FACULTAD DE INFORMÁTICA

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

SOLUCIONES DEL TALLER

Programación orientada a objetos — Unidad 3

Autor: Luis Hernández Yáñez

Un mundo de objetos

Objetos de la clase **Fraccion** (*resolución individual*)

Teniendo en cuenta que disponemos de la clase **Fraccion** vista en la Unidad, determina cuántos objetos se crean en total en el siguiente programa:

```
#include "fraccion.h"

int main() {
  Fraccion f1(4,6), f2(7,3), f3(3,5);
  Fraccion f4(f1);
  f4 = f2;
  f4 = f1 + f2;
  f1++;

  return 0;
}
```

Total de objetos

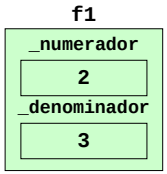
Antes de comenzar... 0

Un mundo de objetos

```
Fraccion f1(4,6), f2(7,3), f3(3,5);
```

```
Fraccion::Fraccion(int n, int d)
  : _numerador(n), _denominador(d) {
  simplifica();
}
```

simplifica() ni acepta ni devuelve objetos; tampoco usa objetos locales.



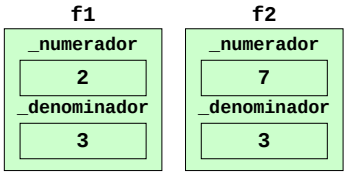
Total de objetos

1

Un mundo de objetos

```
Fraccion f1(4,6), f2(7,3), f3(3,5);
```

```
Fraccion::Fraccion(int n, int d)
  : _numerador(n), _denominador(d) {
  simplifica();
}
```



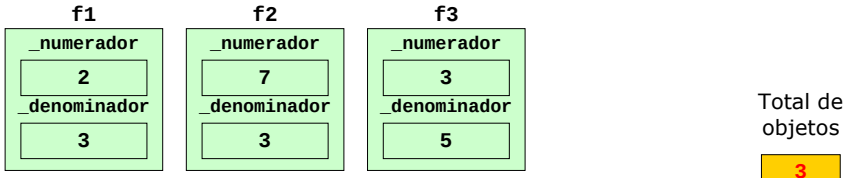
Total de objetos

2

Un mundo de objetos

```
Fraccion f1(4,6), f2(7,3), f3(3,5);
```

```
Fraccion::Fraccion(int n, int d)
  : _numerador(n), _denominador(d) {
  simplifica();
}
```



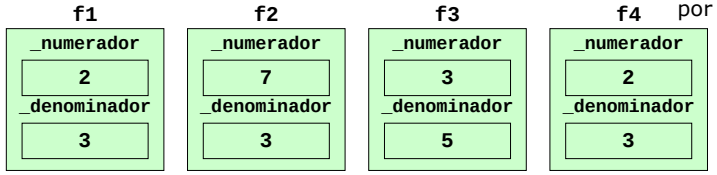
Total de objetos: 3

Un mundo de objetos

```
Fraccion f4(f1);
```

Paso por variable (referencia); se usa directamente el argumento (`f1`).

```
Fraccion::Fraccion(const Fraccion& otro) {
  _numerador = otro._numerador;
  _denominador = otro._denominador;
}
```



Construido por copia

Total de objetos: 4

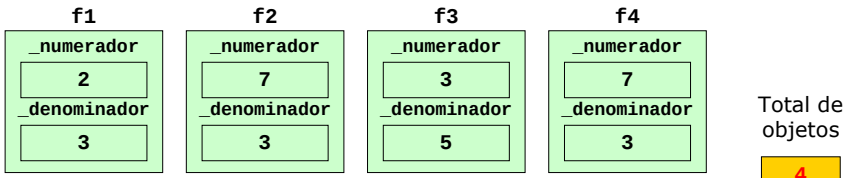
Un mundo de objetos

```
f4 = f2;
```

Paso por variable (referencia); se usa directamente el argumento (`f2`).

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

Devolución por variable (referencia); se devuelve el objeto receptor (`f4`).



Total de objetos: 4

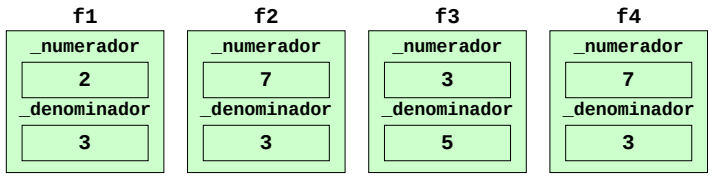
Un mundo de objetos

```
f4 = f1 + f2;
```

Paso por valor; se crea una copia del argumento (`f2`).

```
Fraccion Fraccion::operator+(Fraccion otro) const
{
```

Construido por copia

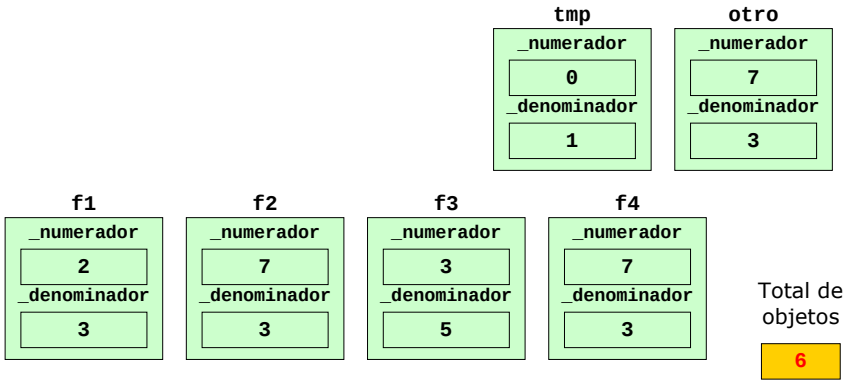


Total de objetos: 5

Un mundo de objetos

```
f4 = f1 + f2;
```

```
Fraccion Fraccion::operator+(Fraccion otro) const
{
  Fraccion tmp;
```

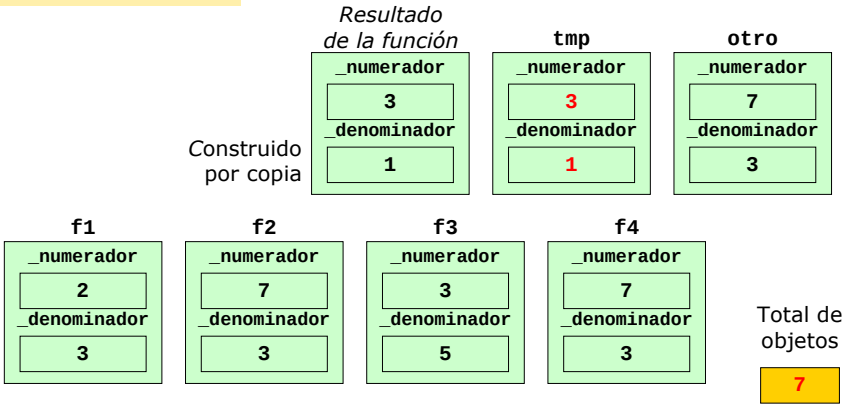


Un mundo de objetos

```
f4 = f1 + f2;
```

```
  ...
  return tmp;
}
```

Devolución por valor; se devuelve una copia de **tmp**.

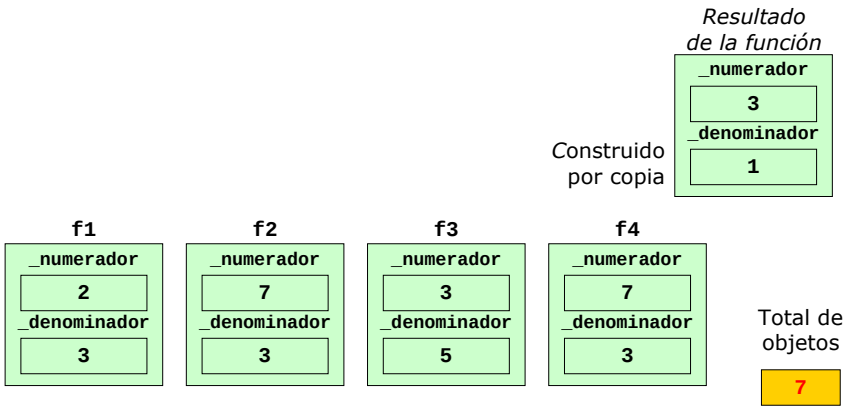


Un mundo de objetos

```
f4 = f1 + f2;
```

```
  ...
}
```

Se destruyen los objetos temporales (**otro** y **tmp**)

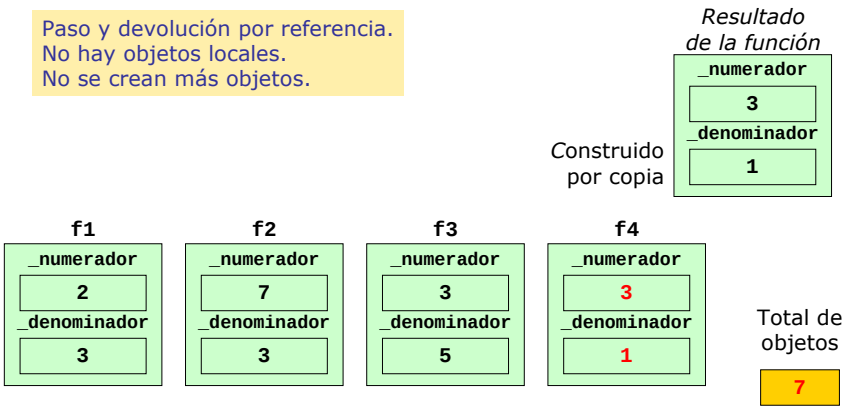


Un mundo de objetos

```
f4 = f1 + f2;
```

```
Fraccion& Fraccion::operator=(const Fraccion& otro)
```

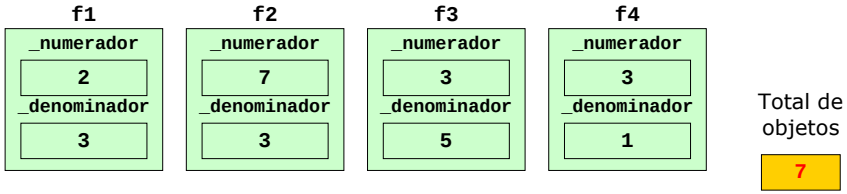
Paso y devolución por referencia.
No hay objetos locales.
No se crean más objetos.



Un mundo de objetos

```
f4 = f1 + f2;
```

Ya no se necesita el objeto devuelto por la función operadora +, por lo que se destruye.



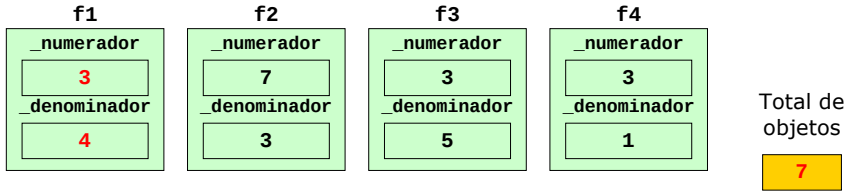
Un mundo de objetos

```
f1++;
```

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

Devolución por variable (referencia); se devuelve el objeto receptor (`f1`).

```
  return 0; // Se destruyen f4, f3, f2 y f1 (en ese orden)
}
```



Un mundo de objetos

Constatación empírica

Si quieres comprobar cómo se van creando, copiando y destruyendo los objetos, puedes añadir salidas por pantalla en los constructores, el operador de asignación y el destructor que te vayan informando de lo que va sucediendo:

```
Fraccion::Fraccion(int n, int d) : _numerador(n), _denominador(d) {
  simplifica();
  cout << "Construyo " << _numerador << "/" << _denominador << endl;
}
Fraccion::Fraccion(const Fraccion& otro) {
  _numerador = otro._numerador; _denominador = otro._denominador;
  cout << "Construyo copia " << _numerador << "/" << _denominador << endl;
}
Fraccion& Fraccion::operator=(const Fraccion& otro) {
  _numerador = otro._numerador; _denominador = otro._denominador;
  cout << "Copio " << _numerador << "/" << _denominador << endl;
  return *this;
}
Fraccion::~Fraccion() {
  cout << "Destruyo " << _numerador << "/" << _denominador << endl;
}
```


Ejercicio práctico

Completando más la clase (*resolución en grupo*)

A partir de la clase **`Contador3`** del taller de la semana anterior, hay que crear otra clase, **`Contador4`**, que sea igual que la anterior pero con estas nuevas características:

- ✓ Las operaciones de incremento/decremento con forma de operadores (++/--).
- ✓ Operadores + y - para suma y resta de contadores.
- ✓ Operadores relacionales (los seis).

Y, como siempre, se ha de probar la nueva clase en una función **`main()`**.

Ejercicio práctico

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

class Contador4 {
public:
  Contador4(int = 0);
  Contador4(const Contador4&);
  Contador4& operator=(const Contador4&);
  ~Contador4();
  Contador4& operator++();
  Contador4& operator--();
  Contador4 operator+(Contador4) const;
  Contador4 operator-(Contador4) const;
  bool operator==(Contador4) const;
  bool operator!=(Contador4) const;
  bool operator<(Contador4) const;
  bool operator<=(Contador4) const;
  bool operator>(Contador4) const;
  bool operator>=(Contador4) const;
  void mostrar() const;
```

continúa

Ejercicio práctico

```
private:
  int _cont;
};

Contador4::Contador4(int val){
  _cont = (val > 0) ? val : 0;
}

Contador4::Contador4(const Contador4& otro) {
  _cont = otro._cont;
}

Contador4& Contador4::operator=(const Contador4& otro) {
  _cont = otro._cont;
  return *this;
}

Contador4::~Contador4() { }
```

continúa

Ejercicio práctico

```
Contador4& Contador4::operator++() {
  _cont++;
  return *this;
}

Contador4& Contador4::operator--() {
  if(_cont>0) _cont--;
  return *this;
}

Contador4 Contador4::operator+(Contador4 otro) const {
  Contador4 tmp;
  tmp._cont = _cont + otro._cont;
  return tmp;
}

Contador4 Contador4::operator-(Contador4 otro) const {
  Contador4 tmp;
  tmp._cont = _cont - otro._cont;
  if(tmp._cont < 0) tmp._cont = 0;
  return tmp;
}
```

continúa

Ejercicio práctico

```
bool Contador4::operator==(Contador4 otro) const {
  return _cont == otro._cont;
}

bool Contador4::operator!=(Contador4 otro) const {
  return _cont != otro._cont;
}

bool Contador4::operator<(Contador4 otro) const {
  return _cont < otro._cont;
}

bool Contador4::operator<=(Contador4 otro) const {
  return _cont <= otro._cont;
}

bool Contador4::operator>(Contador4 otro) const {
  return _cont > otro._cont;
}
```

continúa

Ejercicio práctico

```
bool Contador4::operator>=(Contador4 otro) const {
  return _cont >= otro._cont;
}

void Contador4::mostrar() const {
  cout << _cont << endl;
}

int main() {
  Contador4 c1, c2(7), c3;
  (c1++)++;
  c1.mostrar();
  c3 = c1 + c2;
  c3.mostrar();
  if(c3 > c2) cout << "mayor..." << endl;
  c3 = c1 - c2;
  c3.mostrar();

  return 0;
}
```


Los tipos ... como clases

El tipo **Complejo** (*resolución en grupo*)

¿Recordáis el *tipo* **Complejo** del taller de la Unidad 0?
Todavía no habíamos visto las clases y lo implementamos como pudimos. Ahora es el momento de hacerlo mejor.
Modificad la interfaz (**complejo.h**) y la implementación (**complejo.cpp**) para que el tipo esté implementado como clase.

Comparad las interfaces y las formas de uso de los dos casos.

Los tipos ... como clases

```
// Clase Complejo - Archivo de cabecera "complejo.h"

#ifndef complejo_h // Evitar inclusiones múltiples
#define complejo_h

class Complejo {
public:
  Complejo(double = 0, double = 0);
  Complejo(const Complejo&);
  Complejo& operator=(const Complejo&);
  ~Complejo();
  Complejo operator+(Complejo) const;
  Complejo operator-(Complejo) const;
  Complejo operator*(Complejo) const;
  Complejo operator/(Complejo) const;
  void mostrar() const;
private:
  double _real, _imag;
};

#endif
```

continúa

Los tipos ... como clases

```
// Clase Complejo - Implementación "complejo.cpp"

#include "complejo.h"
#include <iostream>
using namespace std;

Complejo::Complejo(double r, double i) : _real(r), _imag(i) { }

Complejo::Complejo(const Complejo& otro) {
  _real = otro._real;
  _imag = otro._imag;
}

Complejo& Complejo::operator=(const Complejo& otro) {
  _real = otro._real;
  _imag = otro._imag;
  return *this;
}

Complejo::~Complejo() { }
```

continúa

Los tipos ... como clases

```
Complejo Complejo::operator+(Complejo otro) const {
  Complejo comp;
  comp._real = _real + otro._real;
  comp._imag = _imag + otro._imag;
  return comp;
}

Complejo Complejo::operator-(Complejo otro) const {
  Complejo comp;
  comp._real = _real - otro._real;
  comp._imag = _imag - otro._imag;
  return comp;
}

Complejo Complejo::operator*(Complejo otro) const {
  Complejo comp;
  comp._real = _real * otro._real - _imag * otro._imag;
  comp._imag = _real * otro._imag + _imag * otro._real;
  return comp;
}
```

continúa

Los tipos ... como clases

```
Complejo Complejo::operator/(Complejo otro) const {
  Complejo comp;
  // Hacemos que comp sea el complementario de otro
  comp._real = otro._real;
  comp._imag = - otro._imag;
  // Multiplicamos el receptor por el complementario de otro
  comp = (*this) * comp;
  // Sólo queda dividir por la suma de los cuadrados
  // de las partes real e imaginaria de otro
  double divisor = otro._real * otro._real +
                   otro._imag * otro._imag;
  comp._real = comp._real / divisor;
  comp._imag = comp._imag / divisor;
  return comp;
}

void Complejo::mostrar() const {
  cout << "(" << _real << " + " << _imag << "i)\n";
}
```


Los tipos ... como clases

Comparación de las dos alternativas

Las interfaces:

Sin clase

```
struct Complejo {
  double real, imag;
};

Complejo Construye(double, double);
Complejo operator+(Complejo, Complejo);
Complejo operator-(Complejo, Complejo);
Complejo operator*(Complejo, Complejo);
Complejo operator/(Complejo, Complejo);
void mostrar(Complejo);
```

Como clase

```
class Complejo {
public:
  Complejo(double = 0, double = 0);
  ...
  Complejo operator+(Complejo) const;
  Complejo operator-(Complejo) const;
  Complejo operator*(Complejo) const;
  Complejo operator/(Complejo) const;
  void mostrar() const;
private:
  double _real, _imag;
};
```

Encapsulación de código y datos
Ocultamiento de los datos
Mayor simplicidad y claridad

Los tipos ... como clases

El uso:

Como clase

```
Complejo comp1(12,3), comp2(3,4), comp3;


comp1.mostrar();
comp2.mostrar();
comp3 = comp1 + comp2;
comp3.mostrar();
comp3 = comp1 - comp2;
comp3.mostrar();
comp3 = comp1 * comp2;
comp3.mostrar();
comp3 = comp1 / comp2;
comp3.mostrar();
```

Sin clase

```
Complejo comp1, comp2, comp3;
comp1 = Construye(12,3);
comp2 = Construye(3,4);
mostrar(comp1);
mostrar(comp2);
comp3 = comp1 + comp2;
mostrar(comp3);
comp3 = comp1 - comp2;
mostrar(comp3);
comp3 = comp1 * comp2;
mostrar(comp3);
comp3 = comp1 / comp2;
mostrar(comp3);
```

Tipos que usan tipos

Complejos de fracciones (*resolución en grupo*)

A partir de la clase **`Complejo`** anterior, cread otra clase, **`ComplejoFrac`**, que permita trabajar con complejos cuyas partes real e imaginaria sean fracciones.
Probad la clase en una función **`main()`**.

Nota: además del constructor predeterminado habrá otro que acepte dos fracciones; y se añadirán accedentes y mutadores.

¿Cuánto trabajo os ha costado adaptar la clase **`Complejo`** original para que use fracciones en lugar de **`double`**s?

¿Ha habido que cambiar algo en la implementación de **`Fraccion`**?

Tipos que usan tipos

Primeros pasos:

- ✓ Quitar todos los archivos del proyecto.
- ✓ Crear un nuevo archivo (tipo H) y guardarlo con el nombre **`complejofrac.h`**.
- ✓ Abrir el archivo **`complejo.h`**, seleccionar todo su contenido (Ctrl+A) y copiarlo (Ctrl+C). Ir al archivo **`complejofrac.h`** y pegar (Ctrl+V). Reemplazar palabras completas **`Complejo`** por **`ComplejoFrac`**.
- ✓ Volver al archivo **`complejo.h`** y cerrarlo (Close page en el menú contextual de la pestaña).
- ✓ Crear un nuevo archivo (tipo C++) y guardarlo con el nombre **`complejofrac.cpp`**.
- ✓ Abrir el archivo **`complejo.cpp`**, seleccionar todo su contenido (Ctrl+A) y copiarlo (Ctrl+C). Ir al archivo **`complejofrac.cpp`** y pegar (Ctrl+V). Reemplazar palabras completas **`Complejo`** por **`ComplejoFrac`**.
- ✓ Volver al archivo **`complejo.cpp`** y cerrarlo (Close page en el menú contextual de la pestaña).

Tipos que usan tipos

```
// Clase ComplejoFrac - Archivo de cabecera "complejofrac.h"
#ifndef complejofrac_h // Evitar inclusiones múltiples
#define complejofrac_h

#include "fraccion.h"

class ComplejoFrac {
public:
  ComplejoFrac();
  ComplejoFrac(Fraccion, Fraccion);
  ComplejoFrac(const ComplejoFrac&);
  ComplejoFrac& operator=(const ComplejoFrac&);
  ~ComplejoFrac();
  Fraccion real() const;
  Fraccion imag() const;
  void real(Fraccion);
  void imag(Fraccion);
  ComplejoFrac operator+(ComplejoFrac) const;
  ComplejoFrac operator-(ComplejoFrac) const;
  ComplejoFrac operator*(ComplejoFrac) const;
  ComplejoFrac operator/(ComplejoFrac) const;
```

Distintos constructores

Añadimos accedentes y mutadores

continúa

Tipos que usan tipos

```
  void mostrar() const;
private:
  Fraccion _real, _imag;
};

#endif
```

La modificación más relevante en la interfaz

Tipos que usan tipos

```
// Clase ComplejoFrac - Implementación "complejofrac.cpp"

#include "complejofrac.h"
#include <iostream>
using namespace std;

ComplejoFrac::ComplejoFrac() { }

ComplejoFrac::ComplejoFrac(Fraccion r, Fraccion i)
 : _real(r), _imag(i) { }

ComplejoFrac::ComplejoFrac(const ComplejoFrac& otro) {
  _real = otro._real;
  _imag = otro._imag;
}

ComplejoFrac& ComplejoFrac::operator=(const ComplejoFrac& otro) {
  _real = otro._real;
  _imag = otro._imag;
  return *this;
}
```

continúa

Tipos que usan tipos

```
ComplejoFrac::~ComplejoFrac() { }

Fraccion ComplejoFrac::real() const { return _real; }

Fraccion ComplejoFrac::imag() const { return _imag; }

void ComplejoFrac::real(Fraccion f) { _real = f; }

void ComplejoFrac::imag(Fraccion f) { _imag = f; }

ComplejoFrac ComplejoFrac::operator+(ComplejoFrac otro) const {
  ComplejoFrac comp;
  comp._real = _real + otro._real;
  comp._imag = _imag + otro._imag;
  return comp;
}
```

continúa

Tipos que usan tipos

```
ComplejoFrac ComplejoFrac::operator-(ComplejoFrac otro) const {
  ComplejoFrac comp;
  comp._real = _real - otro._real;
  comp._imag = _imag - otro._imag;
  return comp;
}

ComplejoFrac ComplejoFrac::operator*(ComplejoFrac otro) const {
  ComplejoFrac comp;
  comp._real = _real * otro._real - _imag * otro._imag;
  comp._imag = _real * otro._imag + _imag * otro._real;
  return comp;
}

void ComplejoFrac::mostrar() const {
  cout << "(";
  _real.mostrar();
  cout << " + ";
  _imag.mostrar();
  cout << "i)\n";
}
```

Poco trabajo de adaptación

continúa

Tipos que usan tipos

```
ComplejoFrac ComplejoFrac::operator/(ComplejoFrac otro) const {
  ComplejoFrac comp;
  // Hacemos que comp sea el complementario de otro
  comp._real = otro._real;
  comp._imag = - otro._imag;
  // Multiplicamos el receptor por el complementario de otro
  comp = (*this) * comp;
  // Sólo queda dividir por la suma de los cuadrados
  // de las partes real e imaginaria de otro
  Fraccion divisor = otro._real * otro._real +
                     otro._imag * otro._imag;
  comp._real = comp._real / divisor;
  comp._imag = comp._imag / divisor;
  return comp;
}
```

Operación no contemplada en la clase **Fraccion**.
(menos monario o unario; cambio de signo)
Se hace necesario añadirla.

>>> complejofrac.cpp

Tipos que usan tipos

Añadimos la operación de cambio de signo en la clase **Fraccion**.

En la estructura **class** (archivo **fraccion.h**) añadimos el prototipo:

```
Fraccion operator-();
```

En el archivo de implementación (**fraccion.cpp**) añadimos la función:

```
Fraccion Fraccion::operator-()
{
  Fraccion tmp;
  tmp._numerador = - _numerador;
  tmp._denominador = _denominador;
  return tmp;
}
```


Tipos que usan tipos

Un programa principal para probar los **Complejo**s de **Fraccion**es.

```
#include "complejofrac.h"

int main() {
  ComplejoFrac comp1(Fraccion(2,3),Fraccion(2,3)),
               comp2(Fraccion(7,4),Fraccion(5,2));
  comp1.mostrar();
  comp2.mostrar();
  ComplejoFrac comp3 = comp1 + comp2;
  comp3.mostrar();
  comp3 = comp1 - comp2;
  comp3.mostrar();
  comp3 = comp1 * comp2;
  comp3.mostrar();
  comp3 = comp1 / comp2;
  comp3.mostrar();

  return 0;
}
```

>>> complexfrac.cpp