Décimoquinta Sesión

Metodologías y Técnicas de Programación II

Programación Orientada a Objeto (POO) C++

Sobrecarga de Operadores II

15.2 Sobrecarga de Operadores II

Sobrecarga de operadores unitarios (o unarios)

Son aquellos operadores que sólo requieren un operando.

- Asignación. =
- Incremento. ++
- Otros....

En los operadores binarios el primer argumento era el propio objeto de la clase donde se define el operador.

En los operadores unitarios:

- El operando es el propio objeto.
- No requieren operandos (dentro de una clase).

```
<tipo> operator<operador unitario>();
```

15.2 Sobrecarga de Operadores II

Sobrecarga de operadores unitarios

```
class Tiempo
{
...
Tiempo operator++();
...
};
Tiempo Tiempo::operator++()
{
    minuto++;
    while(minuto >= 60)
    {
       minuto -= 60;
       hora++;
    }
    return *this;
}
...
++T1;
```

15.2 Sobrecarga de Operadores II

Operadores unitarios sufijos

En el ejemplo anterior hemos visto preincremento. ¿Cómo es el postincremento? (Ver diferencia ++i , i++)

Regla: si se declara un parámetro para un operador ++ ó -- se sobrecargará la forma sufija del operador. El parámetro se ignorará, así que bastará con indicar el tipo.

Cuando se usa un operador en la forma sufijo dentro de una expresión, primero se usa el valor actual del objeto, y una vez evaluada la expresión, se aplica el operador.

Primero devolvemos el valor actual y luego incrementamos (Ver i++).

Si nosotros queremos que nuestro operador actúe igual deberemos usar un objeto temporal, y asignarle el valor actual del objeto.

Seguidamente aplicamos el operador al objeto actual y finalmente retornamos el objeto temporal.

15.2 Sobrecarga de Operadores II

Sobrecarga de operadores unitarios

```
class Tiempo
{
...
Tiempo operator++(); // Prefijo
Tiempo operator++(int); // Sufijo
...
};

Tiempo Tiempo::operator++()
{
    minuto++;
    while(minuto >= 60)
    {
       minuto -= 60;
       hora++;
    }
    return *this;
}
```

```
Tiempo Tiempo::operator++(int)
{
  //Constructor copia
  Tiempo temp(*this);
    minuto++;
    while(minuto >= 60)
    {
       minuto -= 60;
       hora++;
    }
    return temp;
}

T1.Mostrar();      // 01:01
(T1++).Mostrar(); // 01:01
T1.Mostrar();      // 01:02
(++T1).Mostrar(); // 01:03
T1.Mostrar();      // 01:03
```

15.2 Sobrecarga de Operadores II

Operadores de conversión de Tipos

Suponed que queremos poder hacer esto:

```
Tiempo T1(12,23);
unsigned int minutos = 432;
T1 += minutos;

Tiempo(int h=0, int m=0) : hora(h), minuto(m) {} // Constr.
```

En C++ se realizan conversiones implícitas entre los tipos básicos antes de operar con ellos, por ejemplo para sumar un int y un float, se convierte el entero a float.

Se utiliza el constructor que tenemos. Como sólo hay un parámetro en la llamada, el parámetro m toma el valor 0, y para el parámetroh se convierte el valor "minutos" de unsigned int a int.

El resultado es que se suman 432 horas, y nosotros queremos sumar 432 minutos.

15.2 Sobrecarga de Operadores II

Operadores de conversión de Tipos

Como nosotros queremos sumar a los minutos necesitamos crear un nuevo constructor.

```
Tiempo(unsigned int m) : hora(0), minuto(m)
{
  while(minuto >= 60)
{
      minuto -= 60;
      hora++;
  }
}
```

En general podremos hacer conversiones de tipo desde cualquier objeto a un objeto de nuestra clase sobrecargando el constructor. (Por ejemplo si consideramos float como milisegundos – Ejercicio).

¿Y al revés? Quiero asignar a un entero un objeto de tipo Tiempo.

```
int minutos = T1;  // Error de compilación.
```

15.2 Sobrecarga de Operadores II

Operadores de conversión de Tipos

```
int minutos = T1;  // Error de compilación.
```

Tenemos que diseñar nuestro operador de conversión de tipo, que se aplicará automáticamente en esas llamadas.

```
class Tiempo {
...
operator int();
...
operator int()
{
    return hora*60+minuto;
}
```

No necesitan que se especifique el tipo del valor de retorno, ya que este es precisamente <tipo>.
Al ser operadores unitarios, tampoco requieren argumentos, porque se aplican al propio objeto. **El tipo puede ser cualquier clase o estructura.**

15.2 Sobrecarga de Operadores II

Operador Indexación []

El operador [] se usa para acceder a valores de objetos de una determinada clase como si se tratase de arrays.

Los índices no tienen por qué ser de un tipo entero o enumerado. Ahora no existe esa limitación.

Mayor utilidad con estructuras dinámicas de datos:

- Listas y árboles.
- Arrays asociativos, donde los índices sean por ejemplo, palabras

Veamos un ejemplo. Supongamos que hacemos una clase para hacer un histograma de los valores de rand()/RAND_MAX, entre los márgenes de 0 a 0.0009, de 0.001 a 0.009, de 0.01 a 0.09 y de 0.1 a 1.

Lo que queremos es saber cuántos valores caen entre cada uno de los rangos.

15.2 Sobrecarga de Operadores II

Operador []

```
class Cuenta
{
   public:
    Cuenta()
{ for(int i = 0; i < 4;
  contador[i++] = 0); }
    int &operator[](double n);
    void Mostrar() const;
   private:
    int contador[4];
};
int &Cuenta::operator[](double n)
{
  if(n < 0.001) return contador[0];
  else if(n<0.01)
     return contador[1];
   else if(n<0.1)
        return contador[2];
   else return contador[3];
```

```
void Cuenta::Mostrar() const
{
   cout << "Entre 0 y 0.0009: " <<
  contador[0] << endl;
   cout << "Entre 0.0010 y 0.0099: "
  << contador[1] << endl;
   cout << "Entre 0.0100 y 0.0999: "
  << contador[2] << endl;
   cout << "Entre 0.1000 y 1.0000: "
  << contador[3] << endl;
}
int main()
{
   Cuenta C;
   for(int i = 0; i < 50000; i++)
      C[(double)rand()/RAND_MAX]++;
   C.Mostrar();
   return 0;
}
```

15.2 Sobrecarga de Operadores II

Operador Indexación []

En el ejemplo hemos usado un valor double como índice, pero igualmente podríamos haber usado una cadena o cualquier objeto que hubiésemos querido.

El tipo del valor de retorno de operador debe ser el del objeto que devuelve En nuestro caso, al tratarse de un contador, devolvemos un entero.

En realidad devolvemos una referencia a un entero, para poder aplicarle el operador de incremento al valor de retorno.

Cuando se combina el operador de indexación con estructuras dinámicas de datos como las listas, se puede trabajar con ellas como si se tratada de arrays de objetos, esto nos dará una gran potencia y claridad en el código de nuestros programas.

```
Lista mi_lista;
mi_lista[10].mostrar();
mi_lista[“Perez”].mostrar();
```

15.2 Sobrecarga de Operadores II

Operador llamada ()

Funciona igual que el operador [].

Admite más parámetros.

Permite usar un objeto de la clase para el que está definido como si fuera una función.

[] sólo admite un parámetro entre los corchetes.
() admite los que definamos.

```
Lista mi_lista;
......
mi_lista[5].mostrar();
......
cout << “DNI de Jose Perez” << mi_lista(“José”, “Pérez”) << endl;
......
```

15.2 Sobrecarga de Operadores II

Operador llamada ()

```
class Cuenta
{
...
int operator()(double n, double m);
...
};
```

Por supuesto, el número de parámetros, al igual que el tipo de retorno de la función depende de la decisión del programador.

```
int Cuenta::operator()(double n,
  double m)
{
    int i, j;
    if(n < 0.001) i = 0;
    else if(n < 0.01) i = 1;
    else if(n < 0.1) i = 2;
    else i = 3;
    if(m < 0.001) j = 0;
    else if(m < 0.01) j = 1;
    else if(m < 0.1) j = 2;
    else j = 3;
    if(contador[i] > contador[j])
        return contador[i];
    else
        return contador[j];
}
```

15.3 Repaso

Repaso

La sobrecarga de operadores es simplemente otra forma que tenemos para realizar llamadas a funciones.

Los argumentos para estas funciones no aparecen entre paréntesis, sino que rodean o siguen a los caracteres que siempre pensamos como operadores inalterables (+ , ++ , []).

En C++ es posible definir nuevos operadores que trabajen con clases.

Esta definicion es exactamente como la defición de una funcion ordinaria, excepto que el nombre de la funcion consiste en la palabra reservada **operator** seguida del operador.

Esta es la única diferencia.

El operador se convierte en una funcion como otra cualquiera que el compilador llama cuando ve el prototipo adecuado.

15.3 Repaso

Repaso

No hay razon para sobrecargar un operador excepto si eso hace al código implicado con la clase más sencillo e intuitivo de escribir y especialmente de leer.

Cuidado, porque es tentador usar operadores “por todas partes”.

Nunca podremos sobrecargar operadores así: `1 << 4;`

Para rematar:

Leed el capítulo 12 del libro “Pensar en C++”
Lo veremos en la parte de aspectos avanzados.

15.4 Ejercicios

Clase Complejo

Implementar la clase de los números complejos sobrecargando operadores de forma que podamos usarla como se expone en el main().

```
class Complejo
{
    private:
        int real_;
        int imaginaria_;
    public:
        .......
}

int main()
    Complejo a(1,1), b(2,2);
    Complejo c(b);

    cout << "++ c" << ++c;
    cout << "c --" << c--;
    cout << "Parte Imaginaria de c" << c[1];
    cout << "Parte Real c" << c[0];
    c = a + b;
    cout << c;
    b = a – c;
    cout << b;
```

15.4 Ejercicios

Clase Persona

Implementar la clase Persona sobrecargando los operadores =, + y << teniendo en cuenta que tienen punteros y que la suma de dos personas nos da otra persona cuyo nombre es la suma de los nombres de los sumandos y su altura es la mayor de las dos alturas de los sumandos.

```
class Persona
{
    private:
        int altura_;
        char* nombre_;
    public:
        .......
}
```

Tener en cuenta que hacen falta constructores, constructores copia y destructores.

Hacer lo necesario para poder asignar un entero a una persona de forma que el nombre es esos casos sea “Fabian” y la altura sea la de la asignación.

```
Persona p(“Pepe”,3);
p = 5;
cout << p << endl;
```