

Décimocuarta Sesión

Metodologías y Técnicas de Programación II

Programación Orientada a Objeto (POO) C++

Sobrecarga de Operadores I

14.2 Operadores Sobrecargados

Sobrecarga

Hemos visto que las funciones podían ser sobrecargadas.

De igual forma los operadores también pueden sobrecargarse.

En realidad la mayoría de los operadores en C++ están sobrecargados.

Por ejemplo el operador + realiza distintas acciones cuando los operandos son enteros, o en coma flotante.

El operador * se puede usar como operador de multiplicación o como operador de indirección.

C++ permite al programador sobrecargar a su vez los operadores para sus propios usos.

```
<tipo> operator <operador> (<argumentos>);  
<tipo> operator <operador> (<argumentos>  
{  
    <sentencias>;  
}
```

14.2 Operadores Sobrecargados

A tener en cuenta

- Se pueden sobrecargar todos los operadores excepto ".", ".*", "::" y "?:".
- Los operadores "=", "[]", "->", "()", "new" y "delete", sólo pueden ser sobrecargados cuando se definen como miembros de una clase.
- Los argumentos deben ser tipos enumerados o estructurados: struct, union o class.

```
struct complejo
{
    float a,b;
};
complejo operator +(complejo a, complejo b);
complejo operator +(complejo a, complejo b)
{
    complejo temp = {a.a+b.a, a.b+b.b};
    return temp;
}

complejo x = {10,32};
complejo y = {21,12};
complejo z;
z = x + y;
```

14.2 Operadores Sobrecargados

A tener en cuenta

- Al igual que con las funciones sobrecargadas, la versión del operador que se usará se decide después del análisis de los argumentos.

También es posible usar los operadores en su notación funcional:

```
z = operator+(x,y);
```

```
.....
```

```
z = x + y;
```

```
.....
```

14.2 Operadores Sobrecargados

Sobrecarga de Operadores Binarios

Los operadores binarios son aquellos que requieren dos operandos con la suma o la resta. El operador incremento (++) es un operador unario.

Cuando se sobrecargan operadores en el interior de una clase:

Se asume que el primer operando es el propio objeto de la clase donde se define el operador. Debido a esto, sólo se necesita especificar un operando.

```
<tipo> operator<operador binario>(<tipo> <identificador>);
```

Normalmente el <tipo> es la clase para la que estamos sobrecargando el operador, tanto en el valor de retorno como en el parámetro.

14.2 Operadores Sobrecargados

```
class Tiempo
{
    public:
        Tiempo(int h=0, int m=0)
            : hora_(h), minuto_(m) {}
        void Mostrar();
        Tiempo operator+(Tiempo h);
    private:
        int hora_;
        int minuto_;
};

Tiempo Tiempo::operator+(Tiempo h)
{
    Tiempo temp;
    temp.minuto = minuto + h.minuto;
    temp.hora    = hora    + h.hora;
    if(temp.minuto >= 60) {
        temp.minuto -= 60;
        temp.hora++;
    }
    return temp;
}
```

```
void Tiempo::Mostrar()
{
    cout << hora << ":" << minuto << endl;
}

int main()
{
    Tiempo Ahora(12,24), T1(4,45);
    T1 = Ahora + T1;
    T1.Mostrar();
    (Ahora + Tiempo(4,45)).Mostrar();
    return 0;
}
```

14.2 Operadores Sobrecargados

Sobrecarga de Operadores Binarios

Lo normal es operar sobre dos objetos de la misma clase y devolver otro objeto también de la misma clase.

C++ nos permite esto:

```
int operator+(Tiempo h);
```

Hemos usado un objeto temporal para calcular el resultado de la suma, porque necesitamos operar con los minutos para prevenir el caso en que excedan de 60, en cuyo caso incrementaremos el tiempo en una hora.

Aquí utilizamos también el operador asignación. Lo crea el compilador por defecto a no ser que definamos uno nosotros (¿Os acordáis del constructor copia?)

```
T1 = Ahora + T1;
```

Se crean y utilizan dos objetos temporales son nombre:

```
(Ahora + Tiempo(4,45)).Mostrar();
```

14.2 Operadores Sobrecargados

Sobrecarga del operador asignación

```
class Cadena
{
public:
    Cadena(char *cad);
    Cadena() : cadena_(0) {};
    ~Cadena() { delete[] cadena_; };
    void Mostrar() const;
private:
    char *cadena_;
};
Cadena::Cadena(char *cad)
{
    cadena_ = new char[strlen(cad)+1];
    strcpy(cadena_, cad);
}
void Cadena::Mostrar() const
{
    cout << cadena_ << endl;
}
```

```
Cadena &Cadena::operator=(const Cadena &c)
{
    if(this != &c)
    {
        delete[] cadena;
        if(c.cadena)
        {
            cadena = new char[strlen(c.cadena)+1];
            strcpy(cadena, c.cadena);
        }
        else cadena = NULL;
    }
    return *this; // C1 = C2 = C3;
}
```

¿Qué pasa si no lo implementamos?

Hay que tener en cuenta la posibilidad de que se asigne un objeto a si mismo. **this**

14.2 Operadores Sobrecargados

Sobrecarga de Operadores Binarios

Además del operador + pueden sobrecargarse prácticamente todos los operadores:

+, -, *, /, %, ^, &, |, (,), <, >, <=, >=, <<, >>, ==, !=, &&, ||, =, +=. -=, *=, /=, %=, ^=, &=,

|=, <<=, >>=, [], (), ->, new y delete.

Los operadores =, [], () y -> sólo pueden sobrecargarse en el interior de clases.

14.2 Operadores Sobrecargados

Sobrecarga de operadores

```
class Tiempo
{
    ...
    bool operator>(Tiempo h); // >
    ...
};

bool Tiempo::operator>(Tiempo h)
{
    return (hora > h.hora ||
           (hora == h.hora && minuto
            > h.minuto));
}
...
if(Tiempo(1,32) > Tiempo(1,12))
    cout<<"1:32 mayor que 1:12"<< endl;
else
    cout<<"1:32 menor o igual que 1:12"
    << endl;
...

```

```
class Tiempo
{
    ...
    void operator+=(Tiempo h); // +=
    ...
};

void Tiempo::operator+=(Tiempo h)
{
    minuto += h.minuto;
    hora   += h.hora;
    while(minuto >= 60)
    {
        minuto -= 60;
        hora++;
    }
}
...
Ahora += Tiempo(1,32);
Ahora.Mostrar();
...

```

14.2 Operadores Sobrecargados

Cambiando el significado de los operadores

No es imprescindible mantener el significado de los operadores. Podemos implementar la suma de Tiempo para que en realidad sea la resta, pero acordaos que la idea es hacer más sencilla la programación.

En este ejemplo sobrecargamos >> para que devuelva el mayor de dos operandos.

Utilizamos el puntero this para usar el objeto actual en una comparación.

```
class Tiempo
{
    ....
    Tiempo operator>>(Tiempo h);
};
Tiempo Tiempo::operator>>(Tiempo h)
{
    if(*this > h) return *this; else return h;
}
T1 = Ahora >> Tiempo(13,43) >> T1 >> Tiempo(12,32);
```

```
// EQUIVALENTES
T1 = T1.operator+(Ahora);

T1 = Ahora + T1;
```

14.2 Operadores Sobrecargados

Clases con punteros

Cuando nuestras clases tienen punteros con memoria dinámica asociada, la sobrecarga de funciones y operadores puede complicarse un poco.

```
class Cadena
{...
    Cadena operator+(const Cadena &);
};
Cadena Cadena::operator+(const Cadena &c)
{
    Cadena temp;
    temp.cadena = new char[strlen(c.cadena)+strlen(cadena)+1];
    strcpy(temp.cadena, cadena);
    strcat(temp.cadena, c.cadena);
    return temp;
}

Cadena C1, C2("Primera parte");
C1 = C2 + " Segunda parte";
```

14.2 Operadores Sobrecargados

Clases con punteros

```
Cadena C1, C2("Primera parte");  
C1 = C2 + " Segunda parte";  
//C1.operator=(Cadena(C2.operator+(Cadena(" Segunda parte"))));
```

- 1) Se crea automáticamente un objeto temporal sin nombre para la cadena " Segunda parte". Y se llama al operador + del objeto C2.
- 2) Dentro del operador + se crea un objeto temporal: temp, reservamos memoria para la cadena que almacenará la concatenación de this->cadena y c.cadena, y le asignamos el valor de ambas cadenas, temp contiene la cadena: "Primera parte Segunda parte".
- 3) Retornamos el objeto temporal.
- 4) Ahora el objeto temporal temp se copia a otro objeto temporal sin nombre, y temp es destruido. Y el objeto temporal sin nombre se pasa como parámetro al operador de asignación.
- 5) Se asigna el objeto temporal sin nombre a C1, y se destruye.

Resumamos: el objeto temp se copia en un temporal sin nombre, y después se destruye, ¿qué pasa con el dato temp.cadena?, evidentemente también se destruye, pero el constructor copia por defecto ha copiado ese puntero, por lo tanto, también su cadena es destruida. El resultado es que C1 no recibe la suma de las cadenas.

14.2 Operadores Sobrecargados

Clases con punteros

```
Cadena(const Cadena &c) : cadena(NULL)
{ *this = c; }

Cadena &Cadena::operator=(const Cadena &c)
{
    if(this != &c)
    {
        delete[] cadena;
        if(c.cadena)
        {
            cadena = new char[strlen(c.cadena)+1];
            strcpy(cadena, c.cadena);
        }
        else cadena = NULL;
    }
    return *this;
}
```

Moraleja:

Cuando nuestras clases tengan datos miembro que sean punteros a memoria dinámica debemos sobrecargar siempre el constructor copia.

Nunca sabemos cuándo puede ser invocado sin que nos demos cuenta.

14.2 Operadores Sobrecargados

Sobrecarga de Operadores

Seguiremos con:
Sobrecarga de
Operadores Unarios