4.d) Multiples dimensiones del operador []

Una de las preguntas más frecuentes acerca de la sobrecarga de operadores, es; ¿Si es posible hacer trabajar el **operator[]** con multiples dimensiones?. Hay situaciones donde ésto nos brinda una ayuda en el modelo a diseñar.

Para entender el trabajo de una expresión con [][], considere la forma en que el compilador evalua la siguiente expresión.

```
A[i][j] = 1;
```

La interpretación de la estructura de esta expresión, no es como la de un arreglo bidimensional, se interpreta de la siguiente forma:

Primero se llama al operador sobrecargado A[i], él cual pertenece a la clase X, la función deberá de regresar una referencia a un objeto de la clase X, mediante el cual llamamos al siguiente operador sobrecargado [], el cual pertenece a una clase Y, la función deberá de regresar una referencia, una referencia representa una dirección, en cuya localidad almacenamos el (1).

La evaluación de la expresión se puede considerar también de la forma: **A.operator[](i).operator[](j)**, donde primero se evalúa el operador con i y luego el de j.

Note la importancia de las referencias en esta situación. Para manejar la sobrecarga de los operadores [][] de la manera acostumbrada en la indexación de arreglos, se deberán manejar referencias, de otra forma se deberán regresar los objetos por valor, si no se usan referencias y se regresan las direcciones de los objetos, tendrá que escoger entre estas dos sintaxis.

```
*A[1] -> operator[](2) = 40;
```

o bien

```
*(*A[1])[2]) = 40;
```

Lo cual no resulta muy claro.

A continuación se muestra un ejemplo que usa dos niveles de corchetes.

```
// Programa 52, Prueba de doble []
#include <iostream.h>
#include <string.h>


class interior {
     enum {tam = 40 };
     char x[tam];
```

```
public:
     interior(char *X = "") {
          strcpy(x, X);
     }
     char& operator[](int j) {
          return x[j];
     }
};

class exterior {
     interior y[2];
public:
     exterior(interior &m, interior &n) {
          y[0] = m;
          y[1] = n;
     }
     interior& operator[](int i) {
          return y[i];
     }
};

void main()
{
     exterior o(interior("hola"), interior("María"));

     cout << o[0][0] << o[0][1] << o[0][2] << o[0][3] << endl;
     cout << o[1][0] << o[1][1] << o[1][2] << o[1][3] << endl;
}
```

OBSERVACIONES.

La salida del programa es:

```
hola
Mari
```

Este programa define una clase **exterior**, con un dato miembro el cual es un arreglo del tipo interior, con 2 elementos. El tipo interior es una clase con un elemento del tipo arreglo a char de 40 caracteres.

En realidad los corchetes cuadrados sobrecargados nos permiten referenciar un arreglo bidimensional de caracteres, auxiliandonos de 2 clases **interior** y **exterior**, y de los corchetes sobrecargados.

Cuando declaramos el objeto **o** primero se hace la llamada (2 veces) al constructor de la clase **interior** y se almacena en **x** (separadas) ambas cadenas, para luego mediante los cast a **interior**, llamar al constructor de **exterior** y asignar las direcciones de las cadenas a cada elemento del dato miembro, arreglo **y**.

Después, manejando el objeto indexado, podemos hacer referencia a cada elemento del objeto **o** e irlos imprimiendo en pantalla.

4.e) CERO vs NULL.

De acuerdo a la referencia bibliográfica:

The Annotated C++ Reference Manual
Ellis and Stroustrup
Addison Wesley, 1990

En C++ no se garantiza que la constante simbólica NULL sea compatible con todos los tipos de apuntadores. Algunas implementaciones definen el NULL como **(void *)0**, él cual no puede ser asignado a otro tipo de puntero sin usar un cast, por lo que el símbolo NULL, es un artificio empleado en C, antes de que se usaran los prototipos, para asegurar el uso correcto de argumentos a los que se les asignaba un puntero NULL.

RESUMIENDO. En C++ deberemos usar siempre 0, nunca NULL, aunque se encuentre definido en el paquete.

4.f) El constructor de copiado y el operator=(), los cuales
pueden ser creados por el compilador.

El constructor de copiado, él cual también puede referirse como X(&X), es un constructor que tiene como argumento una referencia a un objeto de la misma clase. Es una función usada por el compilador para copiar objetos por valor dentro de una función cuando los objetos son usados como argumento y hacia fuera de la función cuando un objeto es regresado por valor. Por ejemplo en la definición de la función:

```
X funcx(X c) {
     return c;
}
```

* Primero c es pasado por valor, llamando a X(&X)
* Segundo c es regresado por valor, llamando a X(&X)

El constructor de copiado, también se usa para inicializar un nuevo objeto, desde otro objeto anterior, como se ve en las siguientes expresiones:

```
X a;
X b = a;  // Se llama a X(&X)
```

En versiones anteriores a la 2.0 de AT&T, y los compiladores desarrollados bajo estas características, si el programador no define un constructor de copiado el compilador recurre al copiado a nivel de bit, lo cual significa que la estructura de un objeto es directamente copiada en la estructura de otro objeto. Esto funciona para objetos simples, aquellos que no contienen apuntadores o instancias de otras clases, pero no es apropiado cuando los objetos contienen miembros y apuntadores a otros objetos, por lo cual, el programador estaba obligado a definir el constructor de copiado.

De manera similar el **operator=()** es llamado cuando un objeto ya ha sido inicializado anteriormente. Por ejemplo:

```
X d;
X e = d;  // Se llama a X(&X)

X f;
f = d;    // Se llama a X::operator=(X&)
```

Si **f** contiene apuntadores u objetos miembro, el **operator=()** es el responsable de su inicialización.

Como con el constructor de copiado, si el **operator=()** no fué definido por el programador, en implementaciones anteriores a la 2.0 de AT&T, el compilador recurría al copiado a nivel de bit del operando del lado derecho al del lado izquierdo, ignorando la necesidad de inicialización del operador del lado izquierdo, antes del copiado. Esta omisión nos genera problemas ya que si el objeto del lado izquierdo contiene apuntadores, estos son modificados en el proceso de copia lo que puede propiciar que el programa corra en localidades de memoria no definidas y aborte el sistema. Cuando el programador escriba el **operator=()** deberá asegurarse de inicializar adecuadamente el operando izquierdo, antes de realizar el proceso de asignación.

En versiones resientes de ANSI C++, el programador tiene la opción de decidir si crea estas funciones o no. Si las define, el compilador usa la definición realizada por el programador, si no lo hace, el nuevo ANSI C++, automáticamente genera un inicializador, en lugar de la forma tradicional de copia a nivel de bit, ésto es válido tanto para el operador de copia **X(&X)**, como para el **operator=()**.

Veamos el empleo de X(X &a) y X::operator=()(X &a)

```
// Programa 52A, Ejemplo de uso del operador X(&X) e (=)
#include <iostream.h>
#include <string.h>

class angel {
     char cad[50];
     void *x;
     int tam;
```

```
public:
     angel(char *msg = "") {
          x = msg;
          strcpy(cad, msg);
          tam = strlen(msg);
     }
     angel(angel &c) {
          x = c.cad;
          strcpy(cad, c.cad);
          tam = strlen(c.cad);
     }
     angel operator=(angel c) {
          x = c.cad;
          strcpy(cad, c.cad);
          tam = strlen(c.cad);
          return *this;
     }
     angel operator+(angel c) {

          strcat(cad, c.cad);
          tam = strlen(cad);
          return *this;
     }

     void muestra(char *msg = "") {
          cout << endl;
          cout <<  msg << endl;
          cout <<  "Cadena cad: " << cad << endl;
          cout <<  "Dirección Origen de cad: " << (void *)cad <<
endl;
          cout <<  "Dirección Origen de x  : " << x << endl;
          cout << "Tamaño de cad: " << tam << endl;
     }
};


void main()
{
     angel A("Buenos Dias ");
     angel B("Bienvenida Primavera"), C(A);
     angel D;

     A.muestra("A");
     B.muestra("B");
     C.muestra("C");
     D.muestra("D");
     C = A;
     C.muestra("C = A");
     C = A + B;
```

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

OBSERVACIONES.

El constructor de copiado es llamado en el programa anterior, al declarar angel C(A). También se realiza un llamado a este constructor dentro de la sección de parámetros; tanto del operador (=) como del operador (+), así mismo, el constructor de copiado es llamado al regresar las funciones anteriores un objeto en la sentencia return.

Veamos la corrida.

```
A
Cadena cad: Buenos Dias
Dirección Origen de cad: 0xff90
Dirección Origen de x  : 0x00ac
Tamaño de cad: 12

B
Cadena cad: Bienvenida Primavera
Dirección Origen de cad: 0xff5a
Dirección Origen de x  : 0x00b9
Tamaño de cad: 20

C
Cadena cad: Buenos Dias
Dirección Origen de cad: 0xff24
Dirección Origen de x  : 0xff90
Tamaño de cad: 12

D
Cadena cad:
Dirección Origen de cad: 0xfeee
Dirección Origen de x  : 0x00aa
Tamaño de cad: 0

C = A
Cadena cad: Buenos Dias
Dirección Origen de cad: 0xff24
Dirección Origen de x  : 0xfeb8 /* Esta dirección se debe a la */
Tamaño de cad: 12               /* construcción de un objeto  */
                                /* por el C. de Copiado.      */
C = A + B
Cadena cad: Buenos Dias Bienvenida Primavera
Dirección Origen de cad: 0xff24
Dirección Origen de x  : 0xfe16 /* También es producto del    */
```

```
Tamaño de cad: 32                /* C. de copiado.              */
```

NOTA: El constructor de copiado crea un objeto al copiar el argumento de llamada en el parámetro de la función. Recuerde que estamos pasando un objeto por valor y estamos regresando un objeto por valor.

Tenga precausión ya que la dirección de **x**, puede ser ya no válida.

5.- **Creación de operadores de conversión de tipos propios.**

C++ nos dá la posibilidad de definir un tipo y tratarlo de la misma manera que un tipo predefinido, ésto se extiende también al cast, en los tipos definidos por el usuario. Cuando el compilador se dá cuenta de que se esta utilizando un cierto tipo en donde otro tipo es requerido, llama al operador de cast, el cual también deberá ser definido por el usuario, veamos un ejemplo.

```
// Programa # 53, uso de cast a tipos definidos por el usuario.
#include <stdio.h>

class numero1 {
    float f;
public:
    numero1(float x = 0.0) { f = x; }
    operator int() {    // Regresa el valor establecido por la
        return int(f);  // función.
    }
    void muestra() {
        printf("\nValor de f = %6.3f", f);
    }
};

class numero2 {
    int i;
public:
    numero2(int x = 0) { i = x; }
    operator numero1() {            // cast a un tipo definido
        return numero1(float(i));          // por el usuario.
    }
};


void imprime(numero1 A);


void main()
{
```

```
    numero1 b(5.5);
    numero2 c(25);

    int x = b;  // Se llama al numero1::operator int()
    b = c;      // Se llama al numero2::operator numero1()
    imprime(b);
    printf("\nValor de x = %d", x);
}


void imprime(numero1 A)
{
    A.muestra();
}
```

OBSERVACIONES.

La salida nos resulta:

```
Valor de f = 25.000
Valor de x = 5
```

En el programa se manejan 2 tipos de conversión.

En la clase **nunero1**, el **operator int()** es una manera de convertir un **numero1** en un **int**. Note que no se debe especificar el valor de retorno en el operador de conversión de tipo, ya que el nombre del operador es el tipo del valor regresado.

En la clase **numero2**, el **operator numero1()**, es la forma de la conversión de tipo de un tipo **numero2** a un tipo **numero1**.

La sentencia: int x = b;

Hace que el compilador llame al primer operador de conversión de tipo.

La sentencia: b = c;

Hace que el compilador llame al segundo operador de conversión de tipo.

En el programa anterior, la conversión de tipos se realiza por medio de la llamada a una función miembro de la clase usada como argumento.

Es posible también realizar la conversión de tipos, creando un constructor que tenga un solo argumento, él cual es del otro tipo definido por el usuario.

La clase fuente deberá tener acceso a la clase destino, ésto se hace declarando el constructor de la clase destino como miembro friend de la clase fuente.

También puede hacerse mediante funciones que tengan acceso al constructor del objeto, de tal forma de poder leer los datos necesarios. Veamos un ejemplo, usando ambos metodos:

```
// Programa # 54, Constructor con un solo argumento
// para realizar conversión de tipo.
// En este programa se muestran dos alternativas
// las cuales le permiten al objeto destino accesar
// a los datos privados del objeto fuente.
#include <stdio.h>

class dino;

class plus {
    float f1, f2;
public:
    plus(float x1 = 0.0, float x2 = 0.0) {
        f1 = x1;
        f2 = x2;
    }
    plus(dino &);   // Constructor de conversión de tipos.
    float f_uno() { return f1; } // función de acceso
    float f_dos() { return f2; }
    void muestra() {
        printf("\nValor de f1 = %6.2f", f1);
        printf("\nValor de f2 = %6.2f", f2);
    }
};

class dino {
    int i1, i2;
public:
    dino(int y1 = 0, int y2 = 0) {
        i1 = y1;
        i2 = y2;
    }
    dino(plus &);   // Constructor de conversión de tipos.
    friend plus::plus(dino &);  // Tiene acceso a datos privados
    void muestra() {
        printf("\nValor de i1 = %6d", i1);
        printf("\nValor de i2 = %6d", i2);
    }

};

// Este constructor tiene permiso para leer los datos privados
// en la clase dino, es una función friend.
plus::plus(dino& m)
{
```

```
    f1 = m.i1;
    f2 = m.i2;
}

// Este constructor usa las funciones de acceso en plus para
// leer los datos privados. No es una función friend, entonces
// deberá obtener los datos de forma diferente que la lectura
// directa a los elementos de la clase.



dino::dino(plus& s)
{
    i1 = (int)s.f_uno();
    i2 = (int)s.f_dos();
}

void fun_A(plus a);
void fun_B(dino b);

void main()
{
    plus S(5.5, 6.6), T;
    dino M;

    M = S;      // Se llama a dino::dino(plus &)
    T = M;      // Se llama a plus::plus(dino &)
    fun_A(M);   // Se llama a plus::plus(dino &)
    fun_B(T);   // Se llama a dino::dino(plus &)
}

void fun_A(plus a)
{
    a.muestra();
}

void fun_B(dino b)
{
    b.muestra();
}
```

OBSERVACIONES.

La salida nos resulta:

```
Valor de f1 =   5.00
Valor de f2 =   6.00
```

```
Valor de i1 =      5
Valor de i2 =      6
```

En el inicio del archivo es necesario incluir la declaración de nombre de clase **class dino**, ya que **class plus** usa un argumento del tipo **dino** en el constructor de conversión de tipos, y el compilador deberá saber que el tipo **dino** es legal en la declaración de parámetros.

La **clase plus** tiene sus funciones de acceso f_uno() y f_dos(), las cuales le permiten al constructor de la clase **dino**, **dino::dino(plus &)** tener acceso a los datos de **class plus**, y poder asignar sus datos miembro del tipo float, a los datos miembro de **dino** del tipo int.

En **class dino** no existen estas funciones, el acceso a los datos privados de **class dino** se logra por medio del constructor friend **plus::plus(dino &)** la información es tomada de **class dino** y mediante un cast implícito es asignada a los datos miembro de **class plus**.

Al declarar el objeto **S** de **class plus**, este es inicializado con (5.5, 6.6), al asignarle el valor a M de **class dino** es llamado el constructor **dino::dino(plus &)**, mediante el cual los datos miembro de clase **plus** son convertidos a tipo int y asignados a los datos miembro de clase **dino**.

Al hacer la asignación **T = M**, ahora es llamado el constructor friend **plus::plus(dino &)** él cual realiza la conversión de tipo int a tipo float, y asigna los valores a los datos miembro de clase **plus** de manera implícita.

Por último las funciones muestra() de **plus** y **dino** presentan los datos almacenados en las instancias M y T, mediante las funciones fun_A() y fun_B(), las cuales reciben como argumento objetos del tipo de las clases **dino** y **plus**, respectivamente. Note que ya perdieron la parte decimal debido a las transferencias realizadas.

5.a) Seleccionando entre operadores de cast y constructores de conversión.

Como se puede ver en los ejemplos anteriores, el programador define el tipo de cast a utilizar, ya sea mediante un constructor o un operador el cual nos realiza la conversión de tipos.

El compilador ve a ambas formas de realizar la conversión de la misma manera, ésto es, realiza la llamada de forma implicita a una función la cual nos realiza la conversión cuando es necesaria. La decisión sobre cual utilizar depende de cada caso en particular.

* Si se dispone de la clase que necesita ser convertida, se puede escribir un operador de cast para ella. Como ejemplo se tiene un objeto de la clase **numero2**, el cual necesita ser converido a la clase **numero1**.

* Si no se dispone de la clase que necesita ser convertida, y se tiene la suficiente información, se puede escribir un constructor de conversión para la clase destino. Como ejemplo se puede pensar que en el pasado programa, no se dispone de la clase **plus**, y se tiene la clase **dino**, se define el constructor **dino::dino(plus &)**, mediante el cual convertimos un objeto del tipo **plus** a un objeto de tipo **dino**.
* Se puede usar el operador de cast, cuando se convierte un tipo definido por el usuario a un tipo predefinido, ya que no se puede definir un constructor para un tipo predefinido.
* Es importante no utilizar un constructor y un operador de cast para el mismo tipo de conversión, ya que se puede crear ambigüedad sobre la manera de realizar la conversión. El compilador genera un mensaje de error si se intenta hacer ésto.

5.b) Riesgos en el abuso al usar conversores de tipo.

La conversión automática de tipos es muy conveniente y puede reducir el número de definiciones de funciones que necesitan ser escritas. Por ejemplo, no usando conversión automática de tipos se deberá contar con 2 versiones sobrecargadas de una función, un ejemplo es, llamada a pone();

```
void pone(plus);
void pone(dino);
```

Sin embargo, si existe una función de conversión de tipo, por ejemplo:

```
plus::operator dino();
```

ó

```
dino::dino(plus);
```

Se necesita solo una versión de pone().

```
void pone(dino);
```

Si se llama a la función con un objeto del tipo **dino**, la función corresponde directamente en argumento y parámetro, si la función se llama con un objeto del tipo **plus**, el compilador, mediante el operador de cast o el constructor, lo convierte al tipo **dino**.

Este tipo de conversión ahorra trabajo, el programador puede ser tentado a utilizarlo ampliamente, lo que puede traer problemas.

* **Primero**, si se escriben dos formas de realizar el mismo tipo de conversión, el compilador no puede resolver la ambigüedad, ésto se ilustra en el siguiente código.

```
// Programa # 55, Dos formas de realizar el mismo tipo
```

```
// de conversión.
// NOTA: este programa marca un error, a menos que se elimine
//       una de las funciones de conversión de tipo.

#include <iostream.h>
class hola;

class chica {
    int d;
public:
    chica(int dd = 0) { d = dd; }
    chica(hola & s); // Conversión de tipo de hola a chica
    int pone() { return d; }
};

class hola {
    double xd;
public:
    hola(double dd = 0.0) { xd = dd; }
    // Otra forma de convertir hola a chica
    operator chica() {return chica((int)xd);} // Esta función es
    double pone() { return xd; }  // ambigua con el constructor
};                                // de conversión.

chica::chica(hola& s)
{
    d = int(s.pone());
}

void fun_A(chica);

void main()
{
    hola H(15.52), S(50.5);

    fun_A(H); // Conversión implícita de tipo.
    cout << "Valor de d  = " << H.pone() << endl;
    fun_A(S);
}

void fun_A(chica X)
{
    cout << "Valor de xd = " << X.pone() << endl;
}
```

OBSERVACIONES.

Al compilar el programa anterior el compilador va a generar un mensaje de error. Este tipo de problema es de esperarse, ya que se genera un mensaje de error en el momento de crear la segunda forma de conversión de tipo.

```
operator chica() {return chica((int)xd);}
```

* **Segundo**, otro problema se produce si se desea convertir un objeto de una clase a más de un tipo. Si una función sobrecargada lleva más de un argumento con conversión automática de tipo, va a producir problemas, aquí un ejemplo.

```
// Programa # 56, varias conversiones automáticas de tipo.
// producen ambigüedad.
// NOTA: El programa al compilarse va a producir un error
//       de ambigüedad.
#include <iostream.h>

class chica {
    int d;
public:
    chica(int dd = 0) { d = dd; }
    int pone() { return d; }
};

class hola {
    double xd;
public:
    hola(double dd = 0.0) { xd = dd; }
    double pone() { return xd; }
};

class nina {
    short sh;
public:
    nina(short i = 0) { sh = i; }
    operator hola() { return hola((double)sh); }
    operator chica() { return chica((int)sh); }
};

void fun_A(chica);
void fun_A(hola); // Esta función deberá eliminarse.


void main()
{
    nina H(15);
    hola X;
    chica Y;
```

```
    X = H;
    Y = H;
    fun_A(H); // Conversión implícita de tipo.
    cout << "Valor de d  = " << X.pone() << endl;
    fun_A(Y);
}



void fun_A(chica X)
{
    cout << "class chica: Valor de xd = " << X.pone() << endl;
}

// Para que funcione bien el programa se debe cancelar la
// siguiente función.

void fun_A(hola Y)
{
    cout << "class hola:  Valor de xd = " << Y.pone() << endl;
}
```

OBSERVACIONES.

En este caso el problema no aparece de manera inmediata. El programa funciona bien hasta que no se sobrecarga una función que toma argumentos tanto de la clase **chica** como de la clase **hola**.

El problema se puede solucionar modificando el archivo de cabecera (eliminando la conversión de tipo automática no deseada, la conversión restante deja de ser ambigua). Veamos la corrección al programa anterior.

```
// Programa # 56A, Corrección al programa 56.
#include <iostream.h>
class hola;

class chica {
    int d;
public:
    chica(int dd = 0) { d = dd; }
    chica(hola& s);      // Se agregó de nuevo este conversor.
    int pone() { return d; }
};

class hola {
    double xd;
```

```
public:
    hola(double dd = 0.0) { xd = dd; }
    double pone() { return xd; }
};

class nina {
    short sh;
public:
    nina(short i = 0) { sh = i; }
    operator hola() { return hola((double)sh); }
//  operator chica() { return chica((int)sh); }
};

chica::chica(hola& s)
{
    d = (int)s.pone();
}

void fun_A(chica);
void fun_A(hola);

void main()
{
    nina H(15);
    hola X;
    chica Y;

    X = H;
    Y = H;
    fun_A(H); // Conversión implícita de tipos.
    cout << "Valor de d  = " << X.pone() << endl;
    fun_A(Y);
    fun_A(X);
}

void fun_A(chica X)
{
    cout << "class chica: Valor de d  = " << X.pone() << endl;
}

void fun_A(hola Y)
{
    cout << "class hola:  Valor de xd = " << Y.pone() << endl;
}
```

OBSERVACIONES.

Debemos eliminar una de las conversiones automáticas de tipo, escogemos la siguiente:

```
operator chica() { return chica((int)sh); }
```

Luego definimos un constructor de conversión de tipo, para convertir un objeto de clase **hola** a clase **chica**, con lo que completamos el esquema de conversión, con ello ya es posible sobrecargar la función fun_A(), para las clases **hola** y **chica**.

En las siguientes asignaciones presentadas en la función main(), las llamadas a los conversores de tipo, se realizan de la siguiente manera:

X = H;

En este caso se llama a **nina::operator hola()**, él cual realiza la conversión.

Y = H;

En este caso no hay una conversión directa entre un objeto de **class nina** y un objeto de **class chica**, por lo que primero se llama a la función de conversión **nina::operator hola()**, el objeto es convertido a tipo **hola**, a continuación se llama al constructor de conversión de tipo de **class chica**, llamado **chica::chica(hola&)** él cual convierte del tipo **hola** a un objeto de tipo **class chica**, con lo que se completa el proceso de conversión.

Algunos compiladores, inclusive algunas versiones del mismo compilador, por ejemplo la versión 4 de BCC, no acepta la conversión múltiple, es necesario realizar solo conversiones simples, ésto asegura un mejor control del programa.

El Borland C++ Ver 3.1 si la acepta.

La corrida nos resulta:

```
class hola:  Valor de xd = 15
Valor de d  = 15
class chica: Valor de d  = 15
class hola:  Valor de xd = 15
```

En la sobrecarga de la función **fun_A()**:

* Se llama fun_A(H). Donde H es de class nina, es pasada haciendo un cast a class hola por medio de **nina::operator hola()** y llama a la función sobrecargada fun_A(hola).

* Se llama fun_A(Y). Donde Y es de la class chica, se llama directamente a la función sobrecargada fun_A(chica).

* Se llama fun_A(X). Donde X es de la class hola, se llama directamente a la función sobrecargada fun_A(hola).

5.c) Reglas de Diseño para la conversión de tipos.

Estas Reglas fueron sugeridas por R.B. Murray

1) No cree más de un tipo de conversión implícita, desde cada tipo definido por el usuario, a menos que sea absolutamente necesario, ésto va a eliminar el problema presentado en el programa PLCP53.
   Esto no significa que se esta limitado en el número de tipos que se pueden convertir. Esto significa que solo una conversión deberá permitirse de manera implícita.
   En seguida se presenta un ejemplo que muestra multiples conversiones con solo una conversión implícita.

```
// Programa # 57, múltiple tipo de conversión.
// pero solo una conversión implícita.
// El diseño de esta previene problemas futuros.
#include <iostream.h>
class hola;
class nena;

class chica {
    int d;
public:
    chica(int dd = 0) { d = dd; }
    chica(hola&);
    int pone() { return d; }
};

class hola {
    double xd;
public:
    hola(double dd = 0.0) { xd = dd; }
    hola a_hola(nena a);
    double pone() { return xd; }
};

class nina {
    short sh;
public:
    nina(short i = 0) { sh = i; }
    nina a_nina(nena b);
    operator hola() { return hola((double)sh); }
};
```

```
class nena {
    long lg;
public:
    nena(long x = 0) { lg = x; }
    operator chica() {  // Se realiza una conversión implícita.
        return chica((int)lg); }
    friend hola hola::a_hola(nena a);  // nena -> hola
    friend nina nina::a_nina(nena b);  // nena -> nina
    void muestra() {
        cout << "class nena:  Valor de lg = " << lg << endl;
    }
};

chica::chica(hola & s)
{
    d = (int)s.pone();
}

hola hola::a_hola(nena a)
{
    xd = a.lg;
    return *this;
}

nina nina::a_nina(nena b)
{
    sh = b.lg;
    return *this;
}

void fun_A(chica);
void fun_A(hola);


void main()
{
    nina H(15);
    hola X;
    chica Y;
    nena Z(300);

    X = H;          // nina -> hola
    fun_A(X);       // hola
    Y = H;          // nina -> chica
    fun_A(Y);       // chica
    Z.muestra();    // nena
    X.a_hola(Z);    // nena -> hola DEFINIDA EN ESTE CASO
    fun_A(X);       // hola
```

```
    H.a_nina(Z);    // nena -> nina DEFINIDA EN ESTE CASO
    fun_A(H);       // nina -> hola
}

void fun_A(chica X)
{
    cout << "class chica: Valor de  d = " << X.pone() << endl;
}

void fun_A(hola Y)
{
    cout << "class hola:  Valor de xd = " << Y.pone() << endl;
}
```

OBSERVACIONES.

En **class nena** solo definimos un tipo de conversión implícito **nena::operator chica()**, los otros tipos de conversión de **class nena** a **class hola** y de **class nena** a **class nina**, son tipos de conversión explicitos, los cuales se deben realizar por llamada explícita a la función de conversión, como se puede apreciar en esta versión del programa 53, al cual se le agregó, **class nena**.

La salida nos resulta:

```
class hola:  Valor de xd = 15
class chica: Valor de  d = 15
class nena:  Valor de lg = 300
class hola:  Valor de xd = 300
class hola:  Valor de xd = 300
```

2) Un operador de conversión siempre deberá partir de un tipo complejo hacia un tipo simple. Si un tipo es una extensión lógica de otro tipo, el operador de conversión deberá partir de la extensión y crear el tipo simple desde ella. Como las extensiones se derivan de tipos simples, si queremos ir de un tipo simple a una extensión particular puede causar problemas. El siguiente ejemplo nos muestra la dirección que se debe seguir para manejar los operadores de conversión.

```
// Programa # 58, Operadores de conversión
// Deberán Generar simplicidad.

#include <iostream.h>

class precio {
     unsigned long base;          // Viejos pesos
```

```
     unsigned long t_impuesto;
public:
     void pone_base(unsigned long b) { base = b; }
     void pone_imp(unsigned long pi) { t_impuesto = pi; }
     void imprime(char *msg = "") {
          if(*msg) cout << msg << ": " << endl;
          cout << "Precio = $" << base/100 << "."
                                << base%100 << endl;
          cout << "Impuesto = $" << t_impuesto/100 << "."
                << t_impuesto%100 << endl;
          cout << "Total = $" << (base + t_impuesto)/100
                << "." << (base + t_impuesto)%100 << endl;
     }
     unsigned long precio_base() { return base; }
     unsigned long impuesto() { return t_impuesto; }
};


class no_deducible {
     char *nombre;
     precio costo;
public:
     no_deducible(char *nm, unsigned long p, unsigned long pi) {
          nombre = nm;   // Es posible si se usa una constante de
          costo.pone_base(p);     // cadena
          costo.pone_imp(pi);
     }
     void imprime() {
          cout << "No deducible. ";
          costo.imprime(nombre);
     }
     // Conversión automática de tipo
     // Desde tipo más complicado a más simple
     operator precio() { return costo; }
};


class deducible {
     char *nombre;
     char *categoria;
     int porciento_ded;
     precio costo;
public:
     deducible(char *nm, char *ct, unsigned long p,
                    unsigned long st, int porcentaje) {
          nombre = nm;
          categoria = ct;
          costo.pone_base(p);
          costo.pone_imp(st);
```

```
          porciento_ded = porcentaje;
     }
     void imprime() {
          cout << "Categoría deducible " << categoria << ", ";
          costo.imprime(nombre);
          cout << "Porcentaje deducible: "
                << porciento_ded << endl;
          }
          // Tipo de conversión automático, de más complicado
          // a más simple
          operator precio() { return costo; }
};


// finalmente una clase para sumar todos los precios.
class precio_total {
     unsigned long sum;
public:
     precio_total() { sum = 0; }
     void imprime() {
          cout << "El total es: $" << sum/100 << "."
                << sum%100 << endl;
     }
     void suma(precio p ) {
          sum += p.precio_base();
          sum += p.impuesto();
     }
};

void main()
{
     no_deducible refrigerador("Refrigerador", 745956, 56300);
     deducible estampillas("Estampillas", "Postales", 24000, 0,
100);
     deducible revista1("DDJ", "Publicaciones", 2000, 0, 100);
     deducible revista2("RDP", "publicaciones", 1800, 0, 100);
     no_deducible comida("Pollo", 2253, 180);
     precio_total total;
     total.imprime();
     refrigerador.imprime();
     total.suma(refrigerador);
     total.imprime();
     estampillas.imprime();
     total.suma(estampillas);
     total.imprime();
     revista1.imprime();
     total.suma(revista1);
     total.imprime();
     revista2.imprime();
```

```
     total.suma(revista2);
     total.imprime();
     comida.imprime();
     total.suma(comida);
     total.imprime();
}
```

OBSERVACIONES.

En este caso tanto **class deducible** como **class no_deducible** cuentan con un operador de tipo de conversión implícito, de tal forma de producir un tipo simple **precio**. **class precio_total** cuenta con un método llamado **suma()**, el cual tiene como argumento una instancia del tipo **precio**. Por medio del operador de conversión implícito es posible sumar ambos objetos, de los tipos **deducible** y **no_deducible**, sin la necesidad de sobrecargar el método suma().

3) Es aceptable contar con conversión mutua entre 2 clases. No todas las clases tienen extensiones lógicas. Murray dá el siguiente ejemplo; si los numeros representados por double incluyen todos los numeros en una clase llamada **racional**, entonces es permitida la conversión mutua entre **double** y **racional**. Una función sobrecargada ya sea con un argumento **double** o **racional**, recibirá sin ningún problema cualquiera de los 2 argumentos; si la función solo tiene un tipo de argumento, la conversión de tipos se va a llamar automaticamente para el otro tipo de argumento, por lo tanto, no existe ambigüedad con conversiones mutuas.

RESUMIENDO: No use conversión de tipos implícita, miemtras no sea necesario. Si se usa el tipo de conversión implícito arbitrariamente, puede traer problemas para usuarios futuros de la clase.

Veamos la salida del programa:

```
El total es: $0.0
No deducible. Refrigerador:
Precio = $7459.56
Impuesto = $563.0
Total = $8022.56
El total es: $8022.56
Categoría deducible Postales, Estampillas:
Precio = $240.0
Impuesto = $0.0
Total = $240.0
Porcentaje deducible: 100
El total es: $8262.56
Categoría deducible Publicaciones, DDJ:
Precio = $20.0
Impuesto = $0.0
```

```
Total = $20.0
Porcentaje deducible: 100
El total es: $8282.56
Categoría deducible publicaciones, RDP:
Precio = $18.0
Impuesto = $0.0
Total = $18.0
Porcentaje deducible: 100
El total es: $8300.56
No deducible. Pollo:
Precio = $22.53
Impuesto = $1.80
Total = $24.33
El total es: $8324.89
```

La conversión de tipo tiene lugar cuando es llamada la función miembro de precio_total **suma()**, ya que debe convertir un tipo de argumento **deducible** o **no_deducible** a un tipo más simple, de **class precio**.

Note en el ejemplo anterior que **class precio** es usada como un objeto miembro tanto de **class deducible** como de **class no_deducible**.

6.- **Creación de funciones iostream propias.**

Una de las aplicaciones más útiles de forma inmediata para el manejo de la sobrecarga de operadores es la creación de funciones de Entrada/Salida propias usando iostream, con lo cual podemos hacer lo siguiente:

```
tipo_propio A;
cout << A;
```

Esto nos trae un desarrollo de código práctico y elegante.

El **operator<<()** ya esta sobrecargado en la **class iostream**, pero puede volverse a sobrecargar por una clase propia. La siguiente interface muestra un ejemplo de un par de vectores X-Y sobrecargados por ambas funciones stream; una función de entrada y una función de salida.

```
// Archivo PLCP59.H
// Vectores X-Y, usando función de salida stream
#ifndef XYVEC_H_
#define XYVEC_H_
#include <iostream.h>

const tam = 20;
```

```
class XYvec {
    float X[tam];
    float Y[tam];
    void error(char *msg);
public:
    XYvec(float xini = 0, float xetapa = 0,
          float yini = 0, float yetapa = 0);
    // Se modifican valores para sobrecargar la función
    // de llamada al operador.
    void operator() (int indice, float xval, float yval);
    // stream de E/S
     friend ostream& operator<<(ostream& s, XYvec& v);
     friend istream& operator>>(istream& s, XYvec& v);
};

#endif      // XYVEC_H_
```

En esta sección estamos presentando la interface del programa 56, es el archivo de cabecera PLCP56.H, en el cual declaramos como datos miembro 2 arreglos del tipo float, donde cada uno va a contener los valores de X y los valores de Y respectivamente.

En el constructor se esta pasando como argumento, el punto inicial y el incremento, tanto para X como para Y.

En la declaración de **operator()()**, es otra manera de cambiar el valor de un punto particular (X, Y).

Los operadores a ostream y a iostream, nos estan sobrecargando tanto el operador **<<**, como el operador **>>**.

A continuación se muestra el archivo PLCP59.CPP, donde estan definidos los metodos de la clase XYvec.

```
// Programa 59, metodos para el par de vectores X-Y
#include "plcp59.h"
#include <stdio.h>

void XYvec::error(char *msg)
{
     cerr << "XYvec error: " << msg << endl;
}

XYvec::XYvec(float xini, float xetapa,
                float yini, float yetapa)
{
     for(int i = 0; i < tam; i++) {
          X[i] = xini + i*xetapa;
          Y[i] = yini + i*yetapa;
     }
```

```
}

void XYvec::operator()(int indice, float xval, float yval)
{
     if(indice < 0 || indice >= tam)
          error("Indice fuera de Rango");
     X[indice] = xval;
     Y[indice] = yval;
}

ostream& operator<<(ostream& s, XYvec& v)
{
     s << "\tX\t\tY" << endl;
     s.precision(3);
     for(int i = 0; i < tam; i++)
          s << "\t" << v.X[i] << "\t\t" << v.Y[i] << endl;
     return s;
}


istream& operator>>(istream& s, XYvec& v)
{
     float val;
     int indice = 0;

     while(!s.bad() && !s.eof()) {
          s >> val;
          if(s.bad() || s.eof()) break;
          v.X[indice] = val;
          s >> val;
          if(s.bad() || s.eof()) break;
          v.Y[indice++] = val;
          if(indice == tam) break;
     }
     return s;
}
```

OBSERVACIONES.

La función miembro bad() regresa un valor diferente de CERO, si ocurre un error. La función eof() regresa un valor diferente de CERO si detecta un fin de archivo.

Debido al diseño de **iostreams** la sobrecarga del **operator<<()** o del **operator>>()** deberá hacerse por medio de una función global friend, entonces, como los operadores << y >> son binarios, la función deberá contar con 2 argumentos, el del lado derecho deberá corresponder al objeto y el del lado izquierdo a **ostream** o **istream**, dependiendo si se trata de una operación de salida o una de entrada. Estas

funciones deberan regresar el objeto a stream, usado como argumento, ésto es importante si se tienen expresiones de la forma:

```
cout << "arg1 = " << arg1 << "arg2 = " << arg2 << endl;
```

El efecto de cada argumento es sumado al stream.

En el **operator<<()**, mostramos los 20 puntos (X, Y) del objeto **A**, para ello dentro de la función usamos el operador sobrecargado en iostream [<<], haciendo referencia directa por medio del parámetro objeto **v** de los elementos (X, Y).

El **operator>>()**, recibe como argumento 2 referencias una a iostream y la otra al objeto **v**, si no usamos referencias no es posible disponer de los valores capturados dentro de la función. La función por medio de la expresión s >> val; toma el valor dado del teclado en val, luego val es asignada a un elemento del arreglo (X ó Y), por medio de las expresiones v.X[indice] ó v.Y[indice++].

A continuación mostramos una aplicación de la clase XYvec.

```
// Programa 59A, Una aplicación de la clase XYvec
#include "plcp59.h"

void main()
{
    XYvec A(1.14, 0.47, 2.59, 0.939);
    A(4, 77.77, 111.9);
    cout << "A = " << A << endl;

    XYvec B;
    cin >> B;
    cout << "B = " << B << endl;
}
```

OBSERVACIONES.

En la declaración de **A** usamos el constructor para llenar el arreglo de 20 elementos tanto de X como de Y.

En la segunda línea usamos el **operator()()** para modificar el elemento con indice 4, con los valores dados a continuación.

En la siguiente línea mostramos los 20 puntos generados en el constructor y el modificado, con indice 4.

Por último declaramos el objeto **B**, lo llenamos por teclado y luego mostramos los valores con que fué llenado el arreglo, usando el operador sobrecargado por el usuario [>>].

La salida nos resulta:

```
A =  X         Y
     1.14      2.59
     1.61      3.529
     2.08      4.468
     2.55      5.407
     77.77     111.9
     3.49      7.285
     3.96      8.224
     4.43      9.163
     4.9       10.102
     5.37      11.041
     5.84      11.98
     6.31      12.919
     6.78      13.858
     7.25      14.797
     7.72      15.736
     8.19      16.675
     8.66      17.614
     9.13      18.553
     9.6       19.492
     10.07     20.431
```

// En este punto la computadora pidió datos de entrada, se dieron
// los siguientes:
12.50 54.12 45.32 90.67 61.23 90.34 38.53 76.21 67.43 37.98 ^Z

```
B =  X         Y
     12.5      54.12
     45.32     90.67
     61.23     98.34
     38.53     76.21
     67.43     37.98
     0         0
     0         0
     0         0
     0         0
     0         0
     0         0
     0         0
     0         0
     0         0
     0         0
     0         0
```

```
0       0
0       0
0       0
0       0
```

NOTA: El último valor fué repetido, este es el efecto del ^Z, él cual antes de ser detectado es tomado por la variable local **val**, como no corresponde a su tipo, toma el valor anterior.