Sobrecarga de Funciones y Operadores

La operación de sobrecarga de funciones es uno de los aspectos mas interesantes en el diseño de lenguajes, la mayoría de los lenguajes orientados a objetos incorporan esta característica, que consiste en definir varias funciones que utilizan el mismo nombre. Otro mecanismo muy interesante incorporado en muy pocos lenguajes de programación es la sobrecarga de operadores, que permite al programador dar nuevos significados a los símbolos de los operadores existentes en el lenguaje.

1. Sobrecarga de Funciones

En C++ dos o más funciones pueden compartir el mismo nombre en tanto difieran en el tipo o número de argumentos. Cuando las funciones comparten el mismo nombre representando cada una de ellas un código diferente, se dice que están sobrecargadas (overloaded). Para conseguir la sobrecarga funciones simplemente hay que declarar y definir todas las versiones requeridas de la función.

También es posible y es muy común sobrecargar las funciones constructoras. Hay 3 razones por las que se debe de sobrecargar las funciones constructoras:

- Ganar flexibilidad
- Permitir arreglos dinámicos
- Construir copias de constructores.

A continuación mostramos un ejemplo donde emplearemos la sobrecarga de funciones. La primer función sobrecargada es Abs (que obtiene el absoluto de un número), en este caso especificamos que una de ellas recibirá como argumento un valor entero y otra un valor doble. El compilador se hará cargo de interpretar el argumento y decidir que función se hará cargo de él.

También definimos dos funciones max (mayor que) que regresaran como valor el elemento mayor, ya sea de dos números enteros o de una lista de números (array).

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int random(int n) // genera numeros aleatorios entre 0 y n-1
{return rand() % n; }

// declaracion de la funcion absoluto sobrecardaga

int Abs( int );
double Abs( double );

// declaracion de la function max sobrecargada
int max( int, int );
int max ( int *, int );
```

```
int main(){
	// permite generar numeros aleatorios time_t fecha_hora;
	srand( (unsigned) time( &fecha_hora ) );

	cout << "Valor absoluto de -13 es " << Abs(-13) << endl;
	cout << "Valor absoluto de -2.09 es " << Abs(-2.09) << endl;
	cout << "el mayor de 6 y 9 es " << max(6, 9) << endl;

	int n = 5;
	int *array = new int[n];

	cout << "la lista de numeros es : ";
	for (int i = 0; i < 5; i++) {
		array[i] = random( 10 );
		cout << array[i] << " ";
	}
	cout << " el mayor es : " << max(array, n);
	cin.get();
	return 0;
}

int Abs( int numero ){
	return numero < 0 ? -numero : numero;
}

double Abs( double numero ){
	return numero < 0 ? -numero : numero;
}

int max( int a, int b ){
	return a > b ? a : b;
}

int max( int *vector, int dim ){
	int m = 0;
	for (int i = 0; i < dim; i++)
	m = vector[i] > m ? vector[i] : m;
	return m;
}
```

También es posible emplear la sobrecarga de funciones para pretender que la función constructora puede generar un objeto a partir de cualquier tipo valido, por ejemplo una clase Fecha.

```
#include <iostream>
using namespace std;

class Fecha {
private:
	short dia, mes, anio;
public:
	Fecha(char *);
	Fecha(short, short, short);
};

Fecha::Fecha(char *str){
	cout << "Fecha: " << str << "\n";
}

Fecha::Fecha(short d, short m, short a){
	cout << "Fecha: "
	<< (d < 10 ?"0":"") << d << "/"
	<< (m < 10 ?"0":"") << m << "/"
	<< (a >= 0 && a <10 ? "0":"") << a;
}

int main(){
	Fecha A("26/12/04"); Fecha B(26,12,4); cin.get();
	return 0;
}
```

Argumentos Implícitos

Otra característica relacionada con la sobrecarga es la utilización de argumentos implícitos (como se vio en las clases con el constructor predeterminado) que permite dar un valor a un parámetro cuando no se especifico el argumento correspondiente en la llamada a la función, recuerde que la sustitución de argumentos se realiza de izquierda a derecha:

Sintaxis:

tipo (variable1 = valor, variable2 = valor, ... variableN = valor);

Por ejemplo, imagine que se tiene una función a la cual se le puede asignar valores predeterminados, es decir ya preestablecido, de esta manera la llamada a la función se podrá realizar empleando la cantidad necesaria de argumento, siempre y cuando el numero de argumentos sea menor o igual al establecido en el prototipo de la función.

```
#include <iostream>
using namespace std;

void funcion(int = 5, int = 7);

void main(){
	funcion(); funcion(10); funcion(20,30); cin.get (); return 0;
}

void funcion(int a, int b){
	cout<<"a: "<< a <<" b: "<< b <<"\n";
}
```

La sobrecarga de funciones es quizás uno de los aspectos mas llamativos e importantes de la programación orientada a objetos y puede, en la mayoría de los caso, ahorrar tiempo de diseño y programación. Aun cuando existe el riesgo de incrementar la complejidad del programa, como sucede con la sobrecarga de operadores.

2. Sobrecarga de Operadores

Los operadores de C++, al igual que las funciones, pueden ser sobrecargadas (overloaded), este es uno de los aspectos mas característicos de lenguajes como C++ y C#. La sobrecarga de operadores quiere decir que se pueden redefinir algunos de los operadores existentes en C++ para que actúen de una determinada manera, definida por el programador, con los objetos de una determinada clase. Esto puede resultar muy útil por ejemplo, para definir operaciones matemáticas con elementos tales como vectores y matrices. Así, por ejemplo, sobrecargando adecuadamente los operadores de suma (+) y asignación (=), se puede llegar a sumar dos matrices con una sentencia tan simple como:
C = A + B;

Un operador siempre se sobrecarga en relación a una clase, cuando se sobrecarga un operador no pierde su contenido original, gana un contenido relacionado con la clase. Para sobrecargar un operador se crea una función operadora (operator), que en ciertos casos será una función amiga a la clase. La sintaxis seria la siguiente:

```
tipo nombreClase :: operator operador ( parámetros ){
	sentencias;
}
```

Un operador puede estar sobrecargado o redefinido varias veces, de tal manera que actúe de un modo distinto dependiendo del tipo de objetos que tenga como operándoos. Es precisamente el tipo de los operándoos lo que determina qué operador se utiliza en cada caso.

A continuación se presenta una tabla con todos los operadores validos que pueden ser sobrecargados en C++:

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

Se pueden realizar cualquier actividad al sobrecargar los operadores pero es mejor que las acciones de un operador sobrecargado se ajusten al uso normal de ese operador. La sobrecarga tiene dos restricciones, no puede cambiar la precedencia del operador y que el numero de operadores no puede modificarse.

Los únicos operadores de C que no se pueden sobrecargar son el operador punto (.), el operador condicional (?:) y el operador sizeof. C++ añade otros dos a la lista: el operador de resolución de alcance (::) y puntero a miembro de un objeto (.*).

El objetivo último de la sobrecarga de operadores es simplificar el código a escribir, en realidad se llega a complicar bastante la definición de las clases. Una clase que disponga de operadores sobrecargados es una clase mas compleja de definir, pero mas sencilla e intuitiva de utilizar.

La sobrecarga de operadores regularmente se divide en 3 categorías:

- operadores binarios
- operadores lógicos-relacionales
- operadores unarios

Cada uno de estos grupos debe ser tratado de manera específica, a continuación se describirá el manejo de cada uno de ellos.

Operadores Binarios

La función operator del operador solo tendrá un argumento, este argumento contendrá al objeto que este en el lado derecho del operador. El objeto del lado izquierdo es el que genera la llamada a la función operator y se pasa implícitamente a través de this (referencia al objeto propietario de la llamada a la función).

A continuación describiremos la clase Coordenada que manipulara el operador binario de suma (+) para que realice de manera automática la suma de los objetos.

```
#include <iostream>
using namespace std;

class Coordenada{
private:
        int x, y;

public: Coordenada ()
{ x = 0;y = 0; } Coordenada (int a, int b)
{ x = a; y = b; }
void obtenxy (int &vx, int &vy)
{ vx = x; vy = y; }
Coordenada operator+ ( Coordenada );
```

```
};

Coordenada Coordenada::operator+ (Coordenada obj)
{
Coordenada temp;
temp.x = this->x + obj.x;
temp.y = this->y + obj.y;
return temp;
}

int main(){
Coordenada obj1(10,10), obj2(5,3),obj3;
int x, y;

obj3 = obj1 + obj2;
obj3.obtenxy( x, y );
cout << "Suma de obj1 mas obj2\n";
cout << "Valor de x: "<< x << " e y: " << y << " en obj3";
cin.get();
return 0;
}
```

Advierta que la clase Coordenada no cuenta con datos miembro tipo puntero, lo que implica que el operador de asignación "=" no deberá ser sobrecargado, ya que la transferencia de datos de una clase a otra se realiza byte por byte, generando una copia exacta.

El operador de asignación "=" suele sobrecargarse en el caso de que se tenga una clase en la cual uno o mas de los datos miembro sea un puntero. Más adelante se mostrara un ejemplo de cómo debe ser sobrecargado este operador para tal caso.

La sobrecarga es mas adecuada para clases matemáticas, ya que con frecuencia requieren de un conjunto completo de operadores sobrecargados, para garantizar la consistencia con la forma en que se manejan realmente estas clases matemáticas.

A continuación construiremos una clase Vector, en la cual iremos sobrecargando gradualmente cada uno de los operadores según se requieran. En una primera instancia tomaremos los operadores de suma, producto y la asignación, ya que uno de los elementos miembro de la clase es un puntero.

La declaración de la clase seria la siguiente:

```
class Vector {
private:
        int dim, *lista;
public:
        Vector( int = 1 ); // costructor predeterminado
        Vector( const Vector & ); // costructor de copia
        ~Vector() {
                delete [] lista;
         } // destructor

inline int longitud() { return dim; }
```

```
void captura( ); // captura el vector
void mostrar( ); // muestra el contenido del vector

Vector operator +( const Vector & ); Vector operator *( int );
friend Vector operator *( int, const Vector & );

const Vector & operator =( const Vector & );
};
```

Así pues implementaremos primero el operador de suma "+" para el caso de sumar dos entidades de tipo Vector, lo que dará como resultado un objeto de tipo Vector con el total de la operación. Recuerde que this es la referencia (apuntador) al objeto propietario de la llamada, es decir el objeto a la izquierda del operador.

```
Vector Vector::operator +( const Vector &arg )
{
if ( this->dim == arg.dim ) { Vector tem(this->dim); for(int i = 0; i < dim; i++)
        tem.lista[i] = this->lista[i] + arg.lista[i];
return tem;
}
else
        return Vector(1);
}
```

Tome en cuenta que todas las restricciones matemáticas deberán ser tomadas en cuenta al momento de la construcción de la clase, es decir, si realizamos una suma de vectores es necesario tomar en cuenta que la dimensión en ambos sea la misma. Además como resultado de la operación de suma obtendremos un nuevo vector resultante.

Ahora el resultado de la suma o de cualquier otro operador similar, deberá ser asignado en un objeto del mismo tipo (Vector). Además recordemos que tenemos un dato tipo puntero, por lo que resulta imprescindible realizar la sobrecarga de dicho operador (asignación), para garantizar la consistencia del objeto. A continuación veremos la manera de implementarlo:

```
const Vector & Vector::operator =( const Vector &arg )
{
if ( this != &arg ) { this->dim = arg.dim; delete [] lista;
        lista = new int [this->dim];
        for( int i = 0; i < dim; i++ )
                lista[i] = arg.lista[i];
}
return *this;
}
```

Primero verifica si se trata de una auto-asignación, en cuyo caso es evitada, después se transfiere la nueva dimensión al objeto propietario de la llamada (this), en seguida es liberado el espacio previo de memoria (para evitar almacenar basura en la memoria) y es reasignado con una nueva dimensión. Un ciclo for se encargara de colocar cada uno de los elementos de un vector al otro. Finalmente la función de asignación devolverá el objeto actual (es decir " *this ") como una referencia constante que permite asignaciones en cascada por ejemplo A = B = C.

También es posible sobrecargar los operadores para que se adecuen a las necesidades de la clase que se construye, en este caso sobrecargaremos el operador de producto " * " para multiplicar un Vector por una constante (es decir A * 2), lo que dará como resultado un Vector.

```
Vector Vector::operator *( int c ){
Vector tem( this->dim );
for ( int i = 0; i < tem.dim; i++ )
        tem.lista[i] = lista[i] * c;
return tem;
}
```

En este caso el operando de la izquierda es un objeto (propietario de la llamada) y puede manipular como argumento a la constante, pero existe la posibilidad de invertir esta operación, es decir "2 * A". Como vera la constante 2 no es un objeto Vector y por lo tanto no tendrá acceso a los datos y funciones miembros de la clase. Para estos casos es necesario construir una función friend que permita manipular los elementos privados de la clase, de esta manera ambos elementos de la operación se convertirán en argumentos de una función publica.

```
Vector operator *( int c, const Vector &A ){
        Vector tem( A.dim );
        for ( int i = 0; i < tem.dim; i++ )
                tem.lista[i] = c * A.lista[i];
        return tem;
}
```

Finalmente es posible utilizar la clase, de esta manera construiremos 3 objetos tipo Vector, con los cuales haremos algunas operaciones:

```
int main()
{
int i, j;
cout << "tamaño los vectores A y B ";
cin >> i >> j;
Vector A(i), B(j), C;
 cout << "\n Vector A \n"; A.captura();
cout << "\n Vector B \n";
B.captura();
cout << "\n suma \n";
C = A + B;
C.mostrar();
cout << "\n producto \n";
C = 2 * B; C.mostrar(); cin.get(); return 0;
}
```

Para los operadores sobrecargados miembro de una clase, el primer operando debe de ser siempre un objeto de esa clase, en concreto el objeto que constituye el argumento implícito (this). En la declaración y en la definición sólo hará falta incluir en la lista de argumentos el segundo operando. En los operadores friend el número de argumentos deberá ser el estándar del operador (unario o binario).

Operadores Lógicos y Relaciónales

Este tipo de sobrecarga de operadores no devuelve un objeto; en su lugar devolverán un valor booleano que indique verdad o falsedad en la relación de los objetos expuestos. Esto permite que los operadores se integren en expresiones lógicas y relacionales más extensas que admitan otros tipos de datos.

Ahora sobrecargamos el operador lógico de igualdad "==" y determinaremos si dos objetos de tipo Coordenada son iguales, en caso de ser así la función regresara como valor un verdadero. También implementaremos el operador de desigualdad, que consistirá simplemente en negar el resultado obtenido por el operador de igualdad.

```
class Coordenada
{
private:
int x,y;

public:
Coordenada (){x = 0; y = 0;}
Coordenada (int a, int b) {x = a; y = b;}
void obtenxy (int &, int &);
bool operator== ( Coordenada );
bool operator!= ( Coordenada obj ){
        return !( *this == obj ); }
};

bool Coordenada::operator== (Coordenada obj)
{
if( x == obj.x && y == obj.y )
        return true;
else
        return false;
}

int main(){
Coordenada obj1(10,10), obj2(10,9);

if( obj1 == obj2 )
        cout << "Objeto 1 y Objeto 2 son iguales";
else
        cout << " Objeto 1 y objeto 2 son diferentes";
cin.get();
return 0;
}
```

Ahora veremos la manera de implementar los operadores lógicos en nuestra clase de ejemplo llamada Vector. Sobrecargaremos los operadores de igualdad "==" y desigualdad "!=" que resultarían mas útiles de implementar.

```
class Vector
{
private:
int dim, *lista;
public:
Vector( int = 1 ); // costructor predeterminado
Vector( const Vector & ); // costructor de copia
~Vector() { delete [] lista; } // destructor

inline int longitud() { return dim; }
void captura( ); // captura el vector
void mostrar( ); // muestra el contenido del vector

bool operator ==( const Vector & );
bool operator !=( const Vector &arg )
{ return !( *this == arg ); }
};
```

La igualdad consistirá en verificar si los elementos de un Vector son exactamente los mismos al otro Vector, en cuyo caso regresara trae (verdadero).

```
bool Vector::operator ==( const Vector &arg )
{
if( dim != arg.dim )
        return false;
for( int i = 0; i < dim; i++ )
        if( lista[i] != arg.lista[i] )
                return false;
return true;
}
```

Para el caso de la desigualdad bastara con negar el resultado de la igualdad que ya esta implementado, únicamente hacemos la llamada con los objetos (recuerde que *this el la dereferencia del puntero this).

```
{ return !( *this == arg ); }
```

De esta manera se esta pidiendo al operador de igualdad que verifique si el objeto propietario *this
es igual al argumento arg.

Operadores Unarios

Las funciones miembro de este tipo pueden o no tener parámetro, eso dependerá del tipo de incremento que se implemente, es decir pre-incremento o post-incremento. En tal caso cada llamada será identificada de distinta manera. Recuerde que el operando es quien genera la llamada a la función operadora.

Las versiones prefijas se sobrecargan de la misma manera que un operador unario, en cuyo caso no será necesario colocar argumento alguno. Por otro lado las versiones postfijas necesitan un argumento de entrada que es declarado, pero generalmente nunca utilizado.

Ahora reemplazaremos los operadores de incremento para los objetos Coordenada en los componentes x e y de un objeto.

```
class Coordenada
{
private:
int x, y;
public: Coordenada()
{ x = 0; y = 0; }
Coordenada(int i, int j)
{ x = i; y = j; }
void obtenxy (int &vx, int &vy)
{ vx = x; vy = y; }
Coordenada & operator++(); // prefijo ++i
Coordenada & operator++( int ); // postfijo i++
};

Coordenada & Coordenada::operator++() // ++ objeto
{
++this->x; // incremento prefijo
++this->y;

return *this;
}

Coordenada & Coordenada::operator++( int ) // objeto ++
{
this->x++; // incremento postfijo this->y++;
cout << " objeto ++ ";

return *this;
}

int main()
{
Coordenada objeto(10,8);
int x,y;

objeto++;
objeto.obtenxy(x,y);
cout<< "Valor de x: " << x <<"    Valor de y: "<< y << "\n";
```

```
cin.get();
return 0;
}
```

Operadores de Flujo

Otra capacidad muy utilizada es la sobrecargar los operadores de inserción y extracción en los flujos de entrada (cin >>) y salida (cout <<), de manera que puedan imprimir o leer estructuras o clases complejas con una sentencia estándar. Para este caso es necesario declarar las funciones como amigas (friend) de la clase, ya que de otra manera seria imposible el acceso a los miembros privados del objeto que ingresa como segundo argumento.

```
class Coordenada
{
private:
int x, y;
public:
Coordenada()
{ x = 0; y = 0; }
Coordenada(int i, int j)
{ x = i; y = j; }
void obtenxy (int &vx, int &vy)
{ vx = x; vy = y; }

friend ostream &operator <<(ostream &, const Coordenada &);
friend istream &operator >>(istream &, Coordenada &);
};

ostream &operator <<(ostream &out, const Coordenada &obj)
{
out << "(" << obj.x << "," << obj.y << ")";
return out;
}

istream &operator >>(istream &in, Coordenada &obj)
{
cout << "X,Y "; in >> obj.x; in.ignore();
in >> obj.y;
return in;
}

int main()
{
Coordenada obj;

cin >> obj;
cout<< "coordenadas (x,y) " << obj << "\n";
cin.ignore(); cin.get(); return 0;
}
```

Ahora vamos a sobrecargar los operadores de flujo de entrada/salida para la clase Vector, en este caso el comportamiento que deseamos implementar en dicha clase, para el caso del flujo de entrada (cin >>), necesitamos que la función permita capturar los n elementos disponibles en el vector. Para el caso del operador de flujo de salida (cout <<), es necesario mostrar cada uno de los elementos del objeto Vector y enviarlos a la salida estándar.

Otra cosa importante que no mencionamos es que los métodos capturar y mostrar serán reemplazados por los operadores de flujo sobrecargados, ya que realizaran específicamente esas funciones. De tal forma que la clase quedaría de la siguiente manera:

```
class Vector
{
private:
int dim, *lista;

public:
Vector( int = 1 ); // costructor predeterminado
Vector( const Vector & ); // costructor de copia
~Vector() { delete [] lista; } // destructor

inline int longitud() { return dim; }

/* operador de insersion de flujo cin >> */
friend istream& operator >>( istream&, Vector& );
/* operador de extraccion de flujo cout << */
friend ostream& operator <<( ostream&, const Vector& );
};
```

La implementación de ambas clases es descrita a continuación, tome en cuenta que para el operado de flujo de entrada, la expresión " cin >> X ", implica que el objeto cin realiza la llamada, pero cin no es un objeto tipo Vector, de tal manera que el método deberá se una función amiga (global) de manera que permita a los datos de X ser accedidos de manera global. Además la referencia (&) del objeto cin deberá ser enviada, de esta manera se asegura que el flujo de entrada es correcto.

```
// sobrecarga del operador de influjo cin >>
istream& operator>>(istream& in, Vector& x)
{
for(int i = 0; i < x.dim; i++)
{
cout << "elemento "<<(i+1)<<" ";
in >> x.lista[i];
}
return in;
}
```

De esta manera nos aseguramos que cada uno de los elementos del arreglo será enviado al flujo de entrada y almacenado en la casilla correspondiente. Para el caso del operador de flujo de salida, también será necesario enviar la referencia del objeto cout a donde será enviado.

```
// sobrecarga del operador extraccion cout <<
ostream& operator<<(ostream& out, const Vector& x)
{
for(int i = 0; i < x.dim; i++)
out << x.lista[i] << "\t";
out << endl;

return out;
}
```

De esta forma colocamos cada elemento del arreglo en el flujo de salida, ahora podremos enviar la salida y entrada de flujo de manera transparente al usuario:

```
int main()
{
Vector A(6); cin >> A; cout << A;

return 0;
}
```

Operador de selección [ ]

Los operadores de selección suelen ser un caso muy especial; ya que por lo general tienen un comportamiento muy típico, es decir, necesitan hacer referencia a un elemento explicito dentro de la clase. Es decir nosotros necesitamos únicamente saber su referencia (dirección de memoria) y con esa información nosotros podremos manipular o extraer el contenido del elemento.

A continuación veremos como seria la implementación de un operador de este tipo, tomaremos como ejemplo la clase Vector.

```
class Vector
{
private:
int dim, *lista;

public:
Vector( int = 1 ); // costructor predeterminado
Vector( const Vector & ); // costructor de copia
~Vector() { delete [] lista; } // destructor

inline int longitud() { return dim; }

int & operator[]( int i )
{          if (i >= 0 && i < dim ) return lista[i]; }
};
```

Como vera el índice entre corchetes es transferido como argumento al operador sobrecargado indicándole el elemento i-ésimo requerido, la función miembro verifica que el índice sea valido

(de modo que no solicite una posición inexistente) y posteriormente regresara la referencia (ubicación en memoria) del elemento en la lista. De esta forma nosotros podremos utilizar

```
int main()
{
Vector A(10);
cout << "\n indice \n";
A[A.longitud()-1] = 8;    // A.operator[](A.longitud()-1) = 8
cout << A[0];    // A.operator[](0)

return 0;
}
```

La importancia del constructor de copia

Los casos mas especiales en la construcción de una clase son cuando nuestras clases se ven involucradas con punteros y es indispensable contar con un constructor de copia. El constructor de copia es el responsable de generar un objeto a partir de otro ya existente, al momento de construir un método de clase tenemos la necesidad de que regrese un objeto de la misma, este es el momento en que los constructores de copia son necesarios. La llamada a la función requiere enviar un objeto de la misma clase, entonces el constructor de copia genera la misma al momento de ejecutar el comando return generando una replica exacta solicitada por la llamada.

Bibliografía

Bjarne Stroustrup, "The C++ Programming Language", Addison-Wesley, second edition, 1991, ISBN 0-201-53992-6.

Naba Barkakati, “Object-Oriented Programming in C++”, Sams, first edition, 1992, ISBN 0-672-22800-9.

Harvey Deitel & Paul Deitel, “Cómo programar en C++”, Prentice Hall, segunda edición, 1999, ISBN 970-17-0254-9.

Naba Barkakati, “Turbo C++ Bible”, Sams, first edition, 1992, ISBN 0-672-22742-8.