FACULTAD DE INFORMÁTICA

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

RESUMEN

Programación orientada a objetos — Unidad 3

Autor: Luis Hernández Yáñez

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.

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

Método privado que reduce la fracción

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

```
// Clase Fraccion - Archivo de cabecera "fraccion.h"

#ifndef fraccion_h // Evitar inclusiones múltiples
#define fraccion_h

class Fraccion {
public:
  Fraccion(int = 0, int = 1); // Constructor (predeterminado)
  Fraccion(const Fraccion&); // Constructor de copia
  Fraccion& operator=(const Fraccion&); // Copia
  ~Fraccion(); // Destructor
  // Mutadores:
  void numerador(int);
  void denominador(int);
  // Accedentes:
  int numerador() const;
  int denominador() const;
```

F C O

(*continúa*)

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

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

#endif
```

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

```
// Clase Fraccion - Implementación "fraccion.cpp"

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

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

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

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

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

(*continúa*)

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

```
Fraccion::~Fraccion() { }

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

void Fraccion::denominador(int i) {
  _denominador = i; simplifica();
}

int Fraccion::numerador() const { return _numerador; }

int Fraccion::denominador() const { return _denominador; }

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 lo privado de los objetos argumentos y de los objetos locales que sean de esta misma clase

```
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;
}
```

otro, el parámetro por valor, se construye como copia del argumento

(*continúa*)

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

tmp, el objeto local, se crea con el constructor predeterminado

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

Devolución por valor: se devuelve una copia (de **tmp**)

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

Todos los objetos temporales (**otro**, **tmp**) se destruyen al finalizar la ejecución del método

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

```
// Prueba de la clase Fraccion

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

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

(*continúa*)

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

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

  return 0;
}
```

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

Véase el estudio detallado de la ejecución

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

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->**

Métodos operadores

```
// Clase Fraccion - Archivo de cabecera "fraccion.h"
#ifndef fraccion_h // Evitar inclusiones múltiples
#define fraccion_h

class Fraccion {
public:
  Fraccion(int = 0, int = 1); // Constructor (predeterminado)
  Fraccion(const Fraccion&); // Constructor de copia
  Fraccion& operator=(const Fraccion&); // Copia
  ~Fraccion(); // Destructor
  // Mutadores:
  void numerador(int);
  void denominador(int);
  // Accedentes:
  int numerador() const;
  int denominador() const;
  Fraccion operator+(Fraccion) const;
  Fraccion operator-(Fraccion) const;
  Fraccion operator*(Fraccion) const;
  Fraccion operator/(Fraccion) const;
```

(*continúa*)

Métodos operadores

```
  bool operator==(Fraccion) const;
  bool operator!=(Fraccion) const;
  bool operator<(Fraccion) const;
  bool operator<=(Fraccion) const;
  bool operator>(Fraccion) const;
  bool operator>=(Fraccion) const;
  Fraccion& operator++();
  Fraccion& operator--();
  Fraccion& operator+=(Fraccion);
  Fraccion& operator-=(Fraccion);
  void mostrar() const;
private:
  int _numerador, _denominador;
  void simplifica(); // Método privado
};

#endif
```


Métodos operadores

```
// Clase Fraccion - Implementación "fraccion.cpp"
#include <iostream>
using namespace std;
#include "fraccion.h"

...

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;
}

...
```

La implementación (el cuerpo de la función) queda exactamente igual

(*continúa*)

Métodos operadores

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

...

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

...
```

Se usan moldes para forzar operaciones en coma flotante

(*continúa*)

Métodos operadores

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

...
```

Devuelve una referencia a objeto de la clase

Devuelve el propio objeto receptor del mensaje

>>> fraccion.cpp

Devolución de objetos en los métodos

Comparación de las alternativas

`frac2 = frac1++; (frac1++)++;`

```
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

`frac2 = frac1++; (frac1++)++;`

```
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

`frac2 = frac1++; (frac1++)++;`

```
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