C++ Program Design
-- Operator overloading

Junjie Cao @ DLUT

Summer 2016

http://jjcao.github.io/cPlusPlus

Operators as functions

- Using function overloading to overload operators is called **operator ov erloading**.
- `double z = 2.0;`
- `double w = 3.0;`
- `cout << w + z << endl;`

- `Mystring string1 = "Hello, ";`
- `Mystring string2 = "World!";`
- `std::cout << string1 + string2 << '\n'; //error`

Resolving overloaded operators

- If *all* of the operands are fundamental data types, the compiler will call a built-in routine if one exists.
- If one does not exist, the compiler will produce a compiler error.

- If *any* of the operands are user data types (e.g. one of your classes, or an enum type), the compiler looks to see whether the type has a matching overloaded operator function that it can call.

- If it can’t find one, it will try to convert one or more of the user-defined type operands into fundamental data types so it can use a matching built-in operator (via an overloaded typecast, which we’ll cover later in this chapter).
- If that fails, then it will produce a compile error.

What are the limitations on operator overloading?

1. almost any existing operator in C++ can be overloaded.
   - The exceptions are: conditional (?:), sizeof, scope (::), member selector (.), and member pointer selector (.*).

2. you can only overload the operators that exist. You can not create new operators or rename existing operators

3. at least one of the operands in an overloaded operator must be a user-defined type.

4. it is not possible to change the number of operands an operator supports.
5. all operators keep their default precedence and associativity and this can not be changed

Overloading the arithmetic operators using friend functions

```
class Cents{
…
// add Cents + Cents using a friend function
friend Cents operator+(const Cents &c1, const Cents &c2);
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2){
// use the Cents constructor and operator+(int, int)
// we can access m_cents directly because this is a friend function
return Cents(c1.m_cents + c2.m_cents);
}
```

Overloading operators for operands of different types

```
class Cents{
…
// add Cents + int using a friend function
friend Cents operator+(const Cents &c1, int value);
// add int + Cents using a friend function
friend Cents operator+(int value, const Cents &c1);
…
};
```

Overloading operators using normal functions

```
// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
// use the Cents constructor and operator+(int, int)
// we can access m_cents directly because this is a friend funct
ion
return Cents(c1.m_cents + c2.m_cents);
}
```

Overloading the I/O operators

```
class Point{
private:    double m_x, m_y, m_z;
public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y
(y), m_z(z) { }

    double getX() { return m_x; }
    double getY() { return m_y; }
    double getZ() { return m_z; }
};
        Point point(5.0, 6.0, 7.0);
        std::cout << "Point(" << point.getX() << ", " <<
            point.getY() << ", " <<
            point.getZ() << ")";
```

```
class Point{
public:
    void print(){
     std::cout << "Point(" << m_x << ", " << m_y << ", " << m_z << ")";
    }
};
```

- std::cout << "My point is: ";
- point.print();
- std::cout << " in Cartesian space.\n";

```
cout << "My point is: " << point << " in Cartesian space.\n";
```

Overloading operator<<

- Consider the expression std::cout << point. If the operator is <<, what a re the operands?
  - The left operand is the std::cout object, and the right operand is your Point class object
- `// std::ostream is the type for object std::cout`
- `friend std::ostream& operator<< (std::ostream &out, const Point &point);`

```
class Point{
    friend std::ostream& operator<< (std::ostream &out, const Po
int &point);
};


std::ostream& operator<< (std::ostream &out, const Point &point)
{
    out << "Point(" << point.m_x << ", " << point.m_y << ", " <<
point.m_z << ")";


    return out;
}
```

The trickiest part here is the return type.

- `friend Cents operator+(const Cents &c1, const Cents &c2);`
- `friend std::ostream& operator<< (std::ostream &out, const Point &point);`

- so we can “chain” output commands together, such as std::cout << point << std::endl;
  - If returning void => void << std::endl;
  - If returning ostream& =>std::cout << std::endl;

Overloading operator>>

```
    friend std::ostream& operator<< (std::ostream &out, const Point &poin
t);
    friend std::istream& operator>> (std::istream &in, Point &point);
};

std::ostream& operator<< (std::ostream &out, const Point &point){
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z
<< ")";
    return out;
}

std::istream& operator>> (std::istream &in, Point &point){
    in >> point.m_x;    in >> point.m_y;    in >> point.m_z;
    return in;
}
```

Overloading operators using member functions

1. The overloaded operator must be added as a **member function of the left operand**.
2. The **left operand** becomes the implicit ***this** object
3. All other operands become function parameters.

```
class Cents{
public:
   Cents operator+(int value); // Overload Cents + int
};

Cents Cents::operator+(int value){
    return Cents(m_cents + value);
}
```

overload an operator as a friend or a member, which should use use?

- **Not everything can be overloaded as a friend function**
  - The assignment (=), subscript ([]), function call (()), and member selection (->) operators must be overloaded as member functions, because the language requires them to be.
- **Not everything can be overloaded as a member function**
  - we are not able to overloaded operator<< as a member function.
  - Because the overloaded operator must be added as a member of the left operand.

1. a unary operator => a member function.
2. a binary operator that modifies its left operand => a member function.
3. a binary operator that does not modify its left operand => a normal function or friend function.

Overloading unary operators +, -, and !

```
class Point{
public:
    // Convert a Point into it's negative equivalent
    Point operator- () const;
    // Return true if the point is set at the origin
    bool operator! () const;
```

```
Point point; // use default constructor to set to (0.0, 0.0, 0.0)

if (!point)
   std::cout << "point is set at the origin.\n";
else
   std::cout << "point is not set at the origin.\n";
```

Omit

- Overloading the comparison operators
- Overloading the increment and decrement operators

Overloading the subscript operator

```
class IntList{private:    int m_list[10];
public:
    void setItem(int index, int value) { m_list[index] = value; }
    int getItem(int index) { return m_list[nIndex]; }
};

int main(){
    IntList list;
    list.setItem(2, 3);
```

```
class IntList{
private:
    int m_list[10];
public:
    int* getList() { return m_list; }
};

IntList list;
list.getList()[2] = 3;
```

Overloading operator[]

```
class IntList{private:    int m_list[10];
public:
    int& operator[] (const int index);
};
int& IntList::operator[] (const int index){
    return m_list[index];
}

IntList list;
list[2] = 3; // set a value
std::cout << list[2]; // get a value
```

Dealing with const objects

```
IntList list;
list[2] = 3; // okay: calls non-const version of operator[]
std::cout << list[2];


const IntList clist;
clist[2] = 3; // compile error: calls const version of operator
[], which returns a const reference.  Cannot assign to this.
std::cout << clist[2];
```

```
class IntList{
private:
int m_list[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // give this class some initial state for this example
public:
    int& operator[] (const int index);
    const int& operator[] (const int index) const;
};
 int& IntList::operator[] (const int index) // for non-const objects: can be used for assignment
    return m_list[index];
}
const int& IntList::operator[] (const int index) const // for const objects: can only be used for access
    return m_list[index];
}
```

Error checking

```
int& IntList::operator[] (const int index)
{
    assert(index >= 0 && index < 10);

    return m_list[index];
}
```

Pointers to objects and overloaded operator[] don't mix

```
IntList *list = new IntList;
list [2] = 3; // error: this will assume we're accessing the 3rd element of an array of IntLists


(*list)[2] = 3; // get our IntList object, then call overloaded operator[]
```

The function parameter does not need to be an integer

```
class Stupid{
public:
void operator[] (std::string index);
};

void Stupid::operator[] (std::string index){
std::cout << index;
}

int main(){
Stupid stupid;
stupid["Hello, world!"];
```

Overloading the parenthesis operator

```
class Matrix{
public:
   double& operator()(int row, int col);
   const double& operator()(int row, int col) const; // for const objec
ts
};


double& Matrix::operator()(int row, int col){
    return data[row][col];
}
```

```
Matrix matrix;
    matrix(1, 2) = 4.5;
    std::cout << matrix(1, 2);
```

overload the () operator again

```
void Matrix::operator()()
{
    // reset all elements of the matrix to 0.0
    for (int row=0; row < 4; ++row)
        for (int col=0; col < 4; ++col)
            data[row][col] = 0.0;
}

Matrix matrix;
    matrix(1, 2) = 4.5;
    matrix(); // erase cMatrix
    std::cout << matrix(1, 2);
```

Having fun with functors

Operator() is also commonly overloaded to implement **functors** (or **funct ion object**), which are classes that operate like functions.

```
class Accumulator{
public:
    int operator() (int i) { return (m_counter += i); }
};

int main(){
    Accumulator acc;
    std::cout << acc(10) << std::endl; // prints 10
```

Quiz time

- Write a class that holds a string. Overload operator() to return the substring that starts at the index of the first parameter, and includes however many characters are in the second parameter.

- The following code should run:

```
int main()
{
    Mystring string("Hello, world!");
    std::cout << string(7, 5); // start at index 7 and return 5 characters

    return 0;
}
```

Overloading typecasts

- `int n = 5;`
- `double d = n; // int implicitly cast to a double`

```
class Cents{
public:
    // Overloaded int cast
    operator int() { return m_cents; }
};

Cents cents(7);
printInt(cents); // print 7
```

```
int cents = static_cast<int>(cents);
```

You can overload cast operators for any data type you wish, including your own user-defined data types!

```
class Dollars{
private:    int m_dollars;
public:
    Dollars(int dollars=0)  {
        m_dollars = dollars;
    }
     // Allow us to convert Dollars into Cents
     operator Cents() { return Cents(m_dollars * 100); }
};
```

```
void printCents(Cents cents)
{
    std::cout << cents; // cents will be implicitly cast to an int here
}

Dollars dollars(9);
printCents(dollars); // dollars will be implicitly cast to a Cents here
```

constructor

```
class Fraction{private: int m_numerator;    int m_denominator;
public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)    {
        assert(denominator != 0);
    }
    friend std::ostream& operator<<(std::ostream& out, const Fraction
&f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction &f1){
out << f1.m_numerator << "/" << f1.m_denominator;
return out;
}
```

Recapping the types of initialization

- `int x(5); // Direct initialize an integer`
- `Fraction fiveThirds(5, 3); // Direct initialize a Fraction, calls Fraction(int, int) constructor`

- `int x = 6; // Copy initialize an integer`
- `Fraction six = Fraction(6); // Copy initialize a Fraction, will call Fraction(6, 1)`
- `Fraction seven = 7; // Copy initialize a Fraction. The compiler will try to find a way to convert 7 to a Fraction, which will invoke the Fraction(7, 1) constructor.`

The copy constructor

- `Fraction fiveThirds(5, 3); // Direct initialize a Fraction, calls Fraction(int, int) constructor`
- `Fraction fCopy(fiveThirds); // Direct initialize -- with what constructor?`
- `std::cout << fCopy << '\n';`

- A **copy constructor** is a special type of constructor used to create a new object as a copy of an existing object.
- And much like a default constructor, if you do not provide a copy constructor for your classes, C++ will create a public copy constructor for you.
- **Memberwise initialization**

The copy constructor

```
 // Default constructor
Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator) {
        assert(denominator != 0);
    }

// Copy constructor
Fraction(const Fraction &fraction):
  m_numerator(fraction.m_numerator), m_denominator(fraction.m_de
nominator){
        std::cout << "Copy constructor called\n"; // just to prove it
works
}
```

Preventing copies

- prevent copies of our classes from being made by making the copy constructor private:

```
class Fraction{
private:
    Fraction(const Fraction &copy) :// Copy constructor (private)
        m_numerator(copy.m_numerator), m_denominator(copy.m_denominat
or){}

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {     assert(denominator != 0);  }

};
```

Copy initialization

- This statement uses copy initialization to initialize newly created integer variable x to the value of 5.
- `int x = 5;`

- `Fraction six = Fraction(6);`
- This form of copy initialization is evaluated the same way as the following:
- `Fraction six(Fraction(6));`
- this can potentially make calls to both Fraction(int, int) and the Fraction copy constructor
- it’s better to avoid copy initialization for classes, and use direct or uniform initialization instead.

Other places copy initialization is used

```
Fraction makeNegative(Fraction f) // ideally we should do this by const refere
nce
{
    f.setNumerator(-f.getNumerator());
    return f;
}

int main(){
    Fraction fiveThirds(5, 3);
    std::cout << makeNegative(fiveThirds);

    return 0;
}
```

Copy constructor called
Copy constructor called
-5/3

Converting constructors, explicit, and delete

- By default, C++ will treat any constructor as an implicit conversion operator.
- `std::cout << makeNegative(6); // note the integer here`

```
// Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator){
        assert(denominator != 0);
    }
```

- Constructors eligible to be used for implicit conversations are called **converting constructors**.
- Prior to C++11, only constructors taking one parameter could be converting constructors.
- However, with the new uniform initialization syntax in C++11, constructors taking multiple parameters can now be converting constructors.

explicit

```
class MyString{public:
// explicit keyword makes this constructor ineligible for implic
it conversions
explicit MyString(int x){ m_string.resize(x);}
};
int main(){
MyString x = 'x'; // compile error, since MyString(int) is now e
xplicit and nothing will match this
std::cout << x;
```

However, note that making a constructor explicit only prevents implicit conversions. Explicit conversion s (via direct or uniform initialization or explicit casts) are still allowed:

```
MyString x('x'); // allowed, even though MyString(int) is explicit
```

- In our MyString case, we really want to completely disallow ‘x’ from being converted to a string (whether implicit or explicit, since the results aren’t going to be intuitive). One way to partially do this is to add a MyString(char) constructor, and make it private:

```
private:
MyString(char) // objects of type MyString(char) can't be constr
ucted from outside the class
{ }
```

- However, this constructor can still be used from inside the class.
- A better way to resolve the issue is to use the “delete” keyword (introduced in C++11) to delete the function:

The delete keyword

```
class MyString
{
private:
std::string m_string;

public:
        MyString(char) = delete; // any use of this constructor
is an error
```

Overloading the assignment operator

- **Assignment vs Copy constructor**
  - The purpose of the copy constructor and the assignment operator are almost equivalent -- both copy one object to another.
  - However, the copy constructor initializes new objects,
  - whereas the assignment operator replaces the contents of existing objects.

- Overloading the assignment operator (operator=) is fairly straightforward
- …

```
// Default constructor
Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator){}

// Copy constructor
Fraction(const Fraction &copy) :
m_numerator(copy.m_numerator), m_denominator(copy.m_denominator)
{}

// Overloaded assignment
Fraction& operator= (const Fraction &fraction);
```

Overloading the assignment operator

```
// A simplistic implementation of operator= (see better implementation below)
Fraction& Fraction::operator= (const Fraction &fraction)
{
    // do the copy
    m_numerator = fraction.m_numerator;
    m_denominator = fraction.m_denominator;

    // return the existing object so we can chain this operator
    return *this;
}
```

Overloading the assignment operator

```
int main()
{
    Fraction f1(5,3);
    Fraction f2(7,2);
    Fraction f3(9,5);

    f1 = f2 = f3; // chained assignment

    return 0;
}
```

Issues due to self-assignment

```
int main(){
    MyString alex("Alex", 5); // Meet Alex
    alex = alex; // Alex is himself
    std::cout << alex; // Say your name, Alex
```

```
alex = alex; // Alex is himself
// A simplistic implementation of operator= (do not use)
MyString& MyString::operator= (const MyString &str)
{
    if (m_data) delete m_data;

    m_length = str.m_length;
    // copy the data from str to the implicit object
    m_data = new char[str.m_length];

    for (int i=0; i < str.m_length; ++i)
        m_data[i] = str.m_data[i];

        return *this; // return the existing object so we can chain this operator
}
```

You'll probably get garbage output (or a crash). What happened?

Detecting and handling self-assignment

```
// A better implementation of operator=
Fraction& Fraction::operator= (const Fraction &fraction)
{
    // self-assignment guard
    if (this == &fraction)
        return *this;

    // do the copy
    m_numerator = fraction.m_numerator;
    m_denominator = fraction.m_denominator;

    // return the existing object so we can chain this operator
    return *this;
}
```

Default assignment operator

- Unlike other operators, the compiler will provide a default public assignment operator for your class if you do not provide one.

- This assignment operator does memberwise assignment (which is essentially the same as the memberwise initialization that default copy constructors do).

- Just like other constructors and operators, you can prevent assignments from being made by making your assignment operator private or using the delete keyword:

```
// Overloaded assignment
Fraction& operator= (const Fraction &fraction) = delete; // no c
opies through assignment!
```

Shallow vs. deep copying

Shallow vs. deep copying

- Because C++ does not know much about your class, the default copy constructor and default assignment operators it provides use a copying method known as a memberwise copy (also known as a **shallow copy**.

- This means that C++ copies each member of the class individually (using the assignment operator for overloaded operator=, and direct initialization for the copy constructor).

- When classes are simple (e.g. do not contain any dynamically allocated memory), this works very well.

```
class MyString{
private:
    char *m_data;    int m_length;
public:
MyString(const char *source=""){
assert(source); // make sure source isn't a null string
// Plus one character for a terminator
m_length = strlen(source) + 1;

// Allocate a buffer equal to this length
m_data = new char[m_length];

// Copy the parameter string into our internal buffer
for (int i=0; i < m_length; ++i)    m_data[i] = source[i];
```

```
        // Make sure the string is terminated
        m_data[m_length-1] = '\n';
    }
    ~MyString() // destructor
    {
        // We need to deallocate our string
        delete[] m_data;
    }
    char* getString() { return m_data; }
    int getLength() { return m_length; }
};
```

shallow copy

- The above is a simple string class that allocates memory to hold a string that we pass in.

- Note that we have not defined a copy constructor or overloaded assignment operator.

- Consequently, C++ will provide a default copy constructor and default assignment operator that do a shallow copy.
- The copy constructor will look something like this:

```
MyString::MyString(const MyString &source) :
    m_length(source.m_length), m_data(source.m_data)
{}
```

Now, consider the following snippet of code:

```
int main()
{
    MyString hello("Hello, world!");
    {
        MyString copy = hello; // use default copy constructor
    } // copy gets destroyed here

    std::cout << hello.getString() << '\n'; // this will have un
defined behavior

    return 0;
}
```

Deep copying

```
MyString::MyString(const MyString& source){// Copy constructor
    m_length = source.m_length; // because m_length is not a pointer, we can shallow copy it
   // m_data is a pointer, so we need to deep copy it if it is non-null
    if (source.m_data)
    {
        m_data = new char[m_length];
        for (int i=0; i < m_length; ++i)
            m_data[i] = source[i];
    }
    else
        m_data = 0;
}
```

```
MyString& MyString::operator=(const MyString & source){// Assignment oper
ator
    // check for self-assignment
     if (this == &source)        return *this;
     delete[] m_data;  // first we need to deallocate any value that this string is holding!
     m_length = source.m_length;
    // m_data is a pointer, so we need to deep copy it if it is non-null
     if (source.m_data)     {
          m_data = new char[m_length];
          for (int i=0; i < m_length; ++i)  m_data[i] = source[i];
     }
     else m_data = 0;

     return *this;
}
```

Summary

- The default copy constructor and default assignment operators do shallow copies, which is fine for classes that contain no dynamically allocated variables.

- Classes with dynamically allocated variables need to have a copy constructor and assignment operator that do a deep copy.

- Favor using classes in the standard library over doing your own memory management.