# A <Basic> C++ Course

## 5 – Constructors / destructors – operator overloading

*Julien Deantoni*

*adapted from Jean-Paul Rigault courses*

# This Week

- A first class
- Constructor / destructor
- Operator overloading

# Declaration / definition of member-functions

- A member-function declaration is given within the class definition

```
class Rational {
  private:
    int _num;     // numerator
    int _denom;  // denominator
  public:
    int get_num() const;
    int get_denom() const;
    void set_num(const int newNum);
    void set_denom(const int newDenom);
    // ...
};
```

Rational.h

# Declaration / definition of member-functions

- A member-function declaration is given within the class definition

```
class Rational {
  private:
    int _num;     // numerator
    int _denom;   // denominator
  public:
    int get_num() const;
    int get_denom() const;
    void set_num(const int newNum);
    void set_denom(const int newDenom);
    // ...
};
```

A class defines a name scope

Rational.h

# Declaration / definition of member-functions

- A member-function declaration is given within the class definition

```
     class Rational {
       private:
         int _num;      // numerator
         int _denom;   // denominator
       public:
         int get_num() const;
         int get_denom() const;
         void set_num(const int newNum);
         void set_denom(const int newDenom);
         // ...
     };
```

A prototype with a **const** suffix indicates that the member-function does not modify its instance argument

Rational.h

# Declaration / definition of member-functions

- A member-function declaration is given within the class definition

```
class Rational {
  private:
    int _num;     // numerator
    int _denom;  // denominator
  public:
    int get_num() const;
    int get_denom() const;
    void set_num(const int newNum);
    void set_denom(const int newDenom);
    // ...
};
```

A parameter with a **const** prefix indicates that the parameter is in read only inside the associated function

Rational.h

# Declaration / **definition** of member-functions

- A member-function definition is given outside the class definition

```
int Rational::get_num() const{
   return _num;
}
int Rational::get_denom() const{
   return _denom;
}
void Rational::set_num(const int newNum){
   _num = newNum;
   return;
}
void Rational::set_denom(const int newDenom){
   _denom = newDenom;
   return;
}
```

Rational.cpp

# Declaration / **definition** of member-functions

- A member-function definition is given outside the class definition

```
int Rational::get_num() const{
   return _num;
}
int Rational::get_denom() const{
   return _denom;
}
void Rational::set_num(const int newNum){
   _num = newNum;
   return;
}
void Rational::set_denom(const int newDenom){
   _denom = newDenom;
   return;
}
```

These functions are member functions so their qualified name is required

Rational.cpp

# Declaration / **definition** of member-functions

- A member-function definition is given outside the class definition

```
    int Rational::get_num() const{
       return _num;
    }
    int Rational::get_denom() const{
       return _denom;
    }
    void Rational::set_num(const int newNum){
       _num = newNum;
       return;
    }
    void Rational::set_denom(const int newDenom){
       _denom = newDenom;
       return;
    }
```

We should check that newDenom is different from 0

Rational.cpp

# Declaration / definition of member-functions

- A member-function can only be called through a class instance (selection operator `.`)

```
int main()
{
  Rational aRationalObject = 2;
  aRationalObject.set_num(6); //a call to a member function
 return 0;
}
```

main.cpp

# Declaration / definition of member-functions

- A member-function can only be called through a class instance (selection operator **.** Or **->** if pointer)

```
int main()
{
  Rational aRationalObject = 2;
  Rational * aRationalObjectPointer = &aRationalObject;
  aRationalObject.set_num(6); //a call to a member function
  aRationalObjectPointer->set_num(6); // same than the previous line
 return 0;
}
```

main.cpp

# Inline definition of member-functions

- A member-function body may be given within the class definition

```
class Rational {
   private:
     int _num;               // numerator
     int _denom;             // denominator
   public:
     int get_num() const
     {
         return _num;
     }
     int get_denom() const
     {
         return _denom;
     }
     // ...
};
```

Rational.h

➔ Then the member-function is implicitly **inline**

# Inline definition of member-functions

- A member-function body may be given within the class definition

```
class Rational {
   private:
     int _num;             // numerator
     int _denom;           // denominator
   public:
     int get_num() const
     {
         return _num;
     }
     int get_denom() const
     {
         return _denom;
     }
     // ...
};
```

**Inline** means that function calls may be replaced by the textual expansion of its body instead of generating a function call sequence

Rational.h

→ Then the member-function is implicitly **inline**

# Inline definition of member-functions

**Inline** means that function calls may be replaced by the textual expansion of its body instead of generating a function call sequence

```
inline int Rational::get_num() const{
   return _num;
}
inline int Rational::get_denom() const{
   return _denom;
}
```

Rational.cpp

```
void Rational::set_num(const int newNum){
   _num = newNum;
   return;
}
void Rational::set_denom(const int newDenom){
   _denom = newDenom;
   return;
}
```

Rational.cpp

# This Week

- A little reminder
- **Constructor / destructor**
- Operator overloading

# Constructors

- Initialization constructor
  - Initialize the value with the given parameters (or the default parameters)
  - If necessary, allocate the required memory

```
MyClass(parameterType aParam = defaultValue);
```

- Copy constructor
  - Initialize the value with the one of the object given
  - If necessary, allocate the required memory

```
MyClass(const MyClass &);
```

# Constructors

- Initialization constructor
  - Initialize the value with the given parameters (or the default parameters)
  - If necessary, allocate the required memory

```
MyClass(parameterType aParam = defaultValue);
```

➔ Almost every time called automatically by the compiler

- Copy constructor
  - Initialize the value with the one of the object given
  - If necessary, allocate the required memory

```
MyClass(const MyClass &);
```

➔ Called when an object is created and initialized with another object of the same type (*e.g.*, at the beginning of a function call with a copy paradigm)

# Constructors

- Initialization constructor
  - Initialize the value with the given parameters (or the default parameters)
  - If necessary, allocate the required memory

```
MyClass(parameterType aParam = defaultValue);
```

- Copy constructor
  - Initialize the value with the one of the object given
  - If necessary, allocate the required memory

```
MyClass(const MyClass &);
```

**If the copy constructor is private, you forbid using copy during any function call:**

```
 void f (MyClass c);       //KO
 void f (MyClass& c);      //OK
```

# Destructor

- **Destructor**
  - **Release the memory that has been previously allocated**

```
~MyClass();
```

➔ **Always called automatically by the compiler**

- **An object is destroyed at the end of the block in which it was created unless the memory allocation has been explicit (i.e. except a call to new)**

# `Constructor` `&` Conversions of objects

- A one argument constructor defines an *implicit conversion* from the argument type to the class type

```
Rational(int d);
```

  - The following are all equivalent

```
Rational r = 3;
Rational r = (Rational)3;
Rational r(3);
Rational r {3};      //C++11
```

➔ In all cases there is one constructor call, **`Rational(int)`**

# Constructor & Conversions of objects

- A one argument constructor defines an *implicit conversion* from the argument type to the class type

```
Rational(int d);
```

  - The following are all equivalent

```
Rational r = 3;
Rational r = (Rational)3;
Rational r(3);
Rational r {3};      //C++11
void foo(Rational r);  //defined somewhere
foo(3);   //OK, foo(Rational(3)) → a temporary Rational object is created
```

➔ In all cases there is one constructor call, **`Rational(int)`**

# Constructor & Conversions of objects

- A one argument constructor defines an implicit conversion from the argument type to the class type expected if the explicit keyword is used.

```
explicit Rational(int d);
```

  - The following are all equivalent

```
Rational r = 3;
Rational r = (Rational)3;
Rational r(3);
Rational r {3};      //C++11
void foo(Rational r);  //defined somewhere
foo(3);                //KO, no implicit conversion is done
foo((Rational) 3);     //OK even if old style cast
foo(static_cast<Rational>(3)); //OK
```

→ In most cases there is one constructor call, **`Rational(int)`**

# Constructor & Conversions of objects

- Implicit conversions in the other direction can also be defined

```
Class Rational( public: operator double();};
```

```
Rational::operator double() const {
  return (double)_num/ (double)_denom;
}
```

```
// ...
double x = r;
// ...
x = 3.0 + r;
x = 3.0 + static_cast<double>(r);
x = 3.0 + double(r);
```

➔ In all cases, a call to `Rational::operator double()` is made

# Constructor & Conversions of objects

- Implicit conversions in the other direction can also be defined

```
Class Rational( public: operator double();};
```

```
Rational::operator double() const {
  return (double)_num/ (double)_denom;
}
```

```
// ...
double x = r;
// ...
x = 3.0 + r;
x = 3.0 + static_cast<double>(r);
x = 3.0 + double(r);
```

This is an operator overload !

➔ In all cases, a call to `Rational::Operator double()` is made

# Constructor & Conversions of objects

- Explicit conversions in the other direction can also be defined

```
//C++11
Class Rational( public: explicit operator double();};
```

```
Rational::operator double() const {
  return (double)_num/ (double)_denom;
}
```

```
// ...
double x = r; //KO
// ...
x = 3.0 + r; //KO
x = 3.0 + static_cast<double>(r); //OK
x = 3.0 + double(r); //OK
```

➔ Accept only explicit conversion !

# Constructor & Conversions of objects

- Implicit conversions in the other direction can also be defined

```
Rational::operator MaClass() const {
  return MaClass(_num*_denom);
}
```

```
// ...
Rational r ;
// ...
MaClass mc = static_cast<MaClass>(r) ;
```

➔ a call to `Rational::operator MaClass()` is made if no conversion constructor exists

# copy of objects (remember)

- Two cases where an object is “*copied*”:

1. Initializing a `Rational` from an other `Rational`

```
Rational r = {3, 2};      // (3/2)
Rational r1 = {r};
Rational r2(r);
f(r);   //sometimes !
```

2. Assigning a `Rational` to an other `Rational`

```
Rational r(3, 2), r1(3, 4);
r1 = r;
```

# copy of objects (remember)

- Two cases where an object is copied:

1. Initializing a `Rational` from an other `Rational`

```
Rational r = {3, 2};      // (3/2)
Rational r1 {r};
Rational r2(r);
f(r);   //sometimes !
```

2. Assigning a `Rational` to an other `Rational`

```
Rational r(3, 2), r1(3, 4);
r1 = r;
```

➔ In both cases, default is *memberwise* (here bitwise) *copy of underlying C structures*

# Class `Rational` Member-function call

- Two cases where an object is copied:

1. Initializing a `Rational` from an other `Rational`

```
Rational r = {3, 2};      // (3/2)
Rational r1 {r};
Rational r2(r);
f(r);   //sometimes !
```

2. Assigning a `Rational` to an other `Rational`

```
Rational r(3, 2), r1(3, 4);
r1 = r;
```

- This is an assignment (and *not a construction)*

➔ *Depends on the assignment operator implementation...*

# copy of objects (remember)

- Two cases where an object is copied:

1. Initializing a `Rational` from an other `Rational`

```
Rational r = {3, 2};      // (3/2)
Rational r1 {r};
Rational r2(r);
f(r);   //sometimes !
```

Copy Constructor

2. Assigning a `Rational` to an other `Rational`

```
Rational r(3, 2), r1(3, 4);
r1 = r;
```

Assignment operator

➔ In both cases, default is *memberwise* (here bitwise) *copy of underlying C structures*

# copy of objects (remember)

- Two cases where an object is copied:

1. Initializing a `Rational` from an other `Rational`

```
Rational r = {3, 2};      // (3/2)
Rational r1 {r};
Rational r2(r);
f(r);   //sometimes !
```

Copy Constructor

2. Assigning a `Rational` to an other `Rational`

```
Rational r(3, 2), r1(3, 4);
r1 = r;
```

Assignment operator

➔ In both cases, default is *memberwise* (here bitwise) *copy of underlying C structures*

A default operator is generated for the assignment operator but it is not true for all of them…

# Operator overloading

- Operator overloading is a way to realize classical arithmetic operation in a more readable and natural way:

```
Rational r1(3, 2), r2;

r1.add(r2)              |           r1 + r2
r1.multiply(r2)         |           r1 * r2
```

- An operator overload can be of two kinds:
  1. As a member function
     - Identical to other member functions but with imposed name and number of parameters
  2. As a friend function
     - A friend function is a classical (non member or member of another class) function
     - A friend function has privilege (access to the private attributes of a Class with which it is friend)

# Operator overloading
## Definition as a member function

- The assignment operator:

```
Rational& Rational::operator=(const Rational& r){
  _num = r.num;
  _denom = r.denom
  return *this;
}
```

- Usage

```
Rational r {3, 2};
Rational r1 {4, 5};

r = r1;
r.operator=(r1) //same than the previous line
```

# Operator overloading
## Definition as a member function

- The minus unary operator:

```
Rational Rational::operator-() const {
   return Rational(-_num, _denom);
}
```

- Usage

```
Rational r(3, 2);
Rational r1 = -r;
Rational r1bis = r.operator-() //same than the previous line

r = -r1;
r = r1.operator-() //same than the previous line
```

# Operator overloading
## Definition as a member function

- The multiply binary operator:

```
Rational Rational::operator*(Rational r) const {
   return Rational(_num*r._num, _denom*r._denom);
}
```

- Usage

```
Rational r{3, 2}, r1{4, 3};
Rational r2 = r * r1;
Rational r2bis = r.operator*(r1) //same than the previous line

r2 = r * r1;
r2bis = r.operator*(r1) //same than the previous line
```

# Operator overloading
## Definition as a member function

- The multiply binary operator:

```
Rational Rational::operator*(Rational r) const {
   return Rational(_num*r._num, _denom*r._denom);
}
```

Note that the access control is on a *per class basis and not on a per instance basis*

- Usage

```
Rational r{3, 2}, r1{4, 3};
Rational r2 = r * r1;
Rational r2bis = r.operator*(r1) //same than the previous line

r2 = r * r1;
r2bis = r.operator*(r1) //same than the previous line
```

# Operator overloading
## Definition as a member function

- The multiply binary operator:

```
Rational Rational::operator*(Rational r) const {
   return Rational(_num*r._num, _denom*r._denom);
}
```

- Usage

```
Rational r{3, 2}, r1{4, 3};
Rational r2 = r * r1;
Rational r2bis = r.operator*(r1) //same than the previous line

r2 = r * 3;
r2bis = r.operator*(Rational(3)) //same than the previous line

r2 = 3 * r1;                  //??
r2bis = 3.operator*(r1);      //??
```

# Operator overloading
## Definition as a member function

- The multiply binary operator:

```
Rational Rational::operator*(Rational r) const {
   return Rational(_num*r._num, _denom*r._denom);
}
```

- Usage

```
r2 = 3 * r1;                  //??
r2bis = 3.operator*(r1);   //??
```

Problems:

1. The primitive types are not classes (no selection operator)
2. The `int` *class designer* can not anticipated the creation of new classes
3. **No implicit conversion on the hidden argument of a member function**

# Operator overloading

## Definition as a **friend** function

- The multiply binary operator:

```
Rational friend operator*(Rational r1, Rational r2) const {
   return Rational(r1._num * r2._num, _r1.denom * r2._denom);
}
```

- Usage

```
Rational r(3, 2), r1(4, 3);
Rational r2 = r * r1;
Rational r2bis = operator*(r, r1) //same than the previous line

r2 = r * 3;
r2bis = operator*(r, Rational(3)) //same than the previous line

r2 = 3 * r1;
r2bis = operator*(Rational(3), r1); //same than the previous line
```

# Operator overloading
## Definition as a friend function

- The multiply binary operator:

```
Rational friend operator*(Rational r1, Rational r2) const {
   return Rational(r1._num * r2._num, _r1.denom * r2._denom);
}
```

- Usage

Using friend functions restore the symmetry

(no more hidden parameters)

```
Rational r(3, 2), r1(4, 3);
Rational r2 = r * r1;
Rational r2bis = operator*(r, r1) //same than the previous line

r2 = r * 3;
r2bis = operator*(r, Rational(3)) //same than the previous line

r2 = 3 * r1;
r2bis = operator*(Rational(3), r1); //same than the previous line
```

# Operator overloading
## friend or member ?

- For some of them, there is no choice, they must be members:
  - =
  - [ ]
  - ( )
  - `->`

→ They always represents an asymmetric operation

- For the others, one may choose according to:
  - Stylistic consideration
    - `num(r)` vs `r.num( )` ?
  - Symmetry considerations
    - Taking opportunity of implicit conversions ?

# Printing an object

- Using a member-function (or a friend)

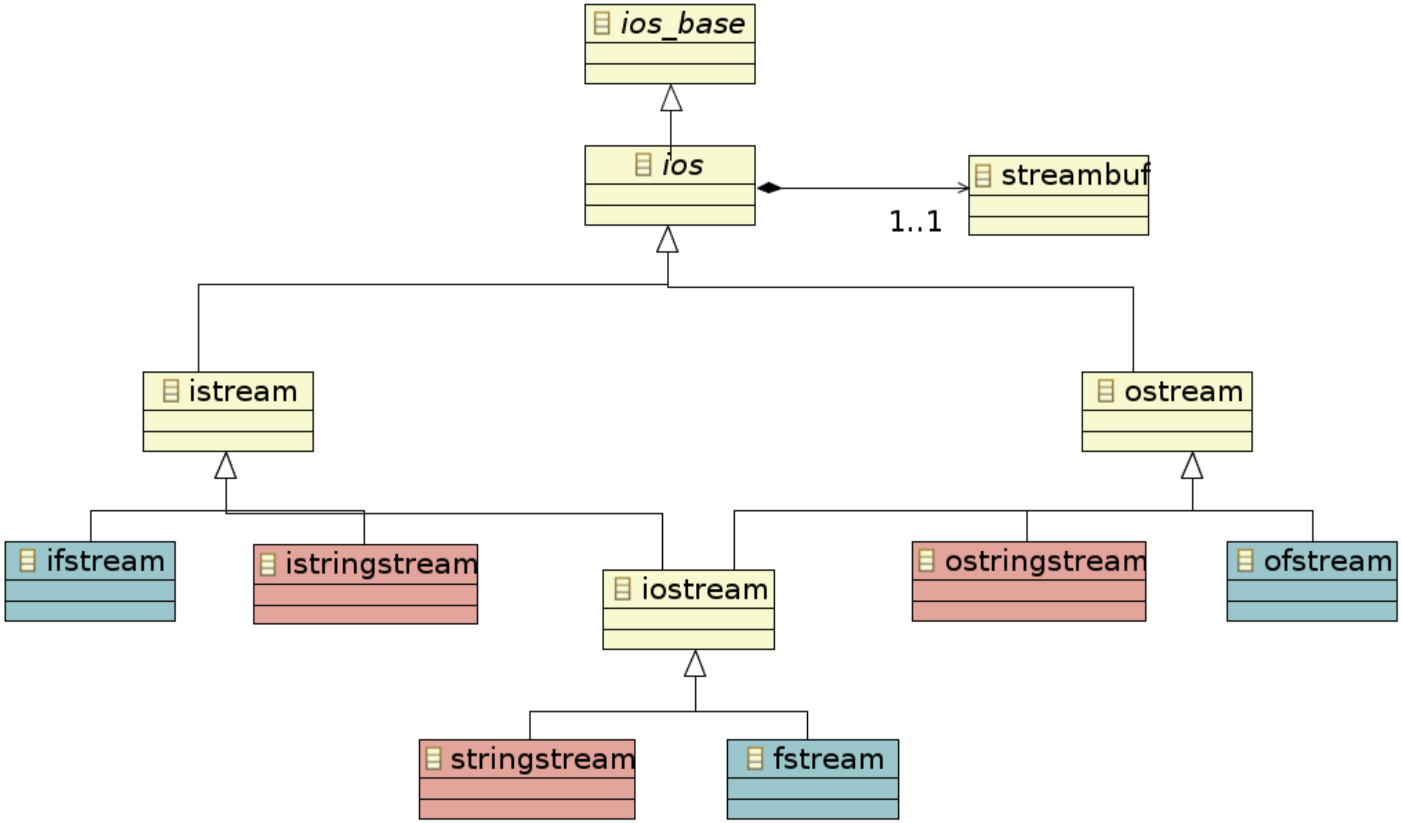
```
#include <iostream>
using namespace std;

void Rational::print() {
  cout << _num << “ / ” << _denom);
}
```

```
Rational r(3, 2);
r.print();                    // stdout --> 3/2
```

☹

# Printing an object

- Using “IO streams”



# Printing an object

- Using output streams

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

class Rational {
 //...

 friend ostream& operator<<(ostream&, Rational);

 // ...
};
```

Object you want to print

# Printing an object

- Using output streams

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

class Rational {
 //...

 friend ostream& operator<<(ostream&, Rational &);

 // ...
};
```

**Reference** on the object you want to print

(to avoid copy of possibly large object)

# Printing an object

- Using output streams

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

class Rational {
 //...

 friend ostream& operator<<(ostream&, const Rational &);

 // ...
};
```

**Constant Reference** on the object you want to print

(because a print is not intended to modify the object)

# Printing an object

- Using output streams

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

class Rational {
 //...

 friend ostream& operator<<(ostream&, const Rational &);

 // ...
};
```

Reference on an output flow

(often the same object modified in the definition of the function)

**Constant Reference** on the object you want to print

(because a print is not intended to modify the object)

# Printing an object

- Using output streams

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

class Rational {
 //...

 friend ostream& operator<<(ostream&, const Rational &);

 // ...
};
```

```
std::ostream& operator<<(std::ostream&  os, const Rational& r){
    os << r.num << '/' << r.denom;
    return os;
}
```

Call to operator<< of int

`operator<<(os, r._num)`

Call to operator<< of char

`operator<<(os, '/')`

Call to operator<< of int

`operator<<(os, r._denom)`

# Printing an object

- Using output streams

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

class Rational {
 //...

 friend ostream& operator<<(ostream&, const Rational &);

 // ...
};
```

```
std::ostream& operator<<(std::ostream&  os, const Rational& r){
    os << r.num << '/' << r.denom;
    return os;
}
```

This print newline and flush the internal stream buffer out

```
// ...
cout << "value of r = " << r << endl;
```

# Editing an object

- Using input streams

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

class Rational {
 //...
 friend istream& operator>>(istream&, Rational &);
 // ...
};
```

```
std::istream& operator>>(std::istream& is, Rational& r){
    is >> r.num;
    char c;
    is >> c;
    is >> r.denom;

    return is;
}
```

```
// ...
cout << "give the value of r " << endl;
cin >> r;
```

# Editing an object

- Using input streams

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

class Rational {
 //...
 friend istream& operator>>(istream&, Rational &);
 // ...
};
```

Why is there a reference ?

```
std::istream& operator>>(std::istream& is, Rational& r){
    is >> r.num;
    char c;
    is >> c;
    is >> r.denom;

    return is;
}
```

```
// ...
cout << "give the value of r " << endl;
cin >> r;
```

# Editing an object

- Using input streams

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

class Rational {
 //...
 friend istream& operator>>(istream&, Rational &);
 // ...
};
```

```
std::istream& operator>>(std::istream&  is, Rational& r){
    is >> r.num;
    char c;
    is >> c;
    is >> r.denom;

    return is;
}
```

```
#include <fstream>
…
Rational r1 = {3,4};
std::fstream file1;   //create an fstream object
file1.open("./temp.txt", std::fstream::out | std::fstream::app); //open temp.txt
file1 << r1 <<std::endl; //writing r to the file
file1.close();  //closing the file
```

# Editing an object

- Using input streams

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

class Rational {
 //...
 friend istream& operator>>(istream&, Rational &);
 // ...
};
```

```
std::istream& operator>>(std::istream&  is, Rational& r){
    is >> r.num;
    char c;
    is >> c;
    is >> r.denom;

    return is;
}
```

```
Rational r1 = {3,4};
std::fstream file1;
file1.open("./temp.txt", std::fstream::out | std::fstream::app);

file1 << r1 <<std::endl;

Rational r2;
file1.close();

file1.open("./temp.txt", std::fstream::in );
file1 >> r2;
file1.close();
```

# Full specification (1/2)

```
class Rational {
 private:
   int _num;        // numerator
   int _denom;      // denominator
 public:
   // Exception classes
   class Bad_Denom {};
   class Bad_Format {};
   // Construction and conversions
   Rational(const Rational&);
   Rational(int n= 0, int d= 1);
   operator double() const;

   // Access functions
   int get_num() const;
   int get_denom() const;

   // Assignment operator
   Rational& operator=(const Rational&);
   // Arithmetic operators
   Rational operator+() const;    // unary plus
   Rational operator-() const;    // unary minus
```

# Class Rational Full specification (1/2)

```
class Rational {
 private:
   int _num = 0;        // numerator C++11
   int _denom = 1;      // denominator C++11
 public:
   // Exception classes
   class Bad_Denom {};
   class Bad_Format {};
   // Construction and conversions
   Rational(const Rational&);
   Rational(int n= 0, int d= 1);
   operator double() const;

   // Access functions
   int get_num() const;
   int get_denom() const;

   // Assignment operator
   Rational& operator=(const Rational&);
   // Arithmetic operators
   Rational operator+() const;    // unary plus
   Rational operator-() const;    // unary minus
```

# Class Rational Full specification (2/2)

```
    // Arithmetic operators (cont.)
    friend Rational operator+(Rational, Rational);
    friend Rational operator-(Rational, Rational);
    friend Rational operator*(Rational, Rational);
    friend Rational operator/(Rational, Rational);

    // Relational operators
    friend bool operator==(Rational, Rational);
    friend bool operator!=(Rational, Rational);
    friend bool operator<(Rational, Rational);
    friend bool operator<=(Rational, Rational);
    friend bool operator>(Rational, Rational);
    friend bool operator>=(Rational, Rational);

    // IO operators
    friend ostream& operator<<(ostream&, const Rational &);
    friend istream& operator>>(istream&, Rational&);
};
```