# Sobrecarga de Operadores

# Operadores

- Los operadores son en realidad funciones, pero con nombre diferente: son símbolos
- Los operandos que le pasamos a un operador son equivalentes a los argumentos que le pasamos a una función
- El nombre de un operador en C++ es operator <symbol>
- Los operadores se llaman de forma diferente a las funciones

# Sobrecarga de operadores

- Sobrecarga de operadores (*operator overloading*): todos los operadores ya están definidos, por lo que no podemos crear operadores nuevos, solamente redefinir su comportamiento
- Un operador puede ser definido como función miembro de una clase o como función no-miembro

# Llamadas a operadores

- Ejemplo: var1 + var2 (operador binario)
  - var1 es el operando 'lhs', var2 es el operando 'rhs'
  - Si definimos operator+() como una función miembro, var1 es el *objeto llamante* y var2 se pasa como argumento

```
Vector2D Vector2D::operator+(const Vector2D& right){...}
```

  - Si definimos operator+() como una función no-miembro, var1 y var2 se pasan como argumento

```
friend Vector2D operator+(const Vector2D& left,
                          const Vector2D& right) {...}
```

# Llamadas a operadores

- Ejemplo: ~var1 (operador unario)
  - var1 es el único operando
  - Si definimos operator~() como una función miembro, var1 es el *objeto llamante*

```
Vector2D Vector2D::operator~()const {...}
```

  - Si definimos operator~() como una función no-miembro, var1 se pasa como argumento

```
friend Vector2D operator~(const Vector2D& vec) {...}
```

# Operadores

- inserción y extracción: >>, <<
  - cout << var;
  - cin >> var;

# Operadores

- inserción y extracción: >>, <<
- aritméticos: +, -, *, /, %
  - var1 + var2
  - var1 - var2
  - var1 * var2
  - var1 / var2
  - var1 % var2

# Operadores

- inserción y extracción: >>, <<
- aritméticos: +, -, *, /, %
- asignación: =
  - var1 = var2;
  - var1 = var2 + var3;

# Operadores

- inserción y extracción: >>, <<
- aritméticos: +, -, *, /, %
- asignación: =
- asignación compuesta: +=, -=, *=, /=, %=
  - var1 += var2;
  - var1 -= var2;
  - var1 *= var2;
  - var1 /= var2;
  - var1 %= var2;

# Operadores

- inserción y extracción: >>, <<
- aritméticos: +, -, *, /, %
- asignación: =
- asignación compuesta: +=, -=, *=, /=, %=
- corchetes y paréntesis: [] and ()
  - object[index_pos] = var;
  - object(var);

# Operadores

- inserción y extracción: >>, <<
- aritméticos: +, -, *, /, %
- asignación: =
- asignación compuesta: +=, -=, *=, /=, %=
- corchetes y paréntesis: [] and ()
- y algunos más: relacionales, lógicos, de bits, incremento/decremento, de punteros, de memoria dinámica, etc…

# Operadores

- inserción y extracción: >>, <<
- aritméticos: +, -, *, /, %
- asignación: =
- asignación compuesta: +=, -=, *=, /=, %=
- corchetes y paréntesis: [] and ()
- y algunos más …
- Sobrecarga prohibida para: :: . .* ?: sizeof, y castings

# Reglas adicionales

- No se pueden crear operadores nuevos
- Para sobrecargar un operador, al menos un parámetro debe ser un tipo definido por el usuario
- No se puede cambiar el nº de args de un operador
- No se puede cambiar el orden de precedencia de los operadores al sobrecargarlos
- Algunos operadores solo permiten sobrecarga como funciones miembro, son los siguientes: = [] () ->
- Un operador sobrecargado no puede tener argumentos por defecto

# Consejos

- Debemos sobrecargar operadores de forma que tengan sentido (por ejemplo, no sobrecargar el operador ‘+’ para hacer una resta)
- Sobrecargar un operador como función miembro si modifica el objeto llamante; y como no-miembro si no lo modifica
- Definir pares de operadores simétricos en función uno de otro (por ejemplo, definir el operador ‘!=’ usando el operador ‘==‘ ya definido antes)

# Sobrecarga: Ejemplo

```
//fraction.h
...
class Fraction
{
  ...
  friend Fraction mult_fracs(const Fraction & lhs, const Fraction & rhs);
  ...
};
```

```
//fraction.cpp
...
Fraction mult_fracs(const Fraction & lhs, const Fraction & rhs)
{
    Fraction temp;
    temp.m_numerator = lhs.m_numerator * rhs.m_numerator;
    temp.m_denominator = lhs.m_denominator * rhs.m_denominator;
    return temp;
}
```

# Sobrecarga: Ejemplo

```
//fraction.h
...
class Fraction
{
  ...
  friend Fraction operator* (const Fraction & lhs, const Fraction & rhs);
  ...
};
```

```
//fraction.cpp
...
Fraction operator* (const Fraction & lhs, const Fraction & rhs)
{
    Fraction result(lhs);
    return (result *= rhs);
}
```

# Sobrecarga: Ejemplo

```
//fraction.h
...
class Fraction
{
  ...
  friend Fraction operator* (const Fraction & lhs, const Fraction & rhs);
  ...
};
```

```
//fraction.cpp
...
Fraction operator* (const Fraction & lhs, const Fraction & rhs)
{
    Fraction result(lhs);
    return (result *= rhs);
}
```

el operador *= debe estar definido

# Sobrecarga: Ejemplo

```
//fraction.h
...
class Fraction
{
  ...
  friend Fraction operator* (const Fraction & lhs, const Fraction & rhs);
  Fraction& operator*= (const Fraction & rhs);
  ...
};
```

```
//fraction.cpp
...
Fraction& Fraction::operator*= (const Fraction & rhs)
{
    m_numerator *= rhs.m_numerator;
    m_denominator *= rhs.m_denominator;
    return (*this);
}
```

# *this

Puntero a objeto Fraction

this

this es el puntero que apunta al objeto Fraction en memoria – **está apuntando al objeto llamante de este método**

Fraction

```
Fraction& Fraction::operator*= (const Fraction & rhs)
{
    m_numerator *= rhs.m_numerator;
    m_denominator *= rhs.m_denominator;
    return (*this);
}
```

# *this

Puntero a Fraction

this

*this es el objeto ejecutando la llamada a *=
***this es el objeto llamante**

Fraction

```
Fraction& Fraction::operator*= (const Fraction & rhs)
{
    m_numerator *= rhs.m_numerator;
    m_denominator *= rhs.m_denominator;
    return (*this);
}
```