FACULTAD DE INFORMÁTICA

Las clases como tipos de datos
(y más sobre métodos)

TEMA

Programación orientada a objetos — Unidad 3

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

FdI UCM

Contenido

FdI UCM

Clases como tipos de datos: una clase **Fraccion**

Las clases también son implementaciones de tipos de datos.

Vamos a ver como ejemplo una clase **Fraccion** que implemente un tipo de datos para representar fracciones.

Los atributos

Las fracciones se caracterizan por dos valores enteros que constituyen el numerador y el denominador de una división que no se realiza (es decir, no se obtiene el valor real resultado de la división). Así, como atributos necesitamos el **_numerador** y el **_denominador**:

```
class Fraccion {
private:
  int _numerador, _denominador;
};
```

Los atributos los ponemos en la sección privada

Clases como tipos de datos: una clase **Fraccion**

Un método auxiliar (privado)

Como queremos trabajar siempre con fracciones reducidas, incluimos un método que simplifique las fracciones. Como este método se utilizará internamente en los otros métodos para simplificar las fracciones después de haberse modificado su estado (haber cambiado el numerador o el denominador), será un método privado:

```
class Fraccion {
private:
  int _numerador, _denominador;
  void simplifica(); // método privado
};
```

Las fracciones siempre se dejarán simplificadas, por lo que no existe necesidad desde el exterior de disponer de la operación de simplificación. Por esto no se ofrece como servicio público.

Clases como tipos de datos: una clase **Fraccion**

Operaciones sobre el tipo

Para empezar, implementaremos las siguientes operaciones:

- ✓ Suma
- ✓ Resta
- ✓ Multiplicación
- ✓ División

Además, para cada atributo habrá accedente y mutador, e incluiremos también un método para mostrar en la pantalla el objeto fracción.

Clases como tipos de datos: una clase **Fraccion**

```
#include <iostream>
using namespace std;
class Fraccion {
public:
  Fraccion(int n = 0, int d = 1) // constructor
    : _numerador(n), _denominador(d) { }
  // Mutadores:
  void numerador(int i)
    { _numerador = i; simplifica(); }
  void denominador(int i)
    { _denominador = i; simplifica(); }
  // Accedentes:
  int numerador() const { return _numerador; }
  int denominador() const { return _denominador; }
  Fraccion mas(Fraccion) const;
  Fraccion menos(Fraccion) const;
```

(*continúa*)

Clases como tipos de datos: una clase **Fraccion**

```
  Fraccion por(Fraccion) const;
  Fraccion entre(Fraccion) const;
  void mostrar() const;
private:
  int _numerador, _denominador;
  void simplifica(); // método privado
};

void Fraccion::simplifica()
// Reduce (simplifica) la fracción
{ ... }

void Fraccion::mostrar() const
{
  cout << _numerador << '/' << _denominador;
}
```

(*continúa*)

Clases como tipos de datos: una clase **Fraccion**

```
Fraccion Fraccion::mas(Fraccion otro) const
{
  Fraccion tmp;
  tmp._numerador = _numerador * otro._denominador +
                   _denominador * otro._numerador;
  tmp._denominador = _denominador * otro._denominador;
  tmp.simplifica();
  return tmp;
}
```

Se tiene acceso a los atributos de los argumentos de esa misma clase en las funciones miembro

```
Fraccion Fraccion::menos(Fraccion otro) const
{
  Fraccion tmp;
  tmp._numerador = _numerador * otro._denominador -
                   _denominador * otro._numerador;
  tmp._denominador = _denominador * otro._denominador;
  tmp.simplifica();
  return tmp;
}
```

(*continúa*)

Clases como tipos de datos: una clase **Fraccion**

```
Fraccion Fraccion::por(Fraccion otro) const
{
  Fraccion tmp;
  tmp._numerador = _numerador * otro._numerador;
  tmp._denominador = _denominador * otro._denominador;
  tmp.simplifica();
  return tmp;
}

Fraccion Fraccion::entre(Fraccion otro) const
{
  Fraccion tmp;
  tmp._numerador = _numerador * otro._denominador;
  tmp._denominador = _denominador * otro._numerador;
  tmp.simplifica();
  return tmp;
}
```


Métodos y objetos parámetros/argumentos

Los métodos pueden tener definidos parámetros para aceptar objetos argumentos de determinadas clases. El *tipo* del parámetro será el nombre de la clase del objeto argumento que se espera:

```
Fraccion Fraccion::por(Fraccion otro) const
```

Si se trata de un parámetro por valor, al comenzar la ejecución del método se crea el objeto parámetro como copia del objeto argumento que se haya utilizado en el paso de mensaje.
Si se trata de un parámetro por referencia, el parámetro se refiere directamente al objeto argumento (pudiendo *mutarlo*).

Si la clase del parámetro es la misma que la del método que se implementa, en el cuerpo del método, además de a lo público, se tiene garantizado el acceso a lo privado de ese objeto (atributos y métodos privados), simplemente cualificando con el nombre del parámetro.

Métodos y objetos parámetros/argumentos

Así, para acceder a un atributo del parámetro, basta preceder el nombre del atributo por el parámetro y un punto cualificador:

```
tmp._numerador = _numerador * otro._numerador;
```

Para pasar un mensaje al parámetro, basta preceder el nombre del mensaje por el parámetro y un punto cualificador:

```
otro.simplifica();
```

Si la clase del parámetro es distinta de la clase en la que se encuentra el método que se implementa, el objeto parámetro sólo se puede manipular solicitándole servicios (pasándole mensajes que se correspondan con métodos públicos de su clase).

Se usa la sintaxis habitual: *parámetro*.*mensaje*...

Métodos y objetos locales

Los métodos pueden tener definidos objetos locales:

```
Fraccion Fraccion::por(Fraccion otro) const
{
  Fraccion tmp;
```

Los objetos locales se manejan de la misma forma que los parámetros que aceptan objetos argumentos (cualificando).

Si la clase del objeto local es la misma que del método que se implementa, en el cuerpo del método, además de a lo público, se tiene garantizado el acceso a lo privado del objeto local (atributos y métodos privados), simplemente cualificando con el nombre del parámetro:

```
tmp._denominador = _denominador * otro._denominador;
tmp.simplifica();
```


Métodos que devuelven objetos

Los métodos pueden devolver objetos. Basta con que el *tipo* de la función miembro sea un nombre de clase:

```
Fraccion Fraccion::por(Fraccion otro) const
```

En estos casos, resulta necesario que el método devuelva siempre un objeto de la clase especificada como *tipo* de la función miembro. La devolución del objeto se realiza con el método usual de devolución de resultados de las funciones: la instrucción **return**:

```
Fraccion Fraccion::por(Fraccion otro) const
{
  Fraccion tmp;
  ...
  return tmp;
}
```


Métodos y objetos: un resumen

En los métodos se pueden identificar distintas categorías de objetos:

- ✓ Objeto receptor del mensaje:
  El que ha provocado la ejecución del método.
  El método se ejecuta *sobre* él.
  Acceso total a sus atributos y métodos sin cualificar.
- ✓ Objetos parámetros:
  Hacen referencia a los objetos argumentos (o a copias de ellos).
  Se usan como cualquier otro objeto.
  Si son de la misma clase, se tiene acceso a lo privado.
- ✓ Objetos locales:
  Declarados en el cuerpo de los métodos.
  Se usan como cualquier otro objeto.
  Si son de la misma clase, se tiene acceso a lo privado.

Tanto los objetos locales como los objetos parámetros son objetos temporales, creándose al comenzar la ejecución del método y destruyéndose al terminar la ejecución del método.

this: una referencia al objeto receptor del mensaje

Los métodos siempre tienen definido un primer parámetro *secreto*, que no se hace figurar en la lista de parámetros y que siempre tiene la misma forma: *puntero* a un objeto de la clase.

El nombre de este parámetro siempre es **this** (palabra reservada).

El parámetro se añade a la lista de parámetros de forma automática durante el proceso de compilación.

El parámetro *fantasma* recibe siempre un puntero al objeto receptor del mensaje que ha provocado la ejecución del método (el ejemplar concreto de la clase que ha recibido el mensaje).

El código que pasa el argumento al parámetro **this** también se añade de forma automática durante el proceso de compilación.

Así, en el código de los métodos podemos utilizar ese parámetro **this** para acceder a los atributos del objeto receptor o hacer que el objeto receptor se pase mensajes a sí mismo.

FdI UCM — `this`: una referencia al objeto receptor del mensaje

Como **this** es un puntero al objeto receptor del mensaje, para acceder a sus miembros (atributos/métodos) se usa el operador flecha (**->**), en lugar del operador punto (más adelante veremos con detenimiento los punteros).

Así, si queremos acceder a un atributo del objeto receptor o hacer que se envíe un mensaje a sí mismo, colocaremos **this->** delante del nombre del atributo o del mensaje:

```
void numerador(int i)
{ this->_numerador = i; this->simplifica(); }
```

Sin embargo, el compilador usa la siguiente regla al compilar el código de un método:

Cualquier nombre de atributo o de mensaje de la clase que no aparezca cualificado (es decir, que no vaya precedido por objeto• *u* objeto→*), se asume que va implícitamente cualificado con* **this->**

FdI UCM — `this`: una referencia al objeto receptor del mensaje

La regla anterior hace que no resulte necesario cualificar con **this->**, por lo que no se suele hacer normalmente:

```
void numerador(int i)
{ _numerador = i; simplifica(); }
```

Se puede omitir **this->**

Pero, aunque para acceder a los atributos o para pasarse mensajes no resulte necesario **this->**, y por eso no figure normalmente, hay ocasiones en las que resulta imprescindible el uso de **this**, como por ejemplo cuando el método ha de devolver una copia del objeto receptor (o el propio objeto receptor):

```
return *this;
```

Más adelante hablaremos más del mecanismo de devolución de objetos.

FdI UCM — Uso de la clase `Fraccion`

```
// Prueba de la clase Fraccion
#include <iostream>
using namespace std;

Declaración e implementación de la clase Fraccion

int main()
{
  Fraccion f1, f2, f3;
  f1.numerador(2);
  f1.denominador(3);
  f1.mostrar();
  cout << endl;
  f2.numerador(5);
  f2.denominador(7);
  f2.mostrar(); cout << endl;
```

```
2/3
5/7
```


FdI UCM — Uso de la clase `Fraccion`

```
  cout << "Numerador: " << f2.numerador()
     << endl;
  cout << "Denominador: " << f2.denominador()
     << endl;
  f3 = f1.mas(f2);
  f3.mostrar(); cout << endl;
  f3 = f1.menos(f2);
  f3.mostrar(); cout << endl;
  f3 = f1.por(f2);
  f3.mostrar(); cout << endl;
  f3 = f1.entre(f2);
  f3.mostrar(); cout << endl;

  return 0;
}
```

```
2/3
5/7
Numerador: 5
Denominador: 7
29/21
-1/21
10/21
14/15
```

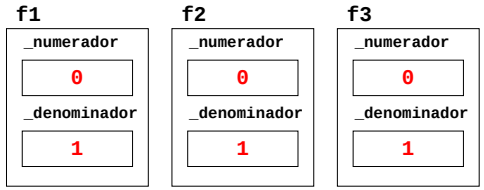
Estudio de la ejecución

```
int main()
{
  Fraccion f1, f2, f3;
```

El *objeto aplicación* crea tres objetos de la clase **Fraccion**, identificándolos como **f1**, **f2** y **f3**.

Se ejecuta automáticamente el constructor sobre cada uno de ellos, inicializándose a cero los atributos.

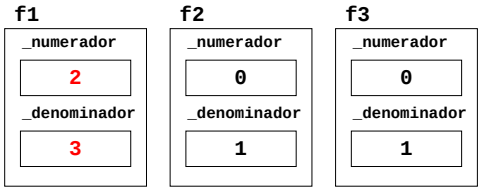
```
Fraccion(int n = 0, int d = 1) : _numerador(n), _denominador(d) { }
```



Estudio de la ejecución

```
int main()
{
  Fraccion f1, f2, f3;
  f1.numerador(2);
  f1.denominador(3);
  f1.mostrar();
```

```
void numerador(int i)
{ _numerador = i; simplifica(); }
void denominador(int i)
{ _denominador = i; simplifica(); }
```



El *objeto aplicación* envía el mensaje **numerador()** al objeto **f1** pasando como argumento el valor **2**.

El *objeto aplicación* envía el mensaje **denominador()** al objeto **f1** pasando como argumento el valor **3**.

En la ejecución de esos dos métodos se hace que el objeto receptor (**f1**) se pase el mensaje **simplifica()** a sí mismo.

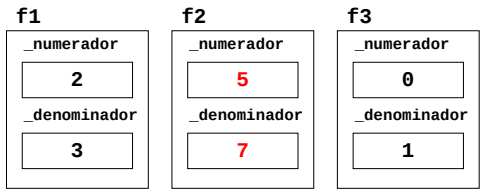
El *objeto aplicación* envía el mensaje **mostrar()** al objeto **f1**, mostrándose en la pantalla la representación del fraccional (**2/3**).

Estudio de la ejecución

```
f1.numerador(2);
f1.denominador(3);
f1.mostrar();
cout << endl;
f2.numerador(5);
f2.denominador(7);
f2.mostrar(); cout << endl;
cout << "Numerador: " << f2.numerador()
   << endl;
cout << "Denominador: " << f2.denominador()
   << endl;
```

A continuación se lleva a cabo un proceso similar con el objeto **f2**.

Y sigue la ejecución mostrando el numerador y el denominador de **f2** por separado.



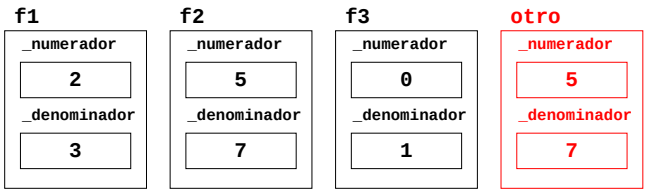
Estudio de la ejecución

```
f3 = f1.mas(f2);
```

El *objeto aplicación* envía el mensaje **mas()** al objeto **f1** pasando como argumento el objeto **f2**.

```
Fraccion Fraccion::mas(Fraccion otro) const
{
  Fraccion tmp;
  tmp._numerador = _numerador * otro._denominador +
                   _denominador * otro._numerador;
  tmp._denominador = _denominador * otro._denominador;
  tmp.simplifica();
  return tmp;
}
```

Se crea el objeto parámetro **otro** y se copia en él el objeto argumento (**f2**).

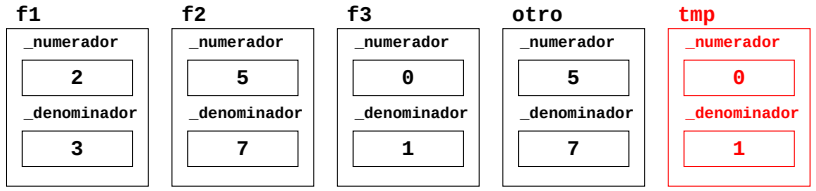


Estudio de la ejecución

```
f3 = f1.mas(f2);
```

```
Fraccion Fraccion::mas(Fraccion otro) const
{
  Fraccion tmp;
  tmp._numerador = _numerador * otro._denominador +
                   _denominador * otro._numerador;
  tmp._denominador = _denominador * otro._denominador;
  tmp.simplifica();
  return tmp;
}
```

Se crea el objeto local **tmp**.

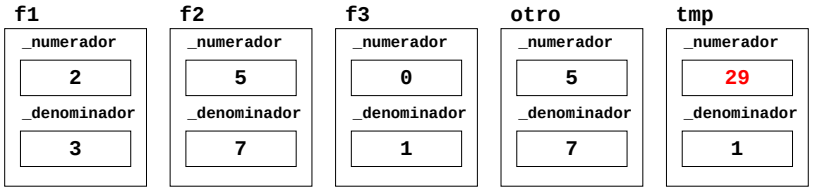


Estudio de la ejecución

```
f3 = f1.mas(f2);
```

```
Fraccion Fraccion::mas(Fraccion otro) const
{
  Fraccion tmp;
  tmp._numerador = _numerador * otro._denominador +
                   _denominador * otro._numerador;
  tmp._denominador = _denominador * otro._denominador;
  tmp.simplifica();
  return tmp;
}
```

Se asigna al atributo **_numerador** del objeto local **tmp** un valor calculado con los atributos del objeto receptor y del objeto argumento identificado como **otro**.

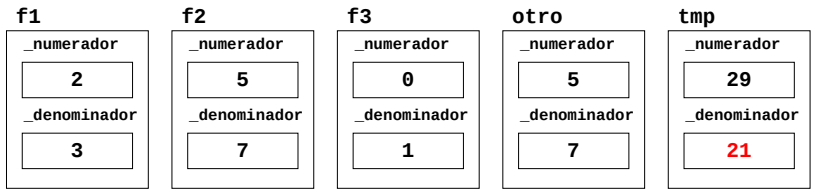


Estudio de la ejecución

```
f3 = f1.mas(f2);
```

```
Fraccion Fraccion::mas(Fraccion otro) const
{
  Fraccion tmp;
  tmp._numerador = _numerador * otro._denominador +
                   _denominador * otro._numerador;
  tmp._denominador = _denominador * otro._denominador;
  tmp.simplifica();
  return tmp;
}
```

Se asigna al atributo **_denominador** del objeto local **tmp** un valor calculado con los atributos del objeto receptor y del objeto argumento identificado como **otro**.

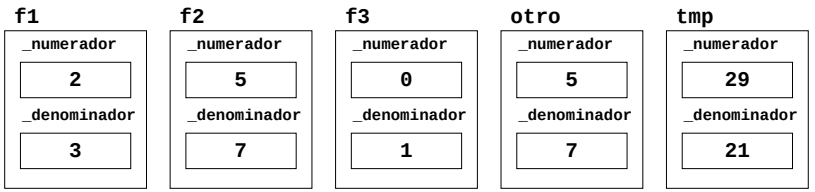


Estudio de la ejecución

```
f3 = f1.mas(f2);
```

```
  ...
  tmp._denominador = _denominador * otro._denominador;
  tmp.simplifica();
  return tmp;
}
```

El objeto **f1** (el receptor del mensaje **mas()** que provocó la ejecución de este método) pasa el mensaje **simplifica()** al objeto local **tmp** (encadenamiento de mensajes).
Como la fracción no se puede reducir más, sus atributos no cambian.

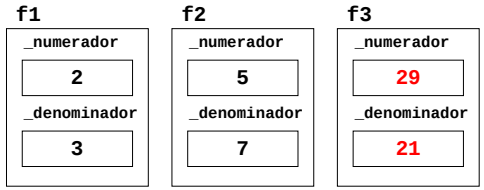


Estudio de la ejecución

```
f3 = f1.mas(f2);
```

```
  ...
  tmp._denominador = _denominador * otro._denominador;
  tmp.simplifica();
  return tmp;
}
```

Termina la ejecución del método **mas()**.
Se devuelve el objeto **tmp** como resultado de la ejecución del método.
De vuelta en el programa principal, se copia ese objeto devuelto en el objeto **f3**.
También se *destruyen* los objetos temporales del método (local y parámetro).

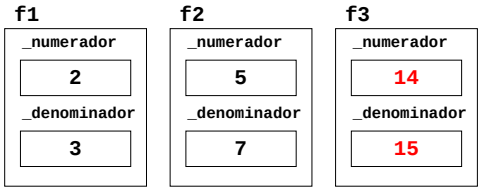


Estudio de la ejecución

```
f3 = f1.mas(f2);
f3.mostrar(); cout << endl;
f3 = f1.menos(f2);
f3.mostrar(); cout << endl;
f3 = f1.por(f2);
f3.mostrar(); cout << endl;
f3 = f1.entre(f2);
f3.mostrar(); cout << endl;
```

La ejecución del resto del programa es similar a lo que hemos visto hasta ahora.

Aparte de los pasos de mensaje **mostrar()**, se trata de operaciones que se ejecutan de forma similar a como hemos visto que se ejecuta el método **mas()**.



Estudio de la ejecución

```
  f3 = f1.mas(f2);
  f3.mostrar(); cout << endl;
  f3 = f1.menos(f2);
  f3.mostrar(); cout << endl;
  f3 = f1.por(f2);
  f3.mostrar(); cout << endl;
  f3 = f1.entre(f2);
  f3.mostrar(); cout << endl;
  return 0;
}
```

Al ejecutarse la instrucción **return** se destruyen todos los objetos del programa principal.

Métodos operadores

Las operaciones sobre fracciones las realizamos de forma *extraña* porque su sintaxis es la normal para pasos de mensajes que se corresponden con funciones miembro *normales*.

Por ejemplo, para sumar dos fracciones ponemos algo como:

```
f3 = f1.mas(f2);
```

siendo **f1**, **f2** y **f3** objetos de la clase **Fraccion**.

Sintaxis de operador para pasos de mensajes

Sería mucho más natural expresar la suma anterior como:

```
f3 = f1 + f2;
```

Para poder hacerlo, debemos definir el servicio con un método operador (función miembro operadora):

```
Fraccion operator+(Fraccion);
```

La clase **Fraccion** con métodos operadores

```
class Fraccion {
public:
  Fraccion(int n = 0, int d = 1) // constructor
    : _numerador(n), _denominador(d) { }
  void numerador(int i) { _numerador=i; simplifica(); }
  void denominador(int i) { _denominador=i; simplifica(); }
  int numerador() const { return _numerador; }
  int denominador() const { return _denominador; }
  Fraccion operator+(Fraccion) const;
  Fraccion operator-(Fraccion) const;
  Fraccion operator*(Fraccion) const;
  Fraccion operator/(Fraccion) const;
  void mostrar() const;
private:
  int _numerador, _denominador;
  void simplifica(); // método privado
};
```


Implementación de los métodos operadores

¡¡¡ La implementación de las funciones es exactamente la misma !!!

Sólo cambia la cabecera.

Por ejemplo, la suma quedará así:

```
Fraccion Fraccion::operator+(Fraccion otro) const
{
  Fraccion tmp;
  tmp._numerador = _numerador * otro._denominador +
                   _denominador * otro._numerador;
  tmp._denominador = _denominador * otro._denominador;
  tmp.simplifica();
  return tmp;
}
```


Cuando los métodos están implementados con funciones operadoras, no se usa la misma sintaxis para los pasos de mensaje que cuando están implementados con funciones *normales*.

Se utiliza la sintaxis habitual de los operadores (notación infija):

objeto_receptor operador objeto_argumento

Ahora podemos enviar los mensajes **+**, **-**, ***** y **/** a los objetos de la clase **Fraccion** de una forma más natural. Por ejemplo:

```
f3 = f1 + f2;
```

El objeto **f1** recibe el mensaje **+**, pasándosele el objeto **f2** como argumento.

Se ejecuta el método **+** de la clase **Fraccion**, que devuelve como resultado otro objeto **Fraccion**. En este caso el objeto que devuelve se asigna al objeto **f3** (se copia en él).

Más operaciones para el tipo **Fraccion**

```
class Fraccion {
public:
  Fraccion(int n = 0, int d = 1) // constructor
    : _numerador(n), _denominador(d) { }
  void numerador(int i) { _numerador=i; simplifica(); }
  void denominador(int i) { _denominador=i; simplifica(); }
  int numerador() const { return _numerador; }
  int denominador() const { return _denominador; }
  // Operadores aritméticos básicos:
  Fraccion operator+(Fraccion) const;
  Fraccion operator-(Fraccion) const;
  Fraccion operator*(Fraccion) const;
  Fraccion operator/(Fraccion) const;
  // Operadores relacionales:
  bool operator==(Fraccion) const;
  bool operator!=(Fraccion) const;
```

(*continúa*)

```
  bool operator<(Fraccion) const;
  bool operator>(Fraccion) const;
  bool operator<=(Fraccion) const;
  bool operator>=(Fraccion) const;
  // Más operadores aritméticos:
  Fraccion& operator++(); // incremento
  Fraccion& operator--(); // decremento
  // Abreviaturas para objeto = objeto operador otro:
  //                   objeto operador= otro
  Fraccion& operator+=(Fraccion);
  Fraccion& operator-=(Fraccion);
  Fraccion& operator*=(Fraccion);
  Fraccion& operator/=(Fraccion);
  void mostrar() const;
private:
  int _numerador, _denominador;
  void simplifica(); // método privado
};
```


Implementación de los operadores relacionales

Para los operadores de igualdad y desigualdad basta con comparar los productos de numerador de una fracción por denominador de la otra fracción. Es decir, si sus *valores* son iguales o no (no basta con comprobar si son iguales sus numeradores y denominadores).

Una fracción es el objeto receptor y la otra el objeto parámetro:

```
bool Fraccion::operator==(Fraccion otro) const
{
  return (_numerador * otro._denominador) ==
         (_denominador * otro._numerador);
}

bool Fraccion::operator!=(Fraccion otro) const
{
  return (_numerador * otro._denominador) !=
         (_denominador * otro._numerador);
}
```


Para los operadores <, <=, > y >= es más fácil obtener el *valor* de las fracciones (realizar realmente la división del numerador entre el denominador) y comparar los dos valores. Se necesitan moldes para que no se realicen divisiones enteras:

```
bool Fraccion::operator<(Fraccion otro) const {
  double izq, der;
  izq = double(_numerador) / double(_denominador);
  der = double(otro._numerador) / double(otro._denominador);
  return izq < der;
}

bool Fraccion::operator<=(Fraccion otro) const     {
  double izq, der;
  izq = double(_numerador) / double(_denominador);
  der = double(otro._numerador) / double(otro._denominador);
  return izq <= der;
}
...
```


Operadores que modifican el objeto receptor

Los operadores que modifican el objeto receptor han de devolver el propio objeto como una referencia:

```
Fraccion& Fraccion::operator++() {
  _numerador++;
  _denominador++;
  simplifica();
  return *this;
}

Fraccion& Fraccion::operator+=(Fraccion otro) {
  _numerador = _numerador * otro._denominador +
               _denominador * otro._numerador;
  _denominador = _denominador * otro._denominador;
  simplifica();
  return *this;
}
```

Devuelve el propio objeto receptor del mensaje

Devuelve una referencia a objeto de la clase

Obviamente no se puede tratar de métodos **const**

Devolución de objetos en los métodos

Acabamos de ver que los métodos operadores que modifican el objeto receptor han de devolver el propio objeto como una referencia.

Un ejemplo es el método operador incremento anterior:

```
Fraccion& Fraccion::operator++() {
  _numerador++;
  _denominador++;
  simplifica();
  return *this;
}
```

Realmente, el método deja modificado el objeto receptor (incrementados en uno su numerador y su denominador) sin necesidad de que se devuelva el objeto.

El motivo por el que, además, se debe devolver el propio objeto receptor como referencia tiene que ver con lo que se puede tener que hacer con el objeto después de aplicarle esta operación.

Devolución de objetos en los métodos

Métodos operadores que no devuelven nada (**void**)

Podemos pensar en otras formas alternativas de implementar el método operador incremento. Por ejemplo, podemos modificar el objeto receptor pero no devolver nada:

```
void Fraccion::operator++()
{ _numerador++; _denominador++; simplifica(); }
```

Efectivamente, se modifica adecuadamente el objeto receptor.

Este operador se puede utilizar para incrementar una **Fraccion**:

```
frac1++;
```

pero se impide *encadenar operaciones*:

```
(frac1++)++; // ERROR: nada que incrementar
frac2 = frac1++; // ERROR: nada que asignar
```

Como no se devuelve objeto, no hay objeto que utilizar en la siguiente operación.

Devolución de objetos en los métodos

Métodos operadores que devuelven nuevos objetos

Otra forma de implementar el método operador incremento podría ser con una función que devuelva un nuevo objeto. El nuevo objeto deberá ser igual al objeto receptor incrementado:

```
Fraccion Fraccion::operator++() {
  Fraccion tmp = *this; // *this es el objeto receptor
  tmp._numerador++;
  tmp._denominador++;
  tmp.simplifica();
  return tmp;
}
```

El problema ahora es que no se modifica el objeto receptor.

Pero esto se puede solucionar fácilmente. Se puede implementar la función de otra forma: incrementando primero el objeto receptor y devolviendo luego una copia suya ...

Devolución de objetos en los métodos

```
Fraccion Fraccion::operator++() {
  _numerador++; _denominador++; simplifica();
  Fraccion tmp = *this; // *this es el objeto receptor
  return tmp;
}
```

Ahora sí se modifica el objeto receptor, pero seguimos teniendo el problema de que lo que se devuelve es una copia.

El operador se puede utilizar para incrementar una **Fraccion** y se puede asignar el resultado a otra **Fraccion**:

```
frac2 = frac1++;
```

Pero si se aplica a continuación otra operación que también modifique el objeto (otro incremento, por ejemplo):

```
(frac1++)++;
```

no se obtiene el resultado deseado, ya que el segundo incremento se realiza sobre la copia que devuelve el primero.

Devolución de objetos en los métodos

Métodos operadores que devuelven el objeto receptor

Si se devuelve el propio objeto receptor (necesariamente como una referencia, ya que **this** es un puntero al mismo), todo funciona:

```
Fraccion& Fraccion::operator++() {
  _numerador++; _denominador++; simplifica();
  return *this; // *this es el objeto receptor
}
```

Se modifica el objeto receptor y se devuelve el mismo como resultado de la ejecución del método operador, de forma que cualquier nueva operación que se efectúe sobre el resultado de esta operación, se efectúa sobre ese mismo objeto:

```
frac2 = frac1++; // sin problemas
(frac1++)++; // sin problemas
```

Como lo que se devuelve es una referencia (que se construye al volver para apuntar al objeto devuelto) y **this** es un puntero, se debe devolver lo apuntado por **this** (el objeto receptor).

Devolución de objetos en los métodos

Comparación de las alternativas

```
void Fraccion::operator++() {
  _numerador++; _denominador++;
  simplifica();
}
```

SI

```
Fraccion Fraccion::operator++() {
  _numerador++; _denominador++;
  simplifica();
  Fraccion tmp = *this;
  return tmp;
}
```

SI

¿ Modifica el objeto receptor ?

```
Fraccion& Fraccion::operator++() {
  _numerador++; _denominador++;
  simplifica();
  return *this;
}
```

SI

Devolución de objetos en los métodos

Comparación de las alternativas

```
void Fraccion::operator++() {
  _numerador++; _denominador++;
  simplifica();
}
```

NO

```
Fraccion Fraccion::operator++() {
  _numerador++; _denominador++;
  simplifica();
  Fraccion tmp = *this;
  return tmp;
}
```

SI

¿ Devuelve objeto ?

```
Fraccion& Fraccion::operator++() {
  _numerador++; _denominador++;
  simplifica();
  return *this;
}
```

SI

Devolución de objetos en los métodos

Comparación de las alternativas

```
Fraccion Fraccion::operator++() {
  _numerador++; _denominador++;
  simplifica();
  Fraccion tmp = *this;
  return tmp;
}
```

Devuelve COPIA del objeto receptor

```
Fraccion& Fraccion::operator++() {
  _numerador++; _denominador++;
  simplifica();
  return *this;
}
```

Devuelve el propio objeto receptor