C++: Operator Overloading

Pierre-Alain Fayolle

Table of contents

Introduction

Consider the following examples of code:

- Concatenation of string:

```
string one = "one";
one += "two";
```

  Even if one is a string object, it is possible to apply $+=$ to it

- Addition of complex numbers:

```
complex a(1,1);
complex b(1,2);
complex c = a + b;
```

  Even if a and b are objects of type complex, it is possible to add them with $+$

- These examples are instances of operator overloading: the ability to redefine C++ operators such that they can work with user defined classes

Introduction

- Operator overloading in C++ when used correctly can make program easier to read and more concise
- Essentially operator overloading is used to:
  - Enable user defined classes to behave like primitive types; e.g. define operators like +, - or * on complex number or allow array notation on a vector class
  - Enable code to interact correctly with the standard library; e.g. to customize the behavior of algorithm, overload $\ll$ to make a class compatible with the stream library or redefine $<$ to interface with STL containers

Operators that can be overloaded

- $=$ assignment operator
- $+$, $-$, $*$, $/$, $\%$ arithmetic operators
- $+=$, $-=$, $*=$, $/=$, $\%=$ compound arithmetic operators
- $-$ unary minus
- $\wedge$, &, $|$ bitwise boolean operators (binary)
- $\sim$ bitwise negation (unary)
- &&, $\|$ boolean operators (binary)
- ! boolean negation (unary)

Operators that can be overloaded

- $<$, $<=$, $==$, $>$, $>=$, $!=$ comparison operators
- [] element selection operator
- $++$, $--$ increment and decrement operators
- $*$, $\rightarrow$ dereferencement operators
- $\ll$, $\gg$ stream operators
- () (function) call operator
- new, new[], delete, delete[]

Operators that can not be overloaded

None of the following operators can be overloaded, as it would lead to subtleties:

- :: scope resolution
- . member selection
- ? : ternary conditional
- .* pointer to member selection
- "sizeof" size of object
- "typeid" runtime type information operator

General guidelines for operator overloading

- Principle of least astonishment: use function's names that are consistent with the function's behavior and with the other naming conventions and idioms
- Example:

```
Vec3d v1(1,0,0);
Vec3d v2(0,0,0);
Vec3d v3 = v1 % v2; // add v1 to v2
```

  Operator % was overloaded for Vec3d objects and is defined as the addition for vectors.
- It is better to overload + rather than % to define the addition of two Vec3d objects:

```
Vec3d v3 = v1 + v2;
```

- When overloading operators, make sure to follow existing conventions to avoid confusion

General principles for operator overloading

- An operator "op" is overloaded by defining a function whose name is "operator op". Examples:
  - "operator=" corresponds to the overloaded assignment operator
  - "operator≪" corresponds to the overloaded stream insertion operator
- Operator overloading is syntax sugar: a way of rewriting one operation with a different syntax. Example:

```
string one = "one", two = "two";
string concat = one + two;
```

is equivalent to:

```
string concat.operator= (operator+ (one, two));
```

General principles for operator overloading

- Compare:

  ```
  string concat = one + two;
  ```

  with:

  ```
  string concat.operator= (operator+ (one, two));
  ```

  The former has a more intuitive syntax than the latter.
- Note that operator= is a member function of string, while operator+ is a function. Operators can be overloaded either as methods (member functions) or as functions.
- Only existing operators can be overloaded. It is not possible to create new operators.

General principles for operator overloading

- When overloading an operator, it is not possible to change: its arity (number of parameters), its associativity and its precedence
- Operator overloading lets you define the meaning of operators for user defined type only (class). It is not possible to overload operator for a primitive type (e.g. overload + for ints)
- As a consequence: at least one of the arguments of the overloaded operator should be a user defined type

Element selection operator []

- Consider a user defined class representing a vector of integers:

```
class MyVector {
 // ...
};
```

We want to be able to access the element of a vector the same way we access elements of an array:

```
MyVector vec(10);
vec[0] = 1;
cout << vec[0] << endl;
```

- The code above is equivalent to:

```
vec.operator[] (0) = 1;
cout << vec.operator[] (0) << endl;
```

- To write a custom element selection operator, we write a member function called "operator[]" that accepts as parameter the value going inside the square brackets

Element selection operator []

- operator[] needs to maintain the same functionality than the operator defined on arrays, so the return value should be a reference to some internal data. Example:

```
class MyVector {
 // ...
 public:
   int& operator[] (int index);
};
```

  operator[] takes as argument an int representing the index in the vector and returns a reference to the int in that position in the vector.

- Assuming that the internal representation of MyVector is an array of int, a possible implementation for operator[] can be:

```
int& MyVector::operator[] (int index) {
  return repr[index];
}
```

Element selection operator []

- operator[] returns a reference to an element
- It is common to find also an overloaded operator[] that returns a const reference to an element:

```
class MyVector {
 // ...
  public:
   int& operator[] (int index);
   const int& operator[] (int index) const;
};
```

- The const version is used in the case where the receiver object is immutable. For example:

```
void f(const MyVector&) {
  cout << MyVector[0] << endl;
}
```

Overloading compound assignment operators

- Compound assignment operators are operators of the form op= where op is $+$ or $-$ or ...
- Such operators are usually defined as member functions
- Example: adding a 3×3 matrix to another one

```
class Mat3x3 {
 public:
  Mat3x3& operator+= (const Mat3x3& other);
 private:
  double entries[3][3];
  //...
};
```

Example

- operator$+=$ can be implemented as:

```
Mat3x3& Mat3x3::operator+= (const Mat3x3& other) {
 for (int i=0; i < 3; ++i)
  for (int j=0; j < 3; ++j)
   entries[i][j] += other.entries[i][j];
  return *this;
 }
```

- It has to return a reference to the receiver object in order to behave like the $+=$ operator on primitive type.

Overloading compound assignment operators

- Similarly we can add $-=$, $*=$ and $/=$:

```
class Mat3x3 {
 public:
  Mat3x3& operator+= (const Mat3x3& other);
  Mat3x3& operator-= (const Mat3x3& other);
  Mat3x3& operator*= (double scale);
  Mat3x3& operator/= (double scale);
 private:
  double coord[3];
  //...
};
```

- The last two operators are a bit different as they do not take an argument of type Mat3x3

operator− =

- Note that the operator − = can be defined in terms of + = and the unary minus operator:

```
Mat3x3& Mat3x3::operator-= (const Mat3x3& other) {
  *this += -other;
  return *this;
}
```

- Where the unary minus operator is declared as:

```
class Mat3x3 {
 public:
  //...
  const Mat3x3 operator- ();
  //...
};
```

operator− =

- operator− = can be defined as;

```
const Mat3x3 Mat3x3::operator-() {
 Mat3x3 result;
 for (int i=0; i<3; ++i)
  for (int j=0; j<3; ++j)
   result.entries[i][j] = -entries[i][j];
 return result;
}
```

Binary mathematical operators

- With the defined $*=$ operator, we can write:

  `Mat3x3 m *= 2.0;`

  to scale each component by 2.0
- Now, we want to be able to write:

  `m = m * 2.0;`

  Which means that we need to add the operator $*$
- Usually, we also want to be able to write:

  `m = 2.0 * m;`

Binary mathematical operators

- If we want to overload $*$ to be defined in both cases, it needs to be done as a function and not as a member function
- Recall that:

  `b * c;`

  will be expanded as b.operator*(c) if operator $*$ is defined as a member function. In the case of a primitive type, it does not make sense
- operator $*$ will therefore be defined as a regular function:

```
const Mat3x3 operator* (const Mat3x3& m, double s) {
 Mat3x3 result = *m;
 m *= s;
 return result;
}
```

Binary mathematical operators

- And symmetrically, we define:

```
const Mat3x3 operator* (double s, const Mat3x3& m) {
 return m*s;
}
```

- As a general rule, binary mathematical operators, like $*$ ($+$, $-$, ldots), should be implemented as regular functions (not as member functions)

Overloading the increment (++) and decrement (−−) operators

- Increment and decrement operators exist in two versions: postfix and prefix:

```
int i = 0;
cout << ++i << endl; // 1
cout << i << endl; //1

i = 0;
cout << i++ << endl; // 0
cout << i << endl; // 1
```

- When overloading operators, the compiler differentiates between the prefix and the postfix version by looking at the number of arguments:
  - The prefix version takes no arguments:
    `Type& operator++ ();`
  - The postfix version takes a dummy int as argument:
    `const Type operator++ (int dummy);`

Overloading ++ and −−

- Assuming that we have already implemented $+=$, we can write the prefix operator as:

```
Type& Type::operator++ () {
 *this += 1:
 return *this;
}
```

- The postfix operator then becomes:

```
const Type Type::operator++ (int dummy) {
 Type result = *this;
 *this += 1;
 return result;
}
```

Relational operators

- Unlike the assignment operator ($=$), which is provided by default by the compiler, other relational operators need to be explicitly overloaded
- While binary mathematical (or arithmetical) operators are usually defined as regular functions, relational operators (i.e. $<$, $>$, ...) are usually defined as member functions
- The declaration of a relational (in that case $<$) usually looks like this:

```
class Type {
 public:
  bool operator< (const Type& other) const;
  // ...
};
```

Relational operators

Overloading relational operators (especially $<$) is often needed when using the STL, e.g.:

- when using the sort algorithm from the STL to sort a collection of objects
- when collecting objects in an STL set
- when mapping objects to value with the STL map
- when using the STL priority queue

Friend

- When marking data members as private, only instances of that class can access them
- Sometimes we also want outside classes or functions to access private members of a class. One example is overloaded operators defined as functions outside of a class
- One possible way to solve this issue without creating public accessor methods (that would make the data open to any objects and therefore would break encapsulation) is to use the "friend" keyword
- A class or function tagged as "friend", can read and modify private members of a class

Friend

- When declaring a class or function as friend, the declaration must precede the implementation of the friend class or function. Example:

```
class Type {
 public:
  //...
  friend void friendFunction(Type& arg);
};
```

  The function "friendFunction(arg)" can then be defined
- The concept of "friend" function or class should be used carefully as it can lead to code that defeats the purpose of encapsulation

Overloading stream operators

- The C++ stream library is designed to be flexible and to allow to define our own stream insertion and extraction operators
- Operators $\ll$ and $\gg$ can be defined for user defined classes, such that the following code is well defined:

```
// Assume type of matrix is Mat3x3
// with Mat3x3 discussed earlier
cin >> matrix;
cout << matrix << endl;
```

Overloading $\ll$

- We want to define $\ll$ for a user defined class e.g. Mat3$\times$3, such that the following is well defined:

```
// Assume type of matrix is Mat3x3
cout << matrix << endl;
```

- $\ll$ can not be defined as a member function otherwise the following code would be valid:

```
matrix << cout;
```

- $\ll$ should be defined as a regular function:

```
ostream& operator<< (ostream& os, const Mat3x3& m);
```

Overloading $\ll$

- The output stream should be passed by reference as it will be modified by $\ll$
- m can be marked const, because it will not be modified by $\ll$
- $\ll$ returns a reference to an ostream, because it should be legal to chain $\ll$ operators like in:
  `cout << m1 << " " << m2 << endl;`
- The stream returned can not be const, otherwise we could not use stream manipulators
- operator$\ll$ needs to be defined as a function. It will need to access data members of Mat3$\times$3 that are most likely private. Thus $\ll$ needs to be declared as a friend of the class Mat3$\times$3:

```
class Mat3x3 {
// ...
public:
 friend ostream&
    operator<< (ostream& os, const Mat3x3& m);
};
```

Overloading $\ll$

- Implementing the function operator$\ll$ is not difficult. We need to print meaningful data from the object with some formatting:

```
ostream& operator<< (ostream& os, const Mat3x3& m) {
 for (int i=0; i<3; ++i) {
  for (int j=0; j<3; ++j) {
    os << m.entries[i][j] << " ";
  }
  os << endl;
 }
 return os;
}
```

More operators to overload

Other operators are sometimes overloaded but we will not cover them in this class:

- new, new[], delete, delete[]: redefining them, allow us to write a custom memory allocator
- $*$, $\rightarrow$: by redefining them, we can make object act like pointers (e.g. smart pointers or iterators)

More operators to overload

- Overloading the operator() allows to make objects act like functions
- Such objects are called functors and are used a lot to customize the behavior of the STL algorithms