## 7.- Seleccionando funciones friend o funciones miembro para sobrecarga de operadores.

En muchas situaciones se obtienen resultados equivalentes si se usa tanto una función friend o una función miembro cuando se sobrecarga un operador. Una función friend solo contiene un argumento extra. (la función friend deberá tener ambos objetos pasados como argumento a ella, mientras la función miembro solo requiere de un argumento). Una pregunta interesante es: ¿Porqué existen ambas alternativas?.

Si se usa una función miembro y un argumento de un tipo diferente, la función miembro solo nos permite que el nuevo tipo se encuentre en el lado derecho del operador. Esto es, A + 2, puede ser válido, pero 2 + A, no lo es.

Una función friend, nos permite ambas combinaciones.

Estas variaciones se muestran en el siguiente ejemplo:

```
// Programa # 60, Porqué se necesitan funciones friend.
#include <iostream.h>

class entero1 {
    int i;
public:
    void pone(int ii = 0) { i = ii; }
    // Operador sobrecargado con un solo argumento.
    entero1 operator+(int);
    entero1 operator+(entero1);
    void muestra() {
        cout << "Salida clase entero1, i = " << i << endl;
    }
};

entero1 entero1::operator+(int x)
{
    entero1 result;
    result .pone(i + x);
    return result;
}

entero1 entero1::operator+(entero1 x)
{
    entero1 result;
    result.pone(i + x.i);
    return result;
}


class entero2 {
```

```
    int i;
public:
    void pone(int ii = 0) { i = ii; }
    // Es necesaria una función para cada posible combinación.
    friend entero2 operator+(entero2, entero2);
    friend entero2 operator+(entero2, int);
    friend entero2 operator+(int, entero2);
    void muestra() {
        cout << "Salida clase entero2, i = " << i << endl;
    }

};


entero2 operator+(entero2 x, entero2 y)
{
    entero2 result;
    result.pone(x.i + y.i);
    return result;
}


entero2 operator+(entero2 x, int a)
{
    entero2 result;
    result.pone(x.i + a);
    return result;
}

entero2 operator+(int a, entero2 x)
{
    entero2 result;
    result.pone(x.i + a);
    return result;
}


void main()
{
    entero1 A; A.pone(10);
    entero1 B;
    entero1 C; C.pone(20);

    cout << "Valores iniciales de A y de C" << endl;
    A.muestra();
    C.muestra();
    B = A + 4;
    cout << "Primer resultado de B" << endl;
    B.muestra();
```

```
    B = A + C;
    cout << "Segundo resultado de B" << endl;
    B.muestra();
//  B = 4 + A; Esto no es legal para la clase entero1
    entero2 D; D.pone(100);
    entero2 E;
    entero2 F; F.pone(200);

    cout << "Valores iniciales de D y de F" << endl;
    D.muestra();
    F.muestra();
    E = D + 40;
    cout << "Primer resultado de E" << endl;
    E.muestra();
    E = D + F;
    cout << "Segundo resultado de E" << endl;
    E.muestra();
    E = 40 + D;  // Esto si es legal para la clase entero2
    cout << "Tercer resultado de E" << endl;
    E.muestra();
}
```

OBSERVACIONES.

En la clase entero1 existen 2 versiones de sobrecarga de **operatoe+()**, uno tiene un argumento **int** y el otro tiene un argumento **entero1**, lo que significa que en main() es posible sumar un **entero1** a un **int** y un **entero1** a un **entero1**, pero no es posible sumar un **int** a un **entero1**.

En la clase **entero2**, **operator+()** es definido como friend, lo cual significa que **entero2**, nos permite las siguientes combinaciones:

* entero2, entero2
* entero2, int
* int, entero2

Lo anterior nos dice que una función **friend** tiene una sintaxis más natural.

NOTA: En el ejemplo anterior NO se esta usando un constructor de tal manera de evitar una conversión implícita y poder demostrar lo anteriormente expuesto.

La salida nos resulta:

```
Valores iniciales de A y de C
Salida clase entero1, i = 10
Salida clase entero1, i = 20
Primer resultado de B
```

```
Salida clase entero1, i = 14
Segundo resultado de B
Salida clase entero1, i = 30
Valores iniciales de D y de F
Salida clase entero2, i = 100
Salida clase entero2, i = 200
Primer resultado de E
Salida clase entero2, i = 140
Segundo resultado de E
Salida clase entero2, i = 300
Tercer resultado de E
Salida clase entero2, i = 140
```

En el ejemplo anterior se puede ver lo tedioso de tener que redefinir el mismo operador: **operator+(entero2, int)** y también; **operator+(int, entero2)**. Mediante el uso de tipo de conversión implícito se pueden eliminar ambas definiciones. Lo cual vemos en el siguiente ejemplo:

```
// Programa 61, Uso de tipo de conversión implicito
// elimina repetición de código.
#include <iostream.h>

class entero3 {
    int i;
public:
    entero3(int ii = 0) { i = ii; }
    friend entero3 operator+(entero3, entero3);
    void muestra() {
        cout << "Valor de entero3, i = " << i << endl;
    }
};

entero3 operator+(entero3 x, entero3 y)
{
    return entero3(x.i + y.i);
}


void main()
{
    entero3 A(10), B(20), C;
    int x = 4;

    cout << "Valores iniciales A, B y x:" << endl;
    A.muestra();
    B.muestra();
    cout << "Valor de x = " << x << endl;
```

```
    cout << "Se muestran las tres combinaciones" << endl;
    C = A + B;
    C.muestra();
    C = A + x;
    C.muestra();
    C = x + A;
    C.muestra();
}
```

OBSERVACIONES.

Cuando se llama a **operator+(entero3, entero3)** con un tipo entero, el compilador llama al constructor para hacer la conversión, la cual no es necesario indicar, se realiza de manera implicita.

La salida nos resulta:

```
Valores iniciales A, B y x:
Valor de entero3, i = 10
Valor de entero3, i = 20
Valor de x = 4
Se muestran las tres combinaciones
Valor de entero3, i = 30
Valor de entero3, i = 14
Valor de entero3, i = 14
```

En este caso estamos utilizando una variable como operando del tipo int (**x = 4**).

RESUMIENDO: Si se requiere de una acción específica, o también si se requiere forzar una sintaxis particular use una función miembro, como, **operator=(int)**, la cual nos previene el uso de formas como 3 = A. Si se requiere que el orden de los argumentos sea cualquiera use una función **friend**, (considere en este caso el tipo de conversión implícito, para reducir código).

## 8.- **Sobrecarga de funciones.**

La sobrecarga de operadores es solo un ejemplo de la idea general de sobrecarga de funciones.

| Sobrecargar una función significa que es posible crear varias funciones con el mismo nombre pero con diferentes argumentos |
|---|

El tipo más común de sobrecarga de una función se encuentra en la declaración de constructores que se requieren en una clase para el manejo de diferentes tipos de inicialización en una clase, las funciones sobrecargadas funcionan de manera automática; el programador no tiene más que definir más de una función con el mismo nombre.

Recuerde que también se pueden sobrecargar funciones que no son miembros de ninguna clase.

Tenga en cuenta que en versiones anteriores del lenguaje se usó la palabra **overload** antepuesta al nombre de la función que iba a ser sobrecargada, ésto ya no es necesario en versiones nuevas.

El compilador diferencia la llamada a una función sobrecargada de otras analizando los argumentos de cada función. Sin embargo, el enlazador ve un nombre, no una lista de argumentos, de esta forma el nombre de una función sobrecargada con una lista de argumentos deberá ser diferente a otra función con una lista diferente de argumentos. El enlazador diferencia una función sobrecargada de otra, ya que el compilador modifica el nombre sumandole algunos caracteres, los cuales contienen información acerca del tipo de argumentos de la función, de tal manera que la representación interna de dos funciones con el mismo nombre en realidad representan 2 funciones diferentes, si cada una tiene una lista de argumentos diferente. Además las funciones miembro de una clase llevan el nombre de la clase implícito. Si se desea ver como trabaja este mecanismo en un compilador en particular, se puede compilar el archivo usando algunas banderas y generar el código ensamblador para luego ver como los nombres son definidos por el ensamblador.

8.a) Manejo de la dirección de una función sobrecargada.

El manejo de la dirección de una función sobrecargada introduce ambigüedad, ya que el compilador tiene más de una función con el mismo nombre por seleccionar. La anbigüedad se puede eliminar especificando los tipos de argumentos en el puntero a la función que va a recibir la dirección de la función sobrecargada. Por ejemplo;

```
void fun(int);
void fun(float);

void (*pr_fun_int)(int) = fun;
void (*pr_fun_float)(float) = fun;
```

8.b) Enlace de seguridad de tipo (type-safe linkage)

En versiones anteriores a la 1.2 de C++ de AT&T, las funciones se manejaron de diferente manera, si llamamos a este tipo de código; plan C, cuando se manejaban funciones sobrecargadas, la primera función se manejaba con plan C y solo los

nombres subsecuentes eran tratados para ser manejados por el compilador como funciones sobrecargadas.

Cuando un módulo en C++ es compilado, el compilador genera nombres para las funciones, los cuales estan en función del nombre propiamente dicho y del tipo de sus argumentos. llamemosle a este proceso composición del nombre, ésto hace posible la sobrecarga de funciones y le ayuda al enlazador a localizar errores cuando se llaman funciones de otros módulos.

Hay situaciones en que no se quiere que el compilador realize la composición del nombre de una función, un caso tipico es cuando se enlazan funciones compiladas en C con funciones compiladas en C++.

Los 2 tipos de enlace soportados por los compiladores de C++ son; "C" y "C++", entonces si se desea declarar que una función usa el enlace C, se escribe:

```
extern "C" void fx(int);
```

Esta declaración le dice al compilador que no maneje a la función fx(int) como una función de C++.

Esta declaración también se puede aplicar a un bloque de funciones:

```
extern "C" {
     void fx(int);
     void fy(int);
     int fz(float);
};
```

Si se desea incluir todo un archivo de cabecera usando el enlace C, se deberá escribir la siguiente sentencia:

```
extern "C" {
#include "mi_arch.h"
#include <sysfile.h>
};
```