UNIDAD 7

Sobrecarga de Operadores y Funciones.

1.- **Introducción.**

El desarrollo de lenguajes de programación es el resultado de ir incorporando nuevas posibilidades, que tal vez a primera vista sean cuestionadas.

Tal vez los primeros programadores de lenguaje máquina, vieron en el lenguaje ensamblador una solución demasiado complicada, para resolver un problema de programación. Un tiempo después de nuevo se repite la misma situación con los programadores de Lenguaje Ensamblador, al enfrentarse a los interpretes y compiladores, pensaban que sus características arbitrarias no merecian la pena de ser consideradas. Así estos programadores pensaban que los conceptos de estructura de datos, funciones y compilación por separado, eran innecesarios y confusos, sin embargo al empezar a utilizar tales características las fueron considerando indispensables para cualquier lenguaje de programación avanzado.

En el caso de C++, algunos programadores piensan que el hecho de cambiar el significado de un operador resulta confuso e innecesario. Como se vio en el capitulo de referencias; la sobrecarga de un operador establece la necesidad de la incorporación de las referencias en C++, si se eliminara el concepto de sobrecarga, también se podría eliminar la referencia y con ello la complejidad y confusión.

Esta consideración, desde el punto de vista de un lenguaje procedural, probablemente sería mejor, sin embargo, C++ se orienta en otra dirección. Aunque es posible crear y mantener grandes proyectos y bibliotecas con lenguajes como C, realmente no tienen las suficientes características para soportar esta clase de actividad, son diseñados para resolver un problema de programación a la vez. Una vez resuelto el problema, es posible realizar pequeños cambios, pero a largo plazo, su mantenimiento se vuelve demasiado complicado.

En lugar de preguntar "¿Cómo resolver el problema de escribir este programa de computadora?".

C++, pregunta:

* ¿Qué hacemos para resolver el problema de programación y de actividades relacionadas con la programación?.
* ¿Qué vamos a hacer para soportar la creación de programas baratos en su mantenimiento y faciles de modificar?.
* ¿Cómo vamos a manipular el código anteriormente escrito.

Estas preguntas cambian la dirección de diseño.

La aportación de C++ y otros lenguajes orientados a objetos es la **extensiblilidad**. En vez de ir creando un programa cada vez más complejo a medida que los problemas planteados se vuelven más complicados. La **extensibilidad** nos

permite cambiar el modelo del lenguaje ajustandolo a nuestro problema. Una vez hecho ésto, la solución se puede expresar de una manera sencilla y clara, permitiendo que el programa sea fácil de entender y modificar.

La **extensibilidad** ha recibido fuertes criticas, debido a que lenguajes como FORTH, la han usado sin ninguna restricción, el hecho de no poner limites a la **extensibilidad** lleva como consecuencia que los programas solo sean entendidos por sus autores.

En el caso del lenguaje C++, la diferencia es que los nuevos elementos del lenguaje (las clases) se apegan a un conjunto de reglas estrictas, determinadas por el compilador de C++. Las clases deben adaptarse a la sintaxis del lenguaje anterior (C).

Todos los tipos de datos, cuentan con ciertas operaciones que pueden realizarse sobre ellos. Las operaciones permitidas para un tipo de dato definido por el usuario, pueden ser funciones miembro ordinarias. Un operador, sin embargo, es otra forma de expresar funcionalidad, y para algunos tipos de datos es más natural el uso de un operador que el de una función miembro. Para integrar un nuevo tipo de dato definido por el usuario al lenguaje, se deberá contar con la habilidad de dar a los operadores un significado especial para este nuevo tipo de dato.

2.- **La sintaxis del operador sobrecargado.**

Para sobrecargar un operador (de la misma forma que para una función, la sobrecarga significa "Darle un significado adicional") es necesario definir una función para que el compilador la llame cuando el operador es utilizado con los tipos de datos apropiados. Cada vez que el compilador ve estos tipos de datos utilizados con el operador, llama a la función. Se pueden tener multiples funciones para sobrecargar un operador, la condición es que tengan diferentes argumentos (igual que en el caso de las funciones sobrecargadas), de tal manera que el compilador pueda diferenciarlos.

La sintaxis de la definición de la función para el operador sobrecargado es diferente que el de una función normal sobrecargada.

El nombre de la función es la palabra reservada, **operator**, seguida por el operador utilizado, seguida por la lista de argumentos y el cuerpo de la función.

Por ejemplo si utilizamos el símbolo @ para representar el operador utilizado en particular, la sintaxis será:

Para una función friend.

```
tipo operator @(Lista de argumentos)
{
     cuerpo de la función
}
```

Para una función miembro de una clase

```
tipo nombre_class::operator @(Lista de argumentos)
{
     cuerpo de la función
}
```

Una función friend, tiene un argumento para un operador unario y 2 argumentos para un operador binario.

Una función miembro no lleva argumentos para un operador unario y lleva un argumento para un operador binario.

La razón de lo anterior es que una función miembro es automaticamente referenciada a la variable, del objeto para el cual fué llamada.

2.a) Operadores disponibles para ser sobrecargados.

La principal restricción al sobrecargar un operador es que su sintaxis y precesencia se concerven como originalmente fueron definidos. También, la sobrecarga es válida solamente dentro del contexto de la clase en la cual se definió la sobrecarga. La siguiente tabla nos muestra los operadores que pueden ser sobrecargados.

+	-	*	/	=	<	>	+=	-=	*=	/=	<<	<<
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	~=	|=	&&	||	%=	[]	()	->	new	delete	

3.- **Ejemplos de sobrecarga de operadores.**

En esta sección se presentan dos ejemplos de sobrecarga de operadores.

Ejemplo 1:

La situación más común en la cual un operador sobrecargado es útil, es cuando manejamos un objeto orientado al campo matemático.

Ejemplo 2:

Uso de un operador en un ambiente no matemático.

3.a) Una clase matemática.

Esta clase muestra ejemplos de sobrecarga de muchos de los operadores. Creamos una clase llamada **punto** la cual contiene puntos en el espacio, todas las operaciones son definidas para **punto**, de tal forma de poder estudiar la sobrecarga de la mayoria de los tipos de operadores.

```
// Archivo PLCP48.H
// Declaración de una clase para representar
// un punto en el espacio.
#ifndef PUNTO_H_
#define PUNTO_H_

class punto {
protected:
    float x, y;
public:
    punto(float xx = 0, float yy = 0) {
    x = xx;
    y = yy;
    }
    punto operator=(punto);          // Asignación.
    float magnitud();                // Vector equivalente.
    float angulo();
    punto operator+(punto);          // Suma 2 puntos.
    punto operator+();               // mas monario.
    punto operator-(punto);          // Resta 2 puntos.
    punto operator-();               // menos monario.
    punto operator*(float);          // Multiplica un punto
                                     // por un escalar.
    punto operator/(float);          // Divide un punto por
                                     // un escalar.
    punto operator%(punto);          // modulo.
    punto operator+=(punto);         // mismo operador con '='
    punto operator-=(punto);
    punto operator*=(float);
    punto operator/=(float);
    punto operator%=(punto);
    punto operator++();              // incremento (prefijo).
    punto operator++(int);           // incremento (posfijo).
    punto operator--();              // decremento (prefijo).
    punto operator--(int);           // decremento (posfijo).
    punto operator[](float);         // Rota el vector.
    punto operator<<(float);         // Gira el vector con
                                      // referencia
                                     // a la coordenada x.
    punto operator>>(float);         // Gira el vector con
                                      // referencia
                                     // a la coordenada y.
    punto operator<<=(float);        // Gira y asigna.
```

```
    punto operator>>=(float);
    int operator<(punto);           // Comparación relacional
    int operator>(punto);           // Regresa 1 si true 0 si
                                    // false
    int operator<=(punto);
    int operator>=(punto);
    int operator==(punto);
    int operator!=(punto);
    int operator!();                // 1, si un punto es cero
    int operator&&(punto);          // 0, si ambos son cero
    int operator||(punto);          // 0, solo si ambos son cero
    float operator&(punto);         // Producto cruzado
                                    // solo magnitud.
    float operator|(punto);         // Producto punto.
    float operator^(punto);         // Angulo entre dos vectores.
    punto operator&=(punto);        // Multiplica punto por cp
                                    // magnitud.
    punto operator|=(punto);        // Multiplica punto por dp
                                    // magnitud.
    punto operator^=(punto);        // Multiplica punto por
                                    //angulo.
    void imprime(char *msg = "");
};
#endif                              // PUNTO_H_
```

Cada operador que tiene **punto** como argumento o regresa **punto** pasa o regresa el objeto por valor.

A continuación se muestran los **metodos** para la clase **punto**, los cuales son puestos en el archivo PLCP458.CPP

```
// PLCP48A.CPP Método para la clase punto
// Representar un punto en el espacio.
// Note que no todas estas definiciones tienen sentido.
#include "plcp48.h"
#include <math.h>
#include <iostream.h>

// Usado para determinar cero

const float tiny = 0.0001;

punto punto::operator=(punto rv)
{
    x = rv.x;
    y = rv.y;
    return *this;       // Regresa una copia de este objeto.
```

```
}

float punto::magnitud()
{
    return sqrt(x*x + y*y);
}

float punto::angulo()
{
    return atan2(y, x);
}

punto punto::operator+(punto p)
{
    return punto(x + p.x, y + p.y);
}

punto punto::operator+()
{
    return *this;       // Operador unario, no hace nada.
}

punto punto::operator-(punto p)
{
    return punto(x - p.x, y - p.y);
}

punto punto::operator-()
{
    return punto(-x, -y);
}

punto punto::operator*(float f)
{
    return punto(x*f, y*f);
}

punto punto::operator/(float f)
{
    return punto(x/f, y/f);
}

punto punto::operator%(punto p)
{
    return punto(fmod(x, p.x), fmod(y, p.y));
}

punto punto::operator+=(punto p)
{
```

```
    x += p.x;
    y += p.y;
    return *this;
}

punto punto::operator-=(punto p)
{
    x -= p.x;
    y -= p.y;
    return *this;
}

punto punto::operator*=(float f)
{
    x *= f;
    y *= f;
    return *this;
}

punto punto::operator/=(float f)
{
    x /= f;
    y /= f;
    return *this;
}

punto punto::operator%=(punto p)
{
    x = floor(x/p.x);
    y = floor(y/p.y);
    return *this;
}


// Forma de prefijo.
punto punto::operator++()
{
    x += 1.0;
    y += 1.0;
    return *this;
}

// Forma posfija.
punto punto::operator++(int)
{
    return operator++();
}
```

```
// Forma de prefijo.
punto punto::operator--()
{
    x -= 1.0;
    y -= 1.0;
    return *this;
}


// Forma posfija.
punto punto::operator--(int)
{
    return operator--();
}

punto punto::operator[](float f)
{
    float nx = magnitud() * cos(angulo()*f);
    float ny = magnitud() * sin(angulo()*f);
    return punto(nx, ny);
}

punto punto::operator<<(float f)
{
    return punto(x+f, y);
}

punto punto::operator>>(float f)
{
    return punto(x, y+f);
}

punto punto::operator<<=(float f)
{
    x += f;
    return *this;
}

punto punto::operator>>=(float f)
{
    y += f;
    return *this;
}

int punto::operator<(punto p)
{
    if(x < p.x && y < p.y)
        return 1;
    return 0;
```

```
}

int punto::operator>(punto p)
{
    if(x > p.x && y > p.y)
        return 1;
    return 0;
}


int punto::operator<=(punto p)
{
    if(x <= p.x && y <= p.y)
        return 1;
    return 0;
}

int punto::operator>=(punto p)
{
    if(x >= p.x && y >= p.y)
        return 1;
    return 0;
}


int punto::operator==(punto p)
{
    if(x == p.x && y == p.y)
        return 1;
    return 0;
}


int punto::operator!=(punto p)
{
    if(x != p.x && y != p.y)
        return 1;
    return 0;
}

int punto::operator!()
{
    if(fabs(x) < tiny && fabs(y) < tiny)
        return 1;
    return 0;
}

int punto::operator&&(punto p)
{
```

```
    if(fabs(x) < tiny && fabs(y) < tiny)
        return 0;
    if(fabs(p.x) < tiny && fabs(p.y) < tiny)
        return 0;
    return 1;
}


float punto::operator&(punto p)
{
    return magnitud() * p.magnitud() *
            sin(punto::operator^(p));
}


float punto::operator|(punto p)
{
    return magnitud() * p.magnitud() *
            cos(punto::operator^(p));
}

float punto::operator^(punto p)
{
    return fabs(angulo() - p.angulo());
}

punto punto::operator&=(punto p)
{
    float cruz = operator&(p);
    x *= cruz;
    y *= cruz;
    return *this;
}

punto punto::operator|=(punto p)
{
    float dot = operator|(p);
    x *= dot;
    y *= dot;
    return *this;
}


punto punto::operator^=(punto p)
{
    float arc = operator^(p);
    x *= arc;
    y *= arc;
    return *this;
```

```
}

void punto::imprime(char *msg)
{
    if(*msg) cout << msg  << " : ";
    cout << "x = " << x << ", y = " << y << endl;
}
```

OBSERVACIONES.

Muchos de los metodos usan funciones de la librería matemática math.h. Así la función **floor**. Es una función de redondeo.

Sintaxis:

```
double floor(double x)   Redondea hacia abajo.
double ceil(double x)    Redondea hacia arriba.
```

Recuerde que la sobrecarga de un operador representa una decisión importante de diseño la cual puede facilitar o dificultar el desarrollo.

Cada operador mostrado es sobrecargado solamente una vez, pero recuerde que puede ser sobrecargado más de una vez, mientras se usen diferentes argumentos con cada sobrecarga.

Como ya habiamos mencionado, todos los metodos de la clase **punto** pasan y regresan objetos o variables por valor, en este caso particular, resulta eficiente por el tamaño de la clase.

Note que en la forma posfija del **operator++**, el valor regresado es el resultado de la llamada a la función prefija **operator++**, la cual regresa también un tipo **punto**.

Es este caso solo se ha definido la sobrecarga del operador, (++), no se debe esperar que funcione como de constumbre, lo cual se puede ver en el siguiente ejemplo.

La forma de expresar la llamada a un operador, por ejemplo:

```
cos(punto::operator^(p)) es equivalente a:  cos(*this^p)
operator&(p)             es equivalente a:  *this & p
operator|(p)             es equivalente a:  *this | p
```

En el primer caso no es necesario utilizar el operador de ámbito (::), ya que nos estamos refiriendo a la misma clase punto.

```
// Programa 48-1, Uso del operador ++
#include "plcp48.h"
```

```
void main()
{
    punto A(1.0, 2.0), B(3.0, 4.0), C;

    A.imprime("A");
    B.imprime("B");
    C = B + A++;
    C.imprime("C");
    A.imprime("A");
    C = B + ++A;
    C.imprime("C");
    A.imprime("A");
}
```

OBSERVACIONES.

En el primer caso C = B + A++;, la expresión se evalúa de la siguiente manera:

1.- Se llama a la función posfija del operator++
2.- Se llama a la función prefija del operator++
3.- Se llama a la función del operator+
4. Se llama a la función del operator=

En el segundo caso C = B + ++A;, la expresión se evalúade la siguiente manera:

1.- Se llama a la función prefija del operator++
2.- Se llama a la función del operator+
3. Se llama a la función del operator=

La Salida nos resulta:

```
A : x = 1, y = 2
B : x = 3, y = 4

C : x = 5, y = 7
A : x = 2, y = 3

C : x = 6, y = 8
A : x = 3, y = 4
```

NOTA: Los espacios no fueron dados por el programa.

3.a.1) Creación de objetos temporales.

Algunos metodos de la clase **punto** usan una forma no común de retorno en la sentencia return, por ejemplo en:

```
punto punto::operator[](float f)
{
    float nx = magnitud() * cos(angulo()*f);
    float ny = magnitud() * sin(angulo()*f);
    return punto(nx, ny);
}
```

La sentencia de retorno es: **return punto(nx, ny)**;

¿Qué significa ésto, se esta empleando el nombre de una clase, con una lista de argumentos. Cuando el compilador se encuentra esta forma, la trata como un constructor, él cual es llamado para crear un **objeto temporal**. Es un objeto sin nombre y normalmente con un tiempo de vida breve. En el ejemplo, se crea espacio de memoria dentro de la función **operator[]** para almacenar un objeto del tipo **punto** y entonces se llama al constructor punto::punto(float, float) para inicializar el objeto, finalmente el objeto temporal es copiado fuera de la función por medio de la sentencia return y el objeto temporal sale fuera de ámbito.

Pueden ser creados objetos temporales en cualquier lugar donde sea posible usar un objeto no temporal. Así por ejemplo si vamos a llamar la función uno(punto), la cual recibe como argumento un objeto del tipo **punto**, normalmente se maneja el siguiente método:

```
punto A(1.2, 2.4);
uno(A);
```

Podemos crear un objeto temporal para el propósito específico de pasarlo como argumento, se haría de la siguiente forma:

```
uno(punto(1.2, 2.4));
```

Mediante el uso de objetos temporales, el código puede ser más breve, más eficiente y fácil de leer.

3.a.2) Distinguiendo entre prefijo y posfijo para los operadores ++ y --

En la declaración y en la definición para los operadores **operator++()** y **operator--()**, el compilador distingue entre el posfijo y el prefijo de estos operadores por la llamada de la función sin argumentos y con un argumento **int** respectivamente. Si para

un objeto tipo punto **p** se escribe **++p**, el compilador va a llamar a **punto::operator++()** pero si escribe **p++** el compilador llama a **punto::operator++(int)**. De esta manera se pueden escribir dos funciones diferentes, con firma distinta y tener un efecto diferente para los operadores de pre- y posfijo. Considere el hecho de que algunos compiladores anteriores no distinguen entre el prefijo y posfijo.

Note que el argumento **int** no tiene ningún identificador, por lo tanto no se usa argumento en la función, el objeto de usar **int** es el de que el compilador distinga entre ambas funciones. Como no hay identificador, el compilador supone que una variable va a ser creada pero no usada.

3.a.3) Operadores de evaluaciones rápidas.

En C y C++ el operador sin sobrecarga && (and lógico) y || (or lógico) tienen la característica de evaluación rápida. Si al evaluar una expresión resulta vardadera o falsa y dicha expresión contiene cualquiera de los operadores && ó ||, el resultado puede ser determinado antes de terminar la evaluación de la expresión, no siendo evaluada toda la expresión. Por ejemplo:

```
if(a && b)
     // ....
if(a || b)
     //....
```

Si en la primera sentencia if, **a** es evaluada como CERO (false) entonces ya no es evaluada la **b**, ya que de antemano se sabe que el resultado va a ser falso.

Si en la segunda sentencia if, **a** es evaluada como UNO (true), se sabe de antemano que el resultado va a ser verdadero y ya no es evaluada la **b**.

Cuando se sobrecargan estos operadores como en el ejemplo **operator&&()** u **operator||()**, no se realiza una evaluación rápida en las expresiones que usan estos operadores sobrecargados, "siempre se evalúa toda la expresión".

3.a.4) Regresando el objeto con this.

En algunas de las funciones de la clase **punto** se esta regresando el objeto en la sentencia return usando ***this**.

La palabra reservada **this** es única en C++, contiene la dirección inicial del objeto referenciado por la función.

```
punto punto::operator>>=(float f)
{
    y += f;
    return *this;
}
```

En este caso la expresión se escribe:

Si declaramos un objeto;
```
punto A;
float f;
A>>=f;
```

En este caso el objeto referenciado es **A** y el nuevo valor se regresa a **A**, de la forma A = A + f;

Veamos lo dicho en el siguiente programa.

```
// Programa 48-2, uso de la palabra this.
#include "plcp48.h"
#include <iostream.h>

void main()
{
    punto A(3.1, 6.0);
    float f = 3.6;

    A.imprime("A");
    cout << "Valor de f= " << f << endl;
    A>>=f;
    A.imprime("Resultado de: A>>=f");
}
```

OBSERVACIONES.

La operación realizada sería; si los datos miembro fueran públicos: A.y = A.y + f, sin embargo la función regresa todo el objeto, como ya se analizó arriba.

La corrida nos resulta:

```
A : x = 3.1, y = 6
Valor de f= 3.6
Resultado de: A>>=f : x = 3.1, y = 9.6
```

El valor afectado es solo el de la **y**.

3.a.5) Un ejemplo del uso del operador sobrecargado [].

```
// Programa 48-3, uso del operador []
#include "plcp48.h"
#include <iostream.h>

void main()
{
```

```
    punto A(2.0, 4.0), B;
    float f = 10.0;

    A.imprime("A");
    B = A[f];
    B.imprime("B = A[f]");
}
```

OBSERVACIONES.

Al correr el programa se obtiene:

A : x = 2, y = 4
B = A[f] : x = 0.339167, y = -4.459256

La operación realizada es la siguiente:

A = $(2.0^2 + 4.0^2)^{½}$ = 4.472135955
nx = 4.472135955*cos(63.4349*10) = 4.472135955*0.07584 = 0.339167
ny = 4.472135955*sin(63.4349*10) = 4.472135955*(-9971) =-4.459256

Con lo que verificamos que las operaciones fueron realizadas de acuerdo a lo señalado en la función **operator[]() .**

3.a.6) Analicemos un programa más general el cual usa también la clase **punto**.

```
// Programa PLCP48.CPP
// Algunas pruebas para la clase punto
#include "plcp48.h"
#include <iostream.h>

void main()
{
    punto A, B(1.1, 2.2), C(3.3, 4.4), D(5.5, 6.6);

    B.imprime("B");
    C.imprime("C");
    D.imprime("D");
    A = B + C - D;
    A.imprime("A = B + C - D");
    A += (B << 8.2) + (D >> 4.1);
    A.imprime("A += (B << 8.2) + (D >> 4.1)");
    cout << "Magnitud de A = " << A.magnitud()
         << ", angulo() de A = " << A.angulo() << endl;
    cout << "Magnitud de C = " << C.magnitud()
         << ", angulo() de C = " << C.angulo() << endl;
    cout << "Magnitud de D = " << D.magnitud()
```

```
        << ", angulo() de D = " << D.angulo() << endl;
    cout << "Angulo C ^ D = " << (C ^ D) << endl;
}
```

OBSERVACIONES.

La corrida nos rsulta:

```
B : x = 1.1, y = 2.2
C : x = 3.3, y = 4.4
D : x = 5.5, y = 6.6
A = B + C - D : x = -1.1, y = 4.768372e-07
A += (B << 8.2) + (D >> 4.1) : x = 13.700001, y = 12.9
Magnitud de A = 18.817545, angulo() de A = 0.755332
Magnitud de C = 5.5, angulo() de C = 0.927295
Magnitud de D = 8.591274, angulo() de D = 0.876058
Angulo C ^ D = 0.051237
```

Primero mostramos el valor de cada objeto; A, B, C, D.
Luego realizamos una operación aritmética:

A = B + C - D;

		x	y
B		1.1	2.2
C	+	3.3	4.4
D	-	5.5	6.6
		---	---
A	=	-1.1	0.0

La siguiente operación es: A += (B << 8.2) + (D >> 4.1)

		x	y		
D		5.5	6.6		
	>>		4.1		
		---	---		
D		5.5	10.7	5.5	10.7
B		1.1	2.2		
	<<	8.2			
		---	---		
B		9.3	2.2	9.3	2.2
A	+			-1.1	0.0
				---	---
A	=			13.7	12.9

La siguiente operacón es: magnitud de A

$A = (13.7^2 + 12.9^2)^{1/2} = 18.817545.$

El calculo del ángulo se realiza como sigue:

```
arctang(12.9/13.7) = (43.27*pi)/180 = 0.755332
```

Se deja como ejercicio el calculo de la operación: C ^ D
él cual es el absoluto de la diferencia de los ángulos de los objetos C - D

4.- **Sobrecarga del operador esotérico.**

La clase punto muestra la sobrecarga de los operadores más comunes, C++ no para aquí, es posible sobrecargar los operadores de dirección [&] y [*], los que en un principio no se pensó redefinir, también es posible sobrecargar los paréntesis de llamada de función. La clase mostrada en el siguiente ejemplo tiene el elemento **indice** declarado antes del arreglo **vv[]**, si se intenta accesar el arreglo de forma directa por medio de **this** se obtendrá **indice** es su lugar, ésto es una mala idea, ya que el acceso a un elemento va a depender de la forma de implementar la clase. Mediante el uso de los operadores **&** y ***** es posible accesar el arreglo **v[]** en vez del elemento **indice**.

Una razón para sobrecargar la llamada a una función, es que nos permite utilizar la sintaxis de un operador con multiples argumentos. El siguiente ejemplo nos muestra el uso de estos operadores sobrecargados.

* Primero se muestra la interface.

```
// Archivo PLCP49.H Sobrecarga de operadores esotéricos.
#ifndef VECTOR_H_
#define VECTOR_H_
const int tam = 20;

class vector {
    int indice;
    float vv[tam];
    void error(char *msg);
public:
    vector(float vi = 0.0);
    float &operator[](int indice);

    // Usa la "dirección del" operador para regresar
    // la dirección del arreglo.
    const float *operator&() { return vv; }
    float operator*() { return vv[0]; }
```

```
   // Se puede sobrecargar el operador de "Llamada a la función.
   // La función asigna en la posición el el valor f y regresa f.
    float operator()(int el, float f);
    void imprime(char *msg = "");
};
#endif      // VECTOR_H_
```

* A continuación se muestra la implementación de sus métodos.

```
// Programa PLC49.CPP  Metodos para la clase vector
#include <iostream.h>
#include <stdlib.h>
#include "plcp49.h"

void vector::error(char *msg)
{
    cout << "Error en vector: " << msg << endl;
    exit(1);
}

vector::vector(float vi)
{
    for(int i = 0; i < tam; i++)
        vv[i] = vi;
}

float &vector::operator[](int indice)
{
    if( indice < 0 || indice >= tam)
        error("Operador fuera de Rango");
    return vv[indice];
}

float vector::operator()(int el, float f)
{
    if(el < 0 || el >= tam)
        error("Operador fuera de Rango");
    vv[el] = f;
    return f;
}

void vector::imprime(char *msg)
{
    if(*msg) cout << msg << ":" << endl;
    for(int i = 0; i < tam; i++) {
        cout.precision(2);
        cout.width(7);
        cout << vv[i];
```

```
        if(i % 5 == 4)  // Se ejecuto LF
            cout << endl;
    }
    cout << endl << endl;
}
```

* Por último un ejemplo de su uso.

```
// Programa 49-1, Uso de la clase vector
#include <iostream.h>
#include "plcp49.h"

void main()
{
    vector V(1.1);

    V.imprime("V");
    for(int i = 0; i < tam; i++)
        V[i] = (float)(i*1.5);
    V[0] = 33.119;
    V.imprime("V después V[i] = (float)(i*1.5); V[0] = 33.119");
    const float *val_ini = &V;
    cout << "*val_ini = " << *val_ini << endl;
    V(4, 399.99);
    V.imprime("V después V(4, 399.99)");
    cout << "*V = " << *V << endl;
}
```

OBSERVACIONES.

La salida del programa nos resulta:

```
V:

      1.1    1.1    1.1    1.1    1.1
      1.1    1.1    1.1    1.1    1.1
      1.1    1.1    1.1    1.1    1.1
      1.1    1.1    1.1    1.1    1.1


V después V[i] = (float)(i*1.5); V[0] = 33.119:
  33.12    1.5      3    4.5      6
    7.5      9   10.5     12   13.5
     15   16.5     18   19.5     21
   22.5     24   25.5     27   28.5
```

```
*val_ini = 33.12
V después V(4, 399.99):
  33.12    1.5      3    4.5 399.99
    7.5      9   10.5     12   13.5
     15   16.5     18   19.5     21
   22.5     24   25.5     27   28.5


*V = 33.12
```

* Primero se muestra el arreglo inicializado por el constructor al valor 1.1, lo cual se realizó en su declaración.

* En el siguiente bloque, por medio de la función miembro:

```
float &vector::operator[](int indice);
```

Cambiamos el contenido del arreglo multiplicando el indice del arreglo por 1.5, note el uso del cast, ya que una constante real, por default es double. Se obtiene el segundo arreglo, al cual después se le asigna al elemento CERO el valor de 33.199, redondeado por la precisión especificada de (2). Es la forma en que es mostrada la salida del arreglo.

* En seguida definimos una variable puntero a un tipo float, (val_ini) la cual es declarada como const para no ir a alterar el dato almacenado en dicha dirección, y seguir manteniendo el control de los datos miembro, los cuales solo pueden ser accesados por las funciones miembro.

Sobrecargando el ***operator&()**, nos regresa la dirección del vector **vv**, como un puntero de base float, de acuerdo a la siguiente función:

```
const float *operator&() { return vv; }
```

4.a) Sobrecarga del operador []

Sobrecargando el **operator[]()**, nos regresa el valor del elemento referenciado por **indice**, usamos la función:

```
float &vector::operator[](int indice);
```

```
// Programa 49-2, Uso de la clase vector

#include <iostream.h>
#include "plcp49.h"
```

```
void main()
{
    vector X;

    for(int i = 0; i < tam; i++)
        X[i] = (float)(i*1.5);
    X.imprime("X[i] = (float)(i*1.5)");
    float val = X[13];
    cout << "val = " << val << endl;
}
```

La salida nos resulta:

```
X[i] = (float)(i*1.5):
       0    1.5      3    4.5      6
     7.5      9   10.5     12   13.5
      15   16.5     18   19.5     21
    22.5     24   25.5     27   28.5


val = 19.5
```

Estamos usando como argumento el índce 13 por lo tanto nos dá el contenido de dicho elemento.

* A continuación usamos la sobrecarga del operador (), mediante la función:

float vector::operator()(int el, float f);

Mediante esta función pasamos como argumentos; el índice y el nuevo valor del elemento del arreglo, el cual al mismo tiempo que es almacenado en dicho arreglo, es regresado por la función. Al imprimir de nuevo el arreglo vemos que realmente se encuentra en la posición [4]

El valor regresado no lo estamos utilizando, aunque es posible hacerlo, como vemos en la siguiente aplicación.

```
// Programa 49-3, Uso de la clase vector
#include <iostream.h>
#include "plcp49.h"

void main()
{
    vector C(2.3);

    C.imprime("C");
```

```
    float x = C(5, 20.5);
    C.imprime("Después de asignar vv[5] = 20.5");
    cout << "x = " << x << endl;
}
```

La salida nos resulta:

```
C:
    2.3    2.3    2.3    2.3    2.3
    2.3    2.3    2.3    2.3    2.3
    2.3    2.3    2.3    2.3    2.3
    2.3    2.3    2.3    2.3    2.3


Después de asignar vv[5] = 20.5:
    2.3    2.3    2.3    2.3    2.3
   20.5    2.3    2.3    2.3    2.3
    2.3    2.3    2.3    2.3    2.3
    2.3    2.3    2.3    2.3    2.3


x = 20.5
```

* Por último usamos la sobrecarga del operador [*], definido en la función:

```
float operator*() { return vv[0]; }
```

Al escribir la expresión *V, nos va a regresar el contenido del elemento con índice CERO del arreglo, como se puede ver en la definición de la función.

4.b) Sobrecarga del operador (,).

En una secuencia de expresiones separadas por comas, las expresiones son evaluadas (incluyendo el efecto lateral) de izquierda a derecha. Todos los valores resultantes de la evaluación de las expresiones son descartadas, excepto el valor final, él cual representa el resultado de la expresión completa. Por lo tanto todas las expresiones, excepto la última son usadas solamente por su efecto lateral.

Cuando la expresión al lado izquierdo de la coma (,) es un objeto, o produce un objeto, el operador (,) puede ser sobrecargado para producir un efecto lateral, apropiado al objeto. Veamos el siguiente ejemplo:

```
// Programa # 50B, Sobrecargando el operador (,)
#include <stdio.h>
#include <string.h>
```

```
class amiga {
     char a[40];
     int j;
public:
     amiga(int x = 0, char *c = "") : j(x) {
     strcpy(a, c);
     }
     void operator,(amiga c);
     void muestra();
};


void amiga::operator,(amiga c)
{
     strcpy(a, c.a);
     j = c.j;
}

void amiga::muestra()
{
     printf("Valor de j =  %d\n", j);
     printf("Cadena     =  %s\n", a);
}

void main()
{
     amiga uno(1, "Paty"), dos(2, "Angel");
     amiga tres(3, "Laura"), cuatro(4, "Velvet");
     amiga cinco(5);

     printf("Llamando a uno.muestra()     ");
     uno.muestra();
     printf("Llamando a dos.muestra()     ");
     dos.muestra();
     printf("Llamando a tres.muestra()    ");
     tres.muestra();
     printf("Llamando a cuatro.muestra()  ");
     cuatro.muestra();

     uno, dos, dos, tres, tres, cuatro;

     printf("Llamando a uno.muestra()     ");
     uno.muestra();
     printf("Llamando a dos.muestra()     ");
     dos.muestra();
     printf("Llamando a tres.muestra()    ");
     tres.muestra();
     printf("Llamando a cuatro.muestra()  ");
     cuatro.muestra();
```

```
}
```

OBSERVACIONES.

El operador (,) de acuerdo a la acción especificada por la sobrecarga va a copiar el contenido de los datos miembro del objeto a su derecha en los datos miembro del objeto a su izquierda, va a tomar de 2 en 2 objetos, (recuerde que es un operador binario), por lo tanto la (,) que esta entre el segundo y tercer objeto, es la estandar de separación de expresiones de "C". De acuerdo al programa, la acción será entre:

```
     uno, dos
     dos, tres
     tres, cuatro

void amiga::operator,(amiga c)
{
     strcpy(a, c.a);
     j = c.j;
}
```

La salida nos resulta:

```
Llamando a uno.muestra()     Valor de j =  1
Cadena     =  Paty
Llamando a dos.muestra()     Valor de j =  2
Cadena     =  Angel
Llamando a tres.muestra()    Valor de j =  3
Cadena     =  Laura
Llamando a cuatro.muestra()  Valor de j =  4
Cadena     =  Velvet

Llamando a uno.muestra()     Valor de j =  2
Cadena     =  Angel
Llamando a dos.muestra()     Valor de j =  3
Cadena     =  Laura
Llamando a tres.muestra()    Valor de j =  4
Cadena     =  Velvet
Llamando a cuatro.muestra()  Valor de j =  4
Cadena     =  Velvet
```

En el primer bloque se muestra la inicialización de los objetos, la cual se llevó a cabo en su declaración.

* El contenido del objeto dos es asignado al objeto uno.
* El contenido del objeto tres es asignado al objeto dos.

* El contenido del objeto cuatro es asignado al objeto tres.

Note que **operator,()** no regresa valor.

Podemos modificar un poco el programa haciendo ahora que el **operator,()**, nos regrese valor y utilizando este valor regresado, en operaciones posteriores. Veamos el ejemplo:

```
// Programa #50A, Sobrecargando el operador (,)
#include <stdio.h>
#include <string.h>


class amiga {
     char a[40];
     int j;
public:
     amiga(int x = 0, char *c = "") : j(x) {
     strcpy(a, c);
     }
     amiga operator,(amiga c);
     amiga operator=(amiga d);
     void muestra();
};


amiga amiga::operator,(amiga c)
{
     strcpy(a, c.a);
     j = c.j;
     return *this;
}

amiga amiga::operator=(amiga d)
{
     strcpy(a, d.a);
     return *this;
}

void amiga::muestra()
{
     printf("Valor de j =  %d\n", j);
     printf("Cadena     =  %s\n", a);
}


void main()
{
```

```
     amiga uno(1, "Paty"), dos(2, "Angel");
     amiga tres(3, "Laura"), cuatro(4, "Velvet");
     amiga cinco(5);

     printf("Llamando a uno.muestra()     ");
     uno.muestra();
     printf("Llamando a dos.muestra()     ");
     dos.muestra();
     printf("Llamando a tres.muestra()    ");
     tres.muestra();
     printf("Llamando a cuatro.muestra()  ");
     cuatro.muestra();
     printf("Llamando a cinco.muestra()   ");
     cinco.muestra();

     cinco = (uno, (dos, (tres, cuatro)));

     printf("Llamando a uno.muestra()     ");
     uno.muestra();
     printf("Llamando a dos.muestra()     ");
     dos.muestra();
     printf("Llamando a tres.muestra()    ");
     tres.muestra();
     printf("Llamando a cuatro.muestra()  ");
     cuatro.muestra();
     printf("Llamando a cinco.muestra()   ");
     cinco.muestra();
}
```

OBSERVACIONES.

Ahora el operator,(), se definió de la siguiente manera:

```
amiga amiga::operator,(amiga c)
{
     strcpy(a, c.a);
     j = c.j;
     return *this;
}
```

Ahora nos regresa el objeto a su izquierda, el cual podrá ser utilizado en una operación posterior.

También se definió la sobrecarga del **operator=()**, él cual a diferencia del **operator,()**, solo copia del objeto a su derecha, el valor de la cadena, en el objeto a su izquierda, y regresa el objeto a su izquierda.

```
amiga amiga::operator=(amiga d) {
     strcpy(a, d.a);
     return *this;
}
```

La salida nos resulta:

```
Llamando a uno.muestra()     Valor de j =  1
Cadena     =  Paty
Llamando a dos.muestra()     Valor de j =  2
Cadena     =  Angel
Llamando a tres.muestra()    Valor de j =  3
Cadena     =  Laura
Llamando a cuatro.muestra()  Valor de j =  4
Cadena     =  Velvet
Llamando a cinco.muestra()   Valor de j =  5
Cadena     =

Llamando a uno.muestra()     Valor de j =  4
Cadena     =  Velvet
Llamando a dos.muestra()     Valor de j =  4
Cadena     =  Velvet
Llamando a tres.muestra()    Valor de j =  4
Cadena     =  Velvet
Llamando a cuatro.muestra()  Valor de j =  4
Cadena     =  Velvet
Llamando a cinco.muestra()   Valor de j =  5
Cadena     =  Velvet
```

En el primer bloque se muestra como en el ejemplo anterior el valor de inicialización de los objetos, note que se agregó un objeto adicional, el cual solo se inicializa con el número (5).

De acuerdo a la operación:

```
cinco = (uno, (dos, (tres, cuatro)));
```

Mediante el **operator,()**, se asigna el valor de cuatro a tres, el valor regresado por la función de operator=(), es asignado al objeto dos, y el valor regresado de esta operación es asignado al objeto uno. Lo que se logra es propagar el valor de los datos miembro de cuatro, hacia la izquierda. Por último, el valor regresado de la última operación es asignado, (solo la cadena), al objeto cinco.

4.c) El operador unario -> (Apuntador smart)

NOTA: El trabajo normal del operador -> es la selección de un elemento de una clase o estructura..

Este operador no puede ser usado sobre miembros static, el empleo del apuntador smart, es un tanto intuitivo ya que se usa sobre un objeto de la clase y no como un apuntador a un objeto, que es el uso normal que se le dá. Lo que significa que el objeto es ideado para representar un apuntador, pero que haga algo diferente a un apuntador ordinario.

Si X es una clase la cual involucra el **operator->()** como una función miembro y **a** es un dato miembro de la clase u otra estructura, clase o union, entonces:

```
X xobj;

xobj -> a;
```

Lo cual es equivalente a:

```
(xobj.operator->())->a;
```

Lo que significa que el operador deberá regresar:

1) Un apuntador a un objeto de una clase conteniendo el elemento **a**.

2) Un objeto de otra clase, esta otra clase deberá contener una definición para el **operator->(),** en este caso el **operator->()** es llamado de nuevo para el nuevo objeto. Este proceso es repetido de forma recursiva hasta que el resultado es un puntero a un objeto de una clase conteniendo el elemento **a**.

Veamos un ejemplo sencillo.

```
// Programa # 51, El operador flecha ->
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class cadena {
     char *c;
public:
     cadena(char *C) {
          c = (char *)malloc(strlen(C) + 15);
          strcpy(c, C);
     }
     char *retorno() { return c; }
     ~cadena() {
      free(c);
```

```
     }
     cadena *operator->();
};

cadena *cadena::operator->()
{
          char x[15];
          strcpy(x, "Susy");
          strcat(c, x);
          return this;
}


void main()
{
     cadena X("Buenos dias, ");

     cout << X -> retorno();
}
```

OBSERVACIONES.

En este caso se llama al operador -> para realizar una copia de una cadena y un concatenación con la cadena definida como dato miembro. Veamos como se evalúa la expresión:

```
X.operator->() -> retorno();
```

El objeto X llama a la función representada por el operador smart (->). Dentro de la función ->, se realiza una copia de una cadena y su concatenación con el dato miembro.
La función -> regresa un puntero a un objeto del tipo cadena

```
          (X ->)
            ↓
regresa:
           (&X -> retorno() )  /* puntero a cadena */
regresa:              ↓
                      a  /* un puntero a char */
```

Note como se usa de nuevo el operador smart para llamar a la función miembro retorno().

A continuación se presenta otro ejemplo en el cual se usa de nuevo la propiedad de de-referenciación recursiva del apuntador smart.

```
// Programa # 51A, El operador flecha ->

#include <iostream.h>
#include <string.h>
#include <stdlib.h>


struct cadena {
     char *c;
     cadena(char *C) {
          c = (char *)malloc(strlen(C) + 15);
          strcpy(c, C);
     }
     ~cadena() {
      free(c);
     }
};

class punt_cad {
     cadena *m;
public:
     punt_cad(cadena& Y) : m(&Y) {}
     cadena *operator->();
};

cadena *punt_cad::operator->()
{
          char x[] = " :Saludos";
          strcat(m -> c, x);
          return m;
}


void main()
{
     punt_cad X(cadena("Hola muchachas"));

     cout << X -> c;
}
```

OBSERVACIONES.

Como se vió anteriormente, el operador (->) deberá regresar un apuntador a una clase, estructura o union, en donde se definió un dato miembro, en este caso del tipo puntero a char, para el cual en el constructor de la estructura cadena se pide memoria al heap, y en el destructor se libera.

El operador apuntador smart, lo que hace es agregar una cadena al elemento **c**, y mediante el objeto **X** se hace referencia directamente al elemento **c** de acuerdo a la expresión:

```
(X.operator->())->c
```

Analicemos esta expresión.

```
          (X -> )
regresa:    ↙
          m  -> c   /* Es un puntero a un objeto cadena */
regresa:    ↓
          c         /* Un puntero a char */
```

Vea que también se llama de manera recursiva el operador smart.

NOTA: El **operator->()** debe regresar un puntero a una clase, estructura o union sino, nos genera un error.

La clase **punt_cad** contiene un apuntador a la clase **cadena**, la función **operator->()**, regresa un puntero a la clase **cadena**, regresando **m** y no **c**, ésto lo sabe el compilador y continua hasta encontrar **c** desde **m**, la cual es un apuntador a char, miembro de la clase **cadena**. Note que esta acción recursiva puede continuar por varios niveles.