Computer Programming

Operator Overloading

Hung-Yun Hsieh

November 29, 2022

Operator Overloading

- Using operators with objects
  - Operators provide a *handy* way for data processing
    - Function calls require passing of parameters and they are less intuitive to use
    - Built-in operators work with fundamental data types such as integers and floating numbers
    - The syntax and functionality of these operators are well-defined
  - It is good to use operators on user-defined objects
    - Overloading existing operators for them to work with user-defined types

```
object3 = object1.add( object2 );
vs.
object3 = object1 + object2;
```

List of Operators

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded				
.	.*	:	?:	sizeof

Caveats

Default assignment operation: member-wise assignment

- Operator overloading
  - Overloading cannot change the original precedence, associativity, and arity of an operator
  - Only existing operators can be overloaded
    - Cannot create new operators (such as `**`)
  - Each operator to use must be overloaded explicitly
    - Overloading `+` and `=` does not overload `+=`
  - All operators must be defined before use, except for
    - Assignment operator (`=`)
    - Address operator (`&`)
    - Comma operator (`,`)

  where a default version is provided by the compiler to operate on *use-defined objects*

These three operators "can" also be overloaded if desired

Writing Overloaded Operators

- Similar to function calls
  - At least one of the operands must be a user-defined type
    - One cannot change how operators act on *built-in data types* (i.e., cannot change integer addition)
  - Operator overloading works similar to function calls
    - Operators in C++ are *implemented as functions*
  - Use a non-static member function or global function
    - Member function for the class with operators to be overloaded
    - Global function needs to take one extra argument as the object of the class with operators to be overloaded
  - Function name becomes the keyword `operator` followed by the symbol or name of the operator
    - The function name `operator+` can be used to overload the addition operator (`+`)

Static member functions can access only static data members of the class

① Overloading Unary Operator

Other similar operators:
+ − ~ & *

- Unary operator
  - Can be overloaded as a non-static member function with no arguments *or* as a global function with one argument
  - Argument must be object or reference to object

```
class AClass
      {
      public:
          bool operator!() const;
          …
      } s;
```

Returning a reference of the same class allows operator chaining

☞ `!s` calls the function `s.operator!()`

```
bool operator!(const AClass &);
```

```
bool operator!(AClass)
```

☞ `!s` calls the function `operator!( s )`

② Overloading Binary Operator

Other similar operators:
`* / % = >`
`+= && []`

- Binary operator
  - Non-static member function takes one argument
  - Global function takes two arguments
  - One argument must be class object or reference

```
class AClass
     {
     public:
        AClass operator+(const AClass &) const;
        …
     } y, z;
```

☞ `y + z` calls the function `y.operator+( z )`

```
AClass operator+(const AClass &, const AClass &);
```

☞ `y + z` calls the function `operator+( y, z )`

Member vs. Global Functions

anObject + 2 *vs.* 2 + anObject

- Member or global functions
  - As a member function
    - Must be defined in the class of the *leftmost* object
    - Use `this` to implicitly get the left operand argument
    - Called when left operand of binary operator is of this class
    - Called when single operand of unary operator is of this class
  - As a global function
    - Can be implemented outside the class definition
    - Need parameters for all operands (e.g. left and right operands)
    - Need to be a `friend` to access `private` data of objects

☞ Sometimes *only* member functions can be used

- `()`, `[]`, `->` and any assignment operator (=, +=, …)
- First operand cannot be built-in data types

③ Overloading << and >>

cin is an object of the standard class istream
cout is an object of the standard class ostream

- Stream insertion and extraction operators

```
rational x(2,3);
cout << "The rational in n/d is " << x << endl;
```

- Left operand is not of the user-defined class type (cout)
  - Use a global function for overloading << since it is not possible to modify the built-in ostream class
  - Return ostream & for operator chaining (no const here)
  - Declare the global function as a friend of the user-defined class for access of private members

```
rational x;
cout << "Enter a rational number in the form of n/d: ";
cin >> x;
```

- Use a global function that returns istream & for overloading the >> operator (no const here)

Rational – Take Two (1/4)

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

class rational {
 friend ostream & operator<<(ostream &, const rational &);
 friend istream & operator>>(istream &, rational &);

 public:
    rational(int x=0, int y=1) {n=x; d=y;}
    rational operator+ (const rational&);
    rational operator* (const rational&);
    rational operator/ (const rational&);

 private:
    int n, d;
};

ostream & operator<<(ostream &, const rational &);
istream & operator>>(istream &, rational &);
```

Do error checking (**`if y==0`**)

Operator overloading

Okay to use **`rational`** here

Rational – Take Two (2/4)

```
rational rational::operator+(const rational& y) {
    rational z;
    z.n = n*y.d + d*y.n;
    z.d = d*y.d;
    return z;
}
```

```
a + b
→ a.operator+(b)
```

```
rational rational::operator*(const rational& y) {
    rational z;
    z.n = n*y.n;
    z.d = d*y.d;
    return z;
}
```

It is possible to call a constructor explicitly to create a temporary object

```
rational rational::operator*(const rational&y)
{
  return rational(n*y.n, d*y.d);
}
```

```
rational rational::operator/(const rational& y) {
    rational z;
    z.n = n*y.d;
    z.d = d*y.n;
    return z;
}
```

Rational – Take Two (3/4)

```
ostream &operator<<(ostream &output, const rational &r) {
    output << r.n << "/" << r.d;
    return output;
}

istream &operator>>(istream &input, rational &r) {
    char c;

    do {
      input >> r.n;
      input >> c;
      input >> r.d;
    } while (c!='/' || r.d==0);

    return input;
}
```

Important to make it a `friend` of `rational` to access `r.n` and `r.d`

Try to write a more sophisticated input function for yourself

Operator chaining:

```
cin >> x >> y;
→
operator>>( operator>>(cin,x), y );
```

Rational – Take Two (4/4)

```
int main( )
{
    rational a, b, c, d, e;

    cout << "Please enter a="; cin >> a;
    cout << "Please enter b="; cin >> b;

    c = a + b;
    d = a * b;
    e = a / b;

    cout << a << " + " << b << " = " <<
    cout << a << " * " << b << " = " <<
    cout << a << " / " << b << " = " <<
}
```

```
int main( )
{
    rational a={4, 5}, b={2, 3};
    rational c, d, e;



    c = rplus(a, b);
    d = rmultiply(a, b);
    e = rdivide(a, b);

     cout<<a.n<<"/"<<a.d<<"+"<<
     b.n<<"/"<<b.d<<"="<<c.n<<"
     /"<<c.d<<endl;
     …
}
```

```
Please enter a=2/3
Please enter b=4/5
2/3 + 4/5 = 22/15
…
```

④ Overloading ++ and --

The dummy parameter (int 0) is used just to distinguish prefix and postfix increment

- Increment and decrement operators
  - Prefix

```
AClass & AClass::operator++();
```

☞ ++d1 becomes d1.operator++()

```
AClass &operator++( AClass & );
```

☞ ++d1 becomes operator++(d1)

  - Postfix

```
AClass AClass::operator++( int );
```

This parameter will not be used inside the function

☞ d1++ becomes d1.operator++(0)

```
AClass operator++( AClass &, int );
```

☞ d1++ becomes operator++(d1, 0)

Example on Operator ++ (1/2)

- The `Digit` class holds a single digit from 0 to 9

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

class Digit
{
    int dgt;

public:
    Digit(int nDigit=0) { dgt = nDigit; }

    Digit& operator++();      // prefix
    Digit  operator++(int);   // postfix

    int GetDigit() const { return dgt; }
};
```

Example on Operator ++ (2/2)

```
Digit& Digit::operator++()
{
    if (dgt == 9)  dgt = 0;
    else          ++dgt;
    return *this;
}

Digit Digit::operator++(int)
{
    Digit cResult(dgt);
    ++(*this);
    return cResult;
}

int main()
{
    Digit cDigit(5);
    ++cDigit;             // calls Digit::operator++();
    cDigit++;             // calls Digit::operator++(int);
}
```

Note the difference on y

`y = ++x;`

`y = x++;`

No reference here (need the value before the increment)

A temporary object is returned instead of incremented `*this`

⑤ Overloading =

- Assignment operator
  - Assignment of the same type (copy assignment)

```
rational x(2,3), y;
y = x;
```

```
rational rational::operator=(const rational &);
```

☞ Member-wise assignment is provided by the compiler

  - Assignment of different types (conversion assignment)

```
rational x;
x = 5;
```

```
rational rational::operator=(int);
```

A Note on Copy Assignment

How many such default functions does a class have?

- Copy assignment operator
  - The compiler will automatically generate a default copy assignment operator if not defined by the programmer
    - Member-wise assignment (shallow copy)
  - Copy assignment operator vs. copy constructor
    - Need to clean up the data members of the assignment's target

☞ *Rule of three*

  - If a class defines one of the following, it should probably explicitly define all three:
    - destructor
    - copy constructor
    - copy assignment operator

```
rational x(2, 3), y;
y = x; // copy assignment

vs.

rational x(2, 3);
rational y = x; // copy const.
```

Overloading = with Pointers

- Typical procedures for assignment overloading
  - Delete the associated memory of the LHS (left-hand-side) object if applicable (LHS is to be overwritten by RHS)
  - Perform the desired assignment
    - May need deep copy in case of pointers
  - Check for no self-assignment
    - Make sure an object is not assigned to itself (e.g. `x=x`)
    - ☞ Self assignment fails because the memory associated with the current object of the LHS is de-allocated before the assignment, which would invalidate using it from the RHS
  - Return `*this`
    - Allow cascading of assignment operations (e.g. `x=y=z`)

Example on Assignment (1/2)

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

class Person
{
    static int total;
    int id;
    char *name;

  public:
    Person() : name(NULL) { id=++total; }
    Person(const char *);
    Person& operator=(const Person &p);
    const char *Name() { return name; }
    ~Person() { if (name!=NULL) delete [] name; }
};

int Person::total = 0;
```

```
Person::Person(const char *tag)
{
    id=++total;
    name = new char[strlen(tag)+1];
    strcpy(name, tag);
}
```

Example on Assignment (2/2)

```
Person& Person::operator=(const Person &p)
{
    if (this != &p)
    {
        id = p.id;
        if (name!=NULL) delete [] name;
        name = new char[strlen(p.name)+1];
        strcpy(name, p.name);
    }
    return *this;
}

int main()
{
    Person a("John"), b("Noname"), c;
    c = b = a;

    cout << b.Name();
}
```

`Person::operator=()` can access private members of `Person` since it is its member function

```
Person::Person(const char *tag)
{
    id=++total;
    name = new char[strlen(tag)+1];
    strcpy(name, tag);
}
```

⑥ Overloading ()

- Parenthesis operator
  - Most overloaded operators allow us to vary the type of parameters for the operator
  - The parenthesis operator allows us to vary both the type and **number** of parameters it takes
  - ☞ Operator () must be implemented as a member function
  - Operator () is commonly overloaded with multiple parameters to *index* a multi-dimensional array
    - The subscript operator [] takes only one parameter
  - Operator () can be used to retrieve a subset of a one dimensional array
    - Return a sub-string beginning from the first position to the second position of the input string

Example on Parenthesis (1/2)

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

class Matrix
{
    double adData[4][4];

  public:
    Matrix();
    double& operator()(const int nCol, const int nRow);
    void    operator()();
};

Matrix::Matrix()
{
    for (int nCol=0; nCol<4; nCol++)
    for (int nRow=0; nRow<4; nRow++)
         adData[nRow][nCol] = 0.0;
}
```

Example on Parenthesis (2/2)

```
double& Matrix::operator()(const int nCol, const int nRow)
{
    assert(nCol >= 0 && nCol < 4);
    assert(nRow >= 0 && nRow < 4);
    return adData[nRow][nCol];
}
void Matrix::operator()()
{
    for (int nCol=0; nCol<4; nCol++)
    for (int nRow=0; nRow<4; nRow++)
        adData[nRow][nCol] = 0.0;
}

int main( )
{
    Matrix cMatrix;
    cMatrix(1, 2) = 4.5;
    cMatrix();
    cout << "Value is " << cMatrix(1, 2);
}
```

Be careful that this statement can be confusing

cf. `Matrix();`

Review

- Operator overloading
  - Use a member function or global function for operator overloading
  - Restrictions on using operator overloading
  - Overloading unary & binary operators
  - Overloading << and >>
  - Overloading ++ and --
  - Overloading assignment operator
  - Overloading ()