Advanced Methods for Scientific Computing (AMSC)
Lecture title: Operators

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2024/2025

Operator

An operator is a special function that differs in the calling syntax from the language's usual functions. Most C++ operators when called take one of the following form

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

where the symbol $\triangle$ indicates an operator. There is also a single ternary operator: the ternary conditional operator: a?b:c.

Besides, we have other important operators: **sizeof**, **operator new**, and casting operators.

C++ Operators

Here the (almost) complete list

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

as well as: **sizeof**(), casting operators, **operator new** and **decltype**.

A complete list may be found, for instance, on wikipedia.

Precedence and associativity rules

Operators in C++ (like in mathematics) have certain rules for precedence and associativity. For instance, as expected, in the statement

```
a = b + c*d;
```

multiplication has precedence over addition and assignment, and addition has precedence over assignement so `c*d` is computed and the result added to `b`, and finally assigned to `a`.

You can alter the precedence with the parenthesis (as usual):

```
a = (b + c)*d;
```

Now the addition is done first.

Another example

Dereference operator has precedence over multiplication

```
std::unique_ptr<double> m_ptr(new double);
....
double c = *m_ptr * 5.0;
```

As you expect, the code first gets the value obtained by dereferencing m_ptr and multiplies it by 5.0 (it would be very awkward if it were the contrary).

Normally the precedence rules are what you expect. But if in doubt, check it out. A full list of operators with precedence and associativity may be found on cppreference.com (and also on Wikipedia and many other places).

Associativity

Associativity refer to the order operations are executed when you have a succession of operators of the same type, for instance

```
x = a + b + c + d;
```

Addition (like all arithmetic operators) has associativity from left to right. So the code first computes `a+b`, then adds the result to `c` and, finally, to `d`. Only then it assigns the result to `x`, since the assignment operator has a lower priority than additions (it is indeed an operator with very low priority, only the comma operator has a lower priority).

We will see in a while that you can overload operators to give them a different semantic, but you cannot change their precedence and associativity.

An important thing to remember

Within subexpressions at the same level of precedence, the order of evaluation is undefined:

```
x = a*b + c*d;
```

Does this code execute a*b first, or c*d? I don't know, and you don't either! It is undefined, the compiler chooses the one it thinks is best. This applies also to function arguments (they are at the same level of precedence)

```
fun(f(x),g(y));
```

Is f(x) computed first? Not necessarily, we do not know.
You may think that it is irrelevant, and indeed you're right in most cases, but look at this code snippet (from a student's code):

```
i=1;
x=  fun(i,++i);
```

Am I calling f(1,2) or f(2,2). Who knows? Nobody knows!

Overloading

A large part of C++ operators can be (re)defined by the programmer, who can give them a special semantic when applied to user defined types. This operation is called *overloading of operators*. A general overview can be found in cppreference.com.

Note: It is not possible to overload `::` (scope resolution), `.` (member selection) and `.*` (member selection through pointer to function). Nor **sizeof** or **decltype**.

Free operators and member operators

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

For a "free" operator the statement `a △ b` translates into `operator △ (a,b)`, while, for an operator member of class MyClass and being `a` and object of that class, `a △ b` translates into `a.operator △ (b)`.

You can always call an operator with its full name (normally you don't do it, but it may be necessary in some cases). This is a strange, but correct, way of multiplying two numbers

```
x = operator *(a,b); // equivalent to x =a*b;
```

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

An example: the Rational class

To describe some operators we have in Rational/rational.hpp a class that implements rational numbers. I'd 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 repository with the Examples.

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 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` → `Rational`. We recall that any constructor accepting a single argument defines an implicit conversion from the parameter type, unless it is declared explicit.
- The reason why we have set this constructor explicit will be discussed later on.
- We have not defined copy/move constructors explicitly because the synthetic ones are ok. You may use the `default` keyword to make it clearer.
- However, for competeness, in the following I describe a possible copy-assigment operator for our Rationals

Operator +=

The copy-add operator allows us to do a+=b, and it is normally defined every time we want the addition operator (+). In our Rational class it is declared as member with signature

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

and 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;
}
```

The copy-add (as well as subtract-add, multiply-add, divide-add) operators are normally implemented as method of the class (but they can be implemented also as free functions)

Addition Operator

The addition operator may be implemented either as method or as free function.

As a member, its general signature is

```
ReturnType operator+ (ArgumentType const &);
```

and it allows to do `r + x` where `r` is the calling obiect (a `Rational` in our case) and `x` the type we wish to be able to add. In our case `ReturnType` and `ArgumentType` would be both `Rational` since we would like to sum two `Rational`s, returning a `Rational`.

But in other situations they may differ, for instance when we want to add a scalar to a vector, and get a vector.
Note that we have to return a value, not a reference, to replicate the usual semantic of $+$ in $a + b + c$.

Addition operation as free function

When the two arguments are of the same type, it is usual to implement the addition operator as a free function, often friend of the class. In general we have

```
RetType operator+ (ArgType const & x, Argtype const & y);
```

The reason is that we want it to be symmetric with respect to the arguments, accounting also for implicit conversions.

This mimics what happens with basic types: if `i` is an **int** and `x` is a **double** both `i+x` and `x+i` convert `i` to **double** implicitly and return the same double.

That's the choice we made for our `Rational` class. Moreover, if the class has the `+=` operator, the addition operator is normally implemented exploiting it.

The addition operation for two Rational

In our case, we want to be able to add two Rational objects and have a Rational. We define a friend function, so we can access data members directly:

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

The definition of the operator is

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

Note that it is implemented using operator += to avoid duplicating code. This is a common practice.

Some final notes

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+d` be valid, but `a+b=c` is meaningless.

Instead, **operator** += () returns a reference to the object. This is to replicate the semantic of the same operator applied to `int`s, where `(i+=5)=3;` is a valid statement (and `i` will be equal to 3 afterwards) (DON'T DO THAT, unless you want to participate to the obfuscated C contest).

Clearly, −=, −,∗=,∗,...etc. are implemented in a similar way.

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 here 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 to `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 is not compulsory).
The `postfix` version takes a dummy integer argument whose only role is to give it a different signature and allow the compiler to distinguish it from the `prefix` version.

operator ++ for Rational

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

Rational  Rational::operator ++(int i){
  Rational temp(*this);
  this->operator++();// or (*this)++
  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.

Comparison operator

We want to be able to compare Rational. When dealing with comparison operators one must ensure that they are consistent. For Rational it is not a big issue since the comparisons reduce to integer comparisons. But the general rule is to define $<$ first, and then

- $a == b \leftrightarrow \neg(a < b \lor b < a)$ (equivalence)
- $a \leq b \leftrightarrow (a < b) \lor (b == a)$ (less-equal)
- $a > b \leftrightarrow \neg(a \leq b)$ (greater)
- $a \geq b \leftrightarrow (a > b) \lor (b == a)$ (greater-equal)

Comparison operator for Rational

Operator $<$ is declared friend and:

```
inline bool operator<(Rational const& l, Rational const & r){
return (l.M_n*r.M_d) < (r.M_n*l.M_d);
}
inline bool operator==(Rational const& l, Rational const & r){
return !(l<r) && ! (r<l);
}
inline bool operator<=(Rational const& l, Rational const & r){
return (l<r) || (l==r);
}
// and so on...
```

Defined inline for efficiency.

A brief look forward

C++20 has introduced the three-way comparison operator, also called the spaceship operator, indicated by <=> that allows to define all comparison operators in a single piece of code, and ensuring that they are consistent. Not only, they allow to distinguish different categories of comparison operators: strong, weak and partial.

I will not explain here the details of how to define your own spaceship operator, but I will describe the defaulted relational operators introduced by C++20 since they may simplify coding.

Defaulted relational operators

Let's start with the 6 relational operators (<, <=, >, >=,==, = !). Since C++20 you can default them using the keyword **default**. What happens is that the relational operator will be applied to all non-static members of the class. For the comparisons, in lexicographic fashion using their order in the class definition.
In a derived class, the rule is applied recursively on the member of the base class(es), and then on the local members.

```
struct Point{
  double x;
  double y;
  bool operator==(const Point&, const Point &)=default;
};
...
p1 == p2;// true if p1.x==p2.x and p1.y==p2.y
```

Defaulted relational operators, another example

```
struct Point{
  double x;
  double y;
  bool operator==(const Point&, const Point &)=default;
  bool operator<(const Point&, const Point &)=default;
};
...
p1 < p2;//true  true if p1.x<p2.x or (p1.x==p2.x and p1.y<p2.y)
```

Defaulted spaceship

If you default the spaceship operator you get the whole lot. Use **auto** as return type!

```
struct Point{
  double x;
  double y;
  auto operator<=>(const Point&, const Point &)=default;
};
...
p1 < p2;// true  true if p1.x<p2.x or (p1.x==p2.x and p1.y<p2.y)
p1 == p2;// true if p1.x==p2.x and p1.y==p2.y
p1 > p2; //true  true if p1.x>p2.x or (p1.x==p2.x and p1.y>p2.y)
...
```

Clearly, the relational operators are generated only if possible!

The default comparison operators operate lexicographically. If that is not what you want, you need to define your own. For our Rational class the default $<$ is no good. Why?

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 since they can be implemented only as free function (often friend), they normally take as first argument a reference to an `output stream`, and return a reference to the same 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) // integer division!
   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 a POD expression with value different from zero is contextually convertible to the `true` boolean value (while zero is false). 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.

If you are confused by the `if(int d=r.M_n / r.M_d)`, replace it with

```
int d = r.M_n / r.M_d;// integer division
if (d != 0)...
```

The input stream operator

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 &);
```

Coding an input streaming operator is usually complex because one has to parse the input, recognize the tokens that compose the value to be read, handle 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` all represent a valid `Rational`. Moreover, we may allow 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 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

The implementation is found in Rational/rational.cpp

Implicit conversions

You can define an implicit conversion to/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 (non-explicit) conversion (cast) operator. A cast 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, it must has return the object converted to the the type given by the name of the operator.

Conversion by construction

A constructor of the type

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

defines a conversion `int` $\to$ `Rational`. A function

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

called as

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

would translate automatically into

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

Casting (conversion) operator

A casting operator, also called conversion 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` → `double`:

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

Note:You may also do **static_cast**<**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 conversion is tricky (and sometimes dangerous). In our the conversion to `double`, an implicit conversion from `int` could create ambiguous situations.

For instance, the expression **double** x=a + 1, with a a `Rational` would be ambiguous. Do we mean a+Rational(1), i.e. 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, we have to write a+**static_cast**<Rational>(i), or a + Rational{i} (or a+Rational(i), don't you love the redundancies of C++?).

The alternative

Also the casting operator may be specified as **explicit** So an alternative is to have the constructor from **int** not explicit (implicit conversion activated) and the conversion to **double** explicit, by declaring in the class

```
explicit operator  double() const;
```

Now you do not have implicit conversion to **double** (you have to be explicit using **static_cast**<**double**>) but you have implicit conversion from **int**. Or, you may decide to have both conversions **explicit**...

The only thing you cannot have (because it generates ambiguous situations) is to have both implicit conversions activated.

The 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 normally 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 have two versions because the first allows to modify the state of the object:(e.g. a[3]=5). The second, instead, does not modify the state and is the only one that operates on a constant object (since it is a **const** method).

The type of the argument does not need to be an int!, it can be any type you wish! And it can also be given as **const** reference.

Example

```
class Triangle{
public:
Triangle(std::array<Point,3>);
Point & operator[](int i){return M_points[i%3];}
Point operator[](int i)const {return M_points[i%3];}
...
private:
std::array<Point,3> M_points;
};
...
const Triangle a{{p1,p2,p3}};
auto p=a[3]; // calls the const version
a[2]=p;// ERROR a is const. Non-const version not viable.
```

call operator ()

The call operator () is the most flexible of operators, indeed it may take an arbitrary number of arguments (even no arguments). It can be implemented only as a method of a class.

A class that defines a call operator is also called a functor or object function since it behaves "like a function".

The call operator

```
class Foo
{
 double operator(double const & x)
 {return a*x;}
 void setA(double const & x){a=x;}
 private:
 double a=10.0;
};
...
Foo foo;
auto y=foo(3); //y is 30;
foo.setA(-10);
auto z=foo(3);//z is -30
```

Considerations about efficiency

There is no easy way of avoiding producing temporaries with arithmetic operators. This can be a drawback for objects of large size (a matrix for instance). I try to explain the situation. Let's assume we have defined the addition and assignment operators for a class of vectors Vcr and we write

```
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 and less flexible) 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!!)

Considerations 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 this technique). It is a rather complex technique, however it brings great advantages in efficiency, maintaining the nice semantic of operators. I will provide some notes for the interested.