usar esto para no romper el encapsulamiento.

```
class Plane {
private:
   double oilStatus;
   double waterStatus;
public:
   Plane(double oil, double water):
         oilStatus(oil), waterStatus(water){}

   void setOilStatus(double oil){oilStatus = oil;};

   void setWaterStatus(double water){
       waterStatus = water;
   };
   friend void showPlaneData(Plane pl);
};
```

```
void showPlaneData(Plane pl){
   std::cout << “The oil status is: " << pl.oilStatus
         << ", the water status is: " << pl.waterStatus
         << std::endl;
}

int main(){
   Plane myPlane(75, 100);
   showPlaneData(myPlane);
   // Imprime: Plane oil is: 75, Plain water is: 100
   return 0;
}
```

Clases *friend*

Similarmente, las clases *friend* son clases que pueden tener acceso a miembros *public*, *private* y *protected* de la clase que declaran esta clase como *friend.*

En este caso, la declaración de *friend* es a nivel clase, por lo que sólo se declara la clase como *friend*, no a nivel objeto.

Al igual que para una función *friend*, la “amistad” tampoco es mutua entre clases. Si una clase tiene una clase *friend*, esta puede acceder los miembros *public*, *private* y *protected de la otra clase*, pero no funciona en la dirección opuesta.

Clases *friend* - Ejemplo

```
class Car;

class Engine {
private:
   double horsepower;
   bool isOn;
   void displayEngine() {
      std::cout << "Power: " << horsepower
                << ", Engine on? " <<
                (isOn ? "Yes" : "No") << std::endl;
   }
public:
   Engine(double horsepower, bool isOn) :
          horsepower(horsepower),isOn(isOn) {}

    // Se hace amigo de la classe Engine
   friend class Car;
};
```

```
class Car {
private:
   std::string maker;  double length;
public:
   Car(std::string maker, double length) : maker(maker),
length(length) {}

   void displayInfo(Engine eg){
      std::cout << "Car maker: " << maker << ", Length: " <<
       length << std::endl << "Datos del motor: ";
      eg.displayEngine();
   }
};

int main() {
   Car car("Toyota", 4.5);
   Engine eng(300, true); // Un motor de 300 hp activado
   car.displayInfo(eng);
   // Imprime: Car maker: Toyota, Length: 4.5
  // Datos del motor: Power: 300, Engine On? Yes
   return 0;
}
```

Sobrecarga de Operadores con *friend*

Se sobrecarga el operador “<<“ de la clase ostream que pertenece a la C++ Standard Library (parte de iostream), para imprimir los datos del objeto Plane.

```
#include <iostream>
#include <string>
using namespace std;
class Plane {
    string model;
    int capacity;
    double wingspan;

public:
    Plane(string m, int c, double w) :
          model(m), capacity(c), wingspan(w) {}

    friend ostream& operator<<(ostream& os, const
    Plane& p); // Amistad con un método
};
```

```
ostream& operator<<(ostream& os, const Plane& p) {
     os << "Plane Model: " << p.model << "\n"
          << "Capacity: " << p.capacity << " passengers\n"
          << "Wingspan: " << p.wingspan << " meters";
    return os;
}

int main() {
    Plane plane("Boeing 747", 416, 68.4);
    // Usa el operador “<<“ para imprimir el objeto
    cout << plane;
    return 0;
}
```

Copiar Objetos – Shallow Copy

Cuando se necesita copiar un objeto, se puede hacer simplemente utilizando el operador "=" si el objeto sólo contiene tipos primitivos.

```
1.    myClass c1(3, 5);
2.    myClass c2;
3.    c2 = c1;
```

A este tipo de asociación se la denomina ***Shallow Copy***.

Una *shallow copy* solo funciona si no se tienen memoria alocada dinámicamente dentro del objeto. Esto se debe a que sólo copiará la dirección del puntero. Generando problemas de doble borrado o a *dangling pointers*.

Copiar Objetos – Deep Copy

Para copiar un objeto que posee punteros con memoria alocada dinámicamente se debe realizar una ***Deep Copy***. Esto dará como resultado un nuevo objeto, con data allocada en una posición distinta.

Una forma de hacer una copia es utilizando ***Copy Constructors***, los cuales permiten:

1. Crear una ***Shallow Copy***,
2. Crear una ***Deep Copy*** con raw pointers,
3. Crear una ***Deep Copy*** con Smart Pointers.

No obstante, estas no son las únicas formas de lograrlo.

Copy Constructor

Un copy constructor es un constructor especial en C++ que se usa para crear un nuevo objeto como copia de un objeto original.

Con este tipo de constructor, el objeto se copia haciendo:

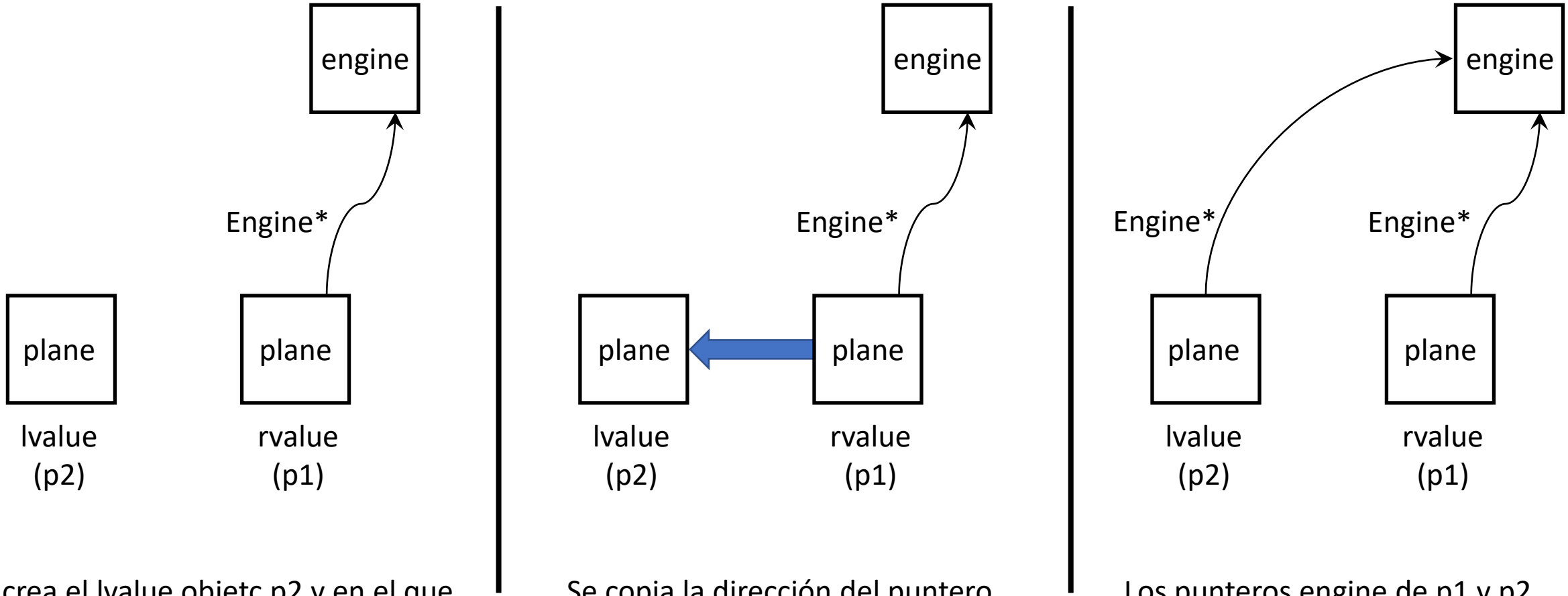
```
MyObject obj1(10);
MyObject obj2 = obj1;
```

En este caso, no es necesario sobrecargar el operador de asignación (=) para tener una copia. Se debe sobrecargar el constructor, de manera tal que permita aceptar el objeto a copiar, en este caso tipo MyObject, y copiar todos los atributos.

Mediante este método se pueden hacer shallow (no recomendable en este caso) o deep copies.

Copy Constructor – Shallow Copy

Se puede ver el problema en la siguiente secuencia de eventos:



Se crea el lvalue objetc p2 y en el que se hará la copia del rvalue object p1.

Se copia la dirección del puntero engine de p1 a p2.

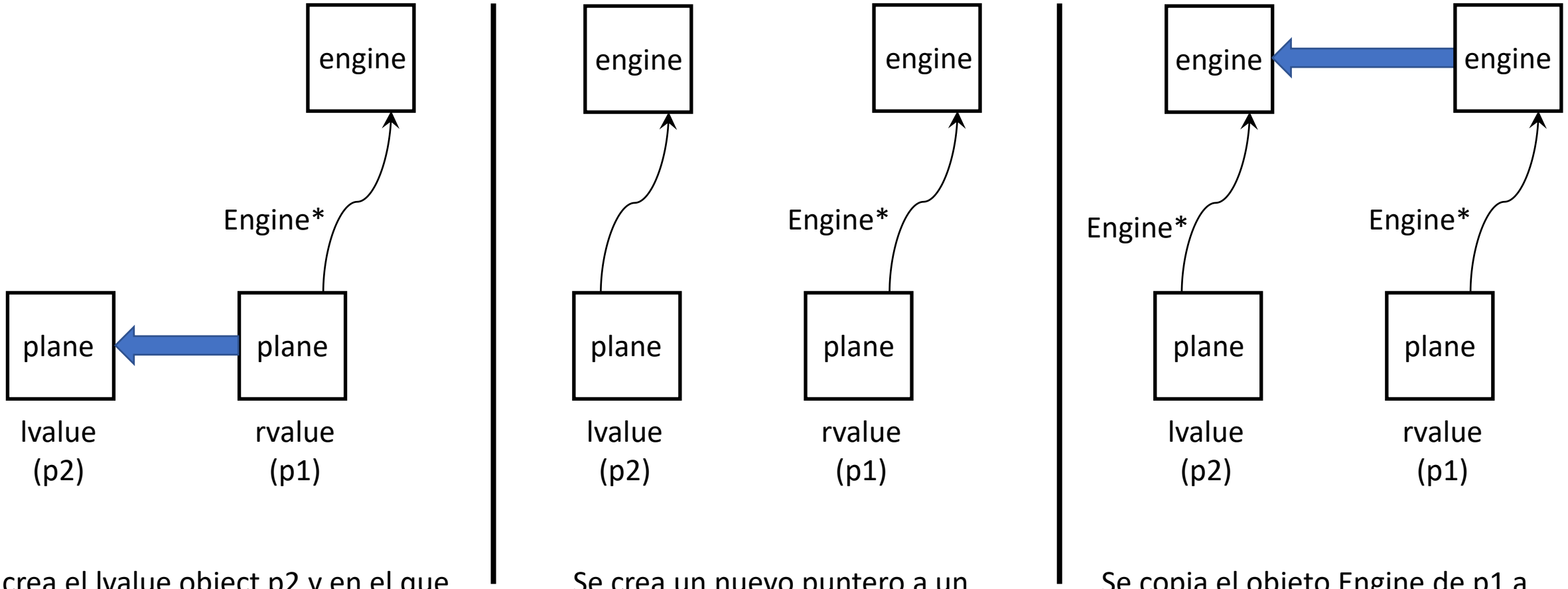
Los punteros engine de p1 y p2 apuntan a la misma dirección.

Copy Constructor – Ejemplo de Shallow Copy

```
class Engine {
public:
   Engine(int power) : power(power) {}
   int getPower() const {return power; }
private:
   int power;
};

class Plane {
public:
    //  Constructor tipo Shallow Copy
   Plane(int power) : engine(new Engine(power)) {}
   // Constructor que copia el puntero
   Plane(const Plane& rht) : engine(rht.engine) {}

   void display() const {
      std::cout << "Engine Power: " << engine->getPower()
      << "\n";
   }
   ~Plane() {
      delete engine;
// Se borraran p1 y p2. Al borrar p2 se producirá un
error porque el puntero a Engine se borro al borrar
p1.
   }
private:
   Engine* engine;
};

int main() {
   Plane p1(200);
   Plane p2 = p1; // Shallow Copy

   p1.display(); // Ambos comparten el
   p2.display(); // mismo puntero a Engine

   return 0;
}
```

Copy Constructor – Deep Copy

Se puede ver el problema en la siguiente secuencia de eventos:



Se crea el lvalue object p2 y en el que se hará la copia del rvalue object p1.

Se crea un nuevo puntero a un objeto Engine para p2.

Se copia el objeto Engine de p1 a la posición de memoria de p2.

Copy Constructor – Ejemplo de Deep Copy

```
class Engine {
public:
   Engine(int power) : power(power) {}
   int getPower() const {return power;}
private:
   int power;
};

class Plane {
public:
   Plane(int power) : engine(new Engine(power)) {}
   // Constructor tipo Deep Copy
   Plane(const Plane& rht) : engine(new Engine(
                                    (rht.engine)->getPower()  ) ){}
   // En la linea anterior se creo un nuevo puntero a Engine
   void display() const {
      std::cout << "Engine Power: " << engine->getPower()
                << "\n";
    }
      ~Plane() { delete engine; }
   private:
      Engine* engine;
};

int main() {
   Plane p1(200);
   Plane p2 = p1; // Al Deep Copy Constructor

   p1.display();
   p2.display(); // Los objetos Engine son
independientes

   return 0;
}
// Como los objetos Engine son distintos
objetos en distintas posiciones de memoria,
son independientes.
```

Copiar Objetos – Deep Copy

```
#include <iostream>
#include <memory> // Para los smart pointers

class Engine {
public:
    Engine(int power) : power(power) {}
    int getPower() const { return power; }
private:
    int power;
};

class Plane {
public:
    Plane(int serial, int length, int power)
        : iSerialNumber(serial), iLength(length),
        engine(std::make_unique<Engine>(power)) {}
    // Deep Copy con Copy Constructor usando shared_ptr
    Plane(const Plane& rht) // Deep copy de engine
        : iSerialNumber(rht.iSerialNumber),
          iLength(rht.iLength),
          engine(std::make_unique<Engine>(
                                rht.engine->getPower() )  ) {}
    void display() const {
        std::cout << "Serial Number: " << iSerialNumber << "\n";
        std::cout << "Length: " << iLength << "\n";
        std::cout << "Engine Power: " << engine->getPower() << "\n";
    }
private:
    int iSerialNumber;
    int iLength;
    std::unique_ptr<Engine> engine; // Shared pointer para Engine
};

int main() {
    Plane p1(1234, 50, 200);
    Plane p2 = p1; // Deep copy usando copy constructor

    std::cout << "Plane 1 Details:\n";
    p1.display();
    std::cout << "\nPlane 2 Details (after deep copy):\n";
    p2.display();

    return 0;
} // No hay necesidad de escribir los destroyers
```

Universidad de SanAndrés

Preguntas?