Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: Operator Overloading

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

Operator

An operator is a special function which differ in the calling of syntax and/or the argument passing mode from the language's usual functions. May C++ operators when called takes one of the following form

- $\triangle$ a: Unary prefix operator, e.g. +a;
- a $\triangle$: Unary postfix operator, e.g. a++;
- a $\triangle$ b: Binary operator, e.g. a+b;

where $\triangle$ is a symbol defining the operator. There is also a single ternary operator, is the ternary conditional operator: a?b:c.
Besides, we have other important operators: operator **new** and **delete**, and conversion operators.

C++ Operators

Here the (almost) complete list

+	-	*	/	%	^	&	|	~	
!	=	<	>	+=	-=	*=	/=	%=	
^=	&=	|=	<<	>>	>>=	<<=	==	!=	as well as:
<=	>=	&&	||	++	--	,	->*	->	
()	[]								

sizeof(), casting operators, **new** and **delete**, conversion operators.
A nice complete list is found on *wikipedia* (look for C++ operators)!

Overloading

Most C++ operators can be defined by the programmer, who can give them a special semantic when applied to user defined types. This operation is usually called *overloading of operators*.
Note: It is not possible to overload `::` (scope resolution), `.` (member selection) and `.*` (member selection through pointer to function).

Free operators and member operators

Some operators are implemented as “free function”, other as member function of a class. Some can be implemented both ways and it is up to the programmer to choose the one most appropriate for the situation at hand.

For a “free” operator the statement `a △ b` translates into `operator △ (a,b)`, while for an operator member of a class `a △ b` translates into `a.operator △ (b)`.

Operators `=`, `[]` e `->` can be implemented only as non static class member.

General rules

- Implement an operator instead of an ordinary function only when its semantic is clear and intuitive. Avoid to implement counter-intuitive operators: prefer ordinary functions/methods!
- the golden rule is when in doubt, do as the `int`'s do. the semantic of an operator should not be too far away to that of the equivalent operation on simple types.
- Even if it is possible, it is in general not advisable to overload the logical operators “and” (`&&` and “or” `||`).
- C++ gives you enough rope to shoot yourself in the foot. When overloading operators stick to the usual conventions for parameter passing and return type, even if C++ gives you much freedom...

La classe Rational

To describe some operators we have developed in Rational/rational.hpp a class that implements rational numbers. Here we would like to define most arithmetic operators, since their semantic for rationals is quite clear. We give a snippet of the class, the full class may be found in the location indicated above.

The class Rational

```
class Rational
{
public:
  explicit Rational(int n=0, int d=1);
  int numerator()const { return M_n;};
  int denominator()const { return M_d;};
  operator double() const;
  Rational &operator +=(Rational const &);
  ...
  Rational operator-() const;
  ...
  Rational & operator++();
  Rational operator++(int);
  ...
  friend Rational operator+(Rational const &,Rational const &);
  friend Rational operator-(Rational const &,Rational const &);
  ...
  friend std::ostream & operator << (std::ostream &, Rational const &);
private:
  int M_n, M_d;
  void M_normalize(); // Normalize the rational
};
```

Some initial comments

The class stores two private members: two integers that contain the numerator and the denominator.
We have a constructor that takes two int arguments, with default values for both. Had the constructor not been declared explicit, it would have defined an implicit conversion `int` $\rightarrow$ `Rational`.
Indeed any constructor accepting a single argument defines an implicit conversion from the type of the argument, unless it is declared explicit.
The reason why we have set the constructor explicit will be discussed later on.
We have not defined a copy constructor explicitly because the synthetic one is enough.

Copy-assignment operator =

This operator is always implemented as a method of the class, with signature

```
T & operator =(const T&).
```

We return a reference since we want that a=b=c be valid. For our rational class the assignment operator has just to copy the variables holding numerator and denominator, so it could be defined as

```
Rational & operator =(Rational const & rhs){
  this->M_n=rhs.M_n;
  this->M_d=rhs.M_d;
  return *this;
}
```

The `this->` can be omitted. The operator returns the same object so that a=b=c works as expected.

Move-assignment operator =

This operator is always implemented as a method of the class, with signature

```
T & operator =(T&&).
```

It moves the content of the rhs. The parameter is a rvalue reference. Since our Rational do not store large data no need of having move operators. Anyway the language provides a synthetic move-assignment that moves members one by one.

```
Rational foo()// A function returning a rational
 ....
 Rational a;
 a = foo(); // calls move assign. if defined.
}
```

Automatic assignment operator

You may not that the Rational class has no assignment operator defined. This is because the language provides a synthetic assignment operator that does exactly what we want: copies variable members (in the order they are declared in the class). However, there are many situations where we need to provide our implementation of the operator. We will cover this situation in a while. Remember that if you define a copy-assignment you do not have the synthetic move-assignment. If you want it you have to define it yourself (or use the **default** statement).

Operator $+=$

The copy-add operator allows to do $a+=b$, and is normally defined everytime we want the addition operator ($+$). We want to be able to add a rational to a rational so we define

```
Rational &operator +=(Rational const &);
```

defined as

```
Rational &
Rational::operator +=(Rational const & r)
{
  M_n =M_n*r.M_d+ r.M_n*M_d;
  M_d *=r.M_d;  M_normalize();
  return *this;
}
```

Operator +

We want to be able to add two rationals. The addition operator is normally implemented as a free function friend of the class. The reason is that we want it to be symmetric with respect to its argument (if the arguments are of the same type).

```
friend Rational operator+(Rational const &,Rational const &);
```

Its definition is

```
Rational operator +(Rational const &l, Rational const & r)
{ Rational temp(l);// Copy constuction
  temp+=r; //Use +=
  return temp;}
```

Note that it returns a Rational and not a Rational &, and that it is implemented using operator += to avoid duplicating code. This is not compulsory yet it is a common practice.

A note

Of course we might overload other operators +, for instance one that takes an integer argument etc. (rules for function overloading apply also to operators).

operator +() returns a value and not a reference since we want that a=b+c be valid, but a+b=c being meaningless.
Instead, **operator** += () returns a reference to the object. This is to replicate the semantic of the same operator applied to ints, where a=(i+=5)*3; is a valid statement (yet DON'T DO IT unless you want to participate to the obfuscated C++ contest).

Clearly, subtraction-copy and subtraction operators are implemented in a similar fashion.

Unary operators + and -

The operators and - have also a unary form (a=−b or a=+b). They are normally implemented as method of the class.

```
Rational operator−() const;
Rational operator+() const;
```

Note that they are **const** methods since they do not change the state of the object. For a rational, unary operator + is a no-operation which returns the value of the object:

```
Rational Rational::operator+() const{
 return *this;}
```

while unary operator - is rather simply implemented using the constructor:

```
Rational Rational::operator−()const{ return
  Rational(−this−>M_n, this−>M_d); }
```

operator ++

The unary increment and decrement operators `++` and `- -`, have a `prefix` and a `postfix` version:

- `++i`: *update and fetch* adds 1 to i and returns a reference ti `i`
  .
- `i++`: *fetch and update* returns the current value of `i` and then increments it by 1.

We want the same thing for a `Rational`. Moreover `i++++` should be forbidden (it is ambiguous), while we want to allow `++++i` (like for the ints!).
Those operators are usually implemented as methods of the class. (but it not compulsory). The `postfix` version takes a dummy integer argument whose only role is to give it a different signature and to allow the compiler to distinguish it from the `prefix` version.

```
Rational & Rational::operator ++(){
  M_n+=M_d;
  return *this;}

Rational  Rational::operator ++(int i){
  Rational temp(*this);
  this->operator++();
  return temp;}
```

Note how the postfix operator is implemented in terms of the prefix operator and that the former returns a reference while the latter returns a value.

streaming operator <<

It would be nice to be capable of writing

```
  Rational a;
..
  std::cout <<a;
```

and have a nice output on the screen. This is possible by defining an output streaming operator (>>) for our `Rational` class. Streaming operators are peculiar in the fact that they can be implemented only as free function and they take as first argument a reference to an `output stream`, and they return a reference to the output stream. The general structure is

```
std::ostream & operator << (std::ostream &, T const &)
```

The ostream returned is just a reference to the one passed as first parameter. This allows to write cout<<a<<b<<endl;.

Output stream for `Rational`

I would like to print a rational number in the form $i + n/m$ where i is its integer part. Moreover I would like that only n/m be printed if the rational is smaller than one.

```
std::ostream & operator << (std::ostream & str,
Rational const & r)
{
 if (r.M_d==1) str<< r.M_n;
  else
   if (int d=r.M_n / r.M_d)
   str<<d<< std::showpos<<r.M_n%r.M_d <<
    std::noshowpos<<'/' << r.M_d;
   else
     str<<r.M_n << '/' <<r.M_d;
 return str;}
```

Some explanations

- the `std::ostream` (output stream) passed by reference as first argument is the returned, again by reference.
- we have used the iostream manipulator `std::showpos` to have the plus sign (+) written in front of a positive number: `cout<< 5;` prints 5, `cout<<showpos<< 5;` prints +5. Manipulators are defined in the standard header file `ios`.
- recall that an POD expression with value different from zero is convertible to the `true` boolean value. This explains statement **if**(**int** d=r.M_n / r.M_d)

The rest is just a simple exercise on the use of the modulo operator.

The input stream operator

Of course it is also possible to define an input stream operator to read data from the terminal or from a file. The general declaration is

```
istream & operator >>(istream &, T &)
```

Normally coding an input stream is more complicated because one has to parse the input to recognize the tokens that compose the value to be read into the variable. And one has to handle also all possible ways to give an acceptable input and treat the possible errors.
In our case we need to recognize that the strings `1+1/2`, `1/2` and `-1` represent a valid `Rational`. Moreover we may have arbitrary blank characters before and after the + sign!...It can be done: C++ has a very powerful set of methods to operate on strings and C++11 has also added tools to treat regular expressions!. But for simplicity (and lack of time) I have implemented a simple version that accepts only input in the form `a/b`.

Input stream for a Rational

An implementation if found in Rational/rational.cpp

Implicit conversion

You can define an implici conversion t/from your designed type and other types in two ways

- By construction. Any constructor not declared explicit that may take a single argument of type `T` defines an implicit conversion from that type.
- By a constructor operator. A constructor operator is a method without return type and without arguments, whose name is the name of the type to which we want to be able to convert. Beware: even if it has not a return type, has to return the converted object! The type is just the name of the operator.

Conversion by construction

A constructor of the type

```
Rational(int num= 0,int den= 1);
```

defines a conversion `int` → `Rational`. A function

```
Rational simplify(Rational const \&);
```

called as

```
r=simplify(1);
```

would translate automatically into

```
r=simplify(Rational(1));
```

Conversion (cast) operator

A conversion operator, also called cast operator, is a *method* of the form

```
operator Type() const;
```

whose role is to convert an object of the class containing the method For instance

```
Rational::operator double() const {
 return
  static_cast<double>(M_n)/static_cast<double>(M_d);}
```

defines the conversion `Rational` $\rightarrow$ `double`:

```
Rational r(1,10);
double d=r+10.9; //  d=r.double()+10.9
```

Note:You may also do **static_cast<double>**(r), as well as (**double**) r.

Why our Rational class has an explicit constructor?

You may wonder why I have made the constructor explicit, thus blocking the implicit conversion from an integer. The reason is that implicit construction is tricky (and sometimes dangerous). In our case having defined arithmetic operators and the conversion to `doubles`, an implicit conversion from `int` could create ambiguous situations.

Assume we had a non-explicit constructor for `Rational` (as the one shown in a previous slide). The statement a + 1, with a a `Rational` would be ambiguous. Did we mean a+Rational(1) (conversion from integer) or (**double**) a + (**double**) 1 (conversion of both `Rational` and `int` to double)??

The compiler is not psychic and will give an error message, not knowing what to do!!. With the explicit constructor to create a `Rational` from an `int` I need to write a+Rational(1) (I can also use **static_cast**<Rational>.

Array subscript operator []

It is an operator that can be implemented only as method of the class and is normally used to return an element of a container. It is usually implemented in the following way

```
T & operator[](integer i);
T   operator[] const (integer i);
```

where `integer` indicates an integral type (for instance **int**, **unsigned int** or **long int**).
We normally have two versions because the first acts when we modify the state of the object:(e.g. a[3]=5). The second instead can operate on constant object (since it is a constant method).

operator ()

The function call operator is the most flexible of operators, indeed it may take an arbitrary number of arguments (even no arguments). Can be implemented only as a method of a class. For instance, I may decide that v(i) should return the i-th element of a Vtr, with bound check. Here a possible definition:

```
double& operator()(int i){
 if(i<0 || i>length) error("Out of bounds");
 return ets[i];}
```

where error exits the program:

```
void error(char* m){
  std::cerr<<m<<std::endl;
  exit(1);}
```

This operator is o fundamental importance in conjunction with function objects.

Consideration about efficiency

There is no easy way of avoiding that arithmetic operator produce temporaries. This can be a drawback for objects of big size (a matrix for instance). We try to explain the situation. Let assume that we have defined the addition and assignment operators a class of vectors Vcr and we carry out

```
v= u + w;
```

We have **operator**+(u,w) returning a temporary object of type Vcr which is then assigned as argument to the copy-assignment:
v.**operator**=(**operator**+(u,w))
If the vector is big (let's say 100 Mbytes) the creation (and later destruction) of the temporary uses up time and memory. The more trivial (but less elegant end more rigid) way

```
for (i=0;i<v.size();++i)v[i]=u[i]+w[i];
```

is much less demanding (the temporary is here of the dimension of 1 double!!)

Consideration about efficiency

There are several possibilities to remedy this situation.

- Using specialized methods and not operators. This however can be cumbersome if you have to account for many different cases.
- Using *expression templates* (the `eigen` library for instance use them a lot). This is a rather complex technique that however may bring great advantages in efficiency. We will dedicate a lecture.
- Using special constructors. This technique somehow mimics expression template but it is more limited. We will give an example when introducing expression templates.
- The new move semantic introduced with the C++11 standard may ease the situation in some cases. We will talk about it later on.