Week 3

Chapter 5: Operator Overloading

Outline:

- Motivation
- Unary operator overloading
- Binary operator overloading
- Overloading the output << operator
- Restrictions

Motivation

- Operator overloading allows objects to become operands of existing operators (those provided by the syntax).
- The semantics of the operation is defined by the programmer.
- Operator overloading makes classes appear like primitive types.
- Java does not support operator overloading.

Unary Operator Overloading

- Let OP be a unary operator (++, --, !, *, &, ...) and obj be an object.
- C++ considers the following expressions equivalent

  OP obj;

  obj.operatorOP();
- That is, unary operator OP is simply a 0-parameter member function operatorOP that can be invoked with a unary operation syntax.
- Thus when operatorOP is implemented to work on obj, the function operatorOP is overloaded

Unary Operator Overloading Example (1)

```
$ cat uoo.cpp
#include <iostream>
using namespace std;
class P {
      int x, y;                                  // private: by default
public:
      P(int X=0, int Y=0): x(X), y(Y) {}
      void operator*() {                         // overload unary operator *
            x = ++x; y = --y;
      }
      void operator&() {                         // overload unary operator &
            cout << x << '\t' << y << endl;
      }
};
```

Unary Operator Overloading Example (2)

```
int main() {
    P p00;              // x and y are set to 0 by default
    p00.operator*();    // increment x and decrement y
    p00.operator&();    // print x and y
    P q00;
    *q00;               // Use unary operator *
    &q00;               // Use unary operator &
    return 0;
}
```

```
$ g++ uoo.cpp
$ a.out
1        -1
1        -1
```

Overloading the Prefix and Postfix ++ / -- Operators

- To distinguish the prefix member operator function from that of the postfix, the latter pretends to take an int parameter.
- To be consistent with the expected semantics, the prefix operator function should return the new value but the postfix operator function should return the original value.

Overloading ++ / -- Operators Example (1)

```
$ cat pp.cpp
#include <iostream>
using namespace std;
class P {
    int x, y;
public:
    P(int X=0, int Y=0): x(X), y(Y) {}
    P(const P & p): x(p.x), y(p.y) {}
    P operator++() { x++; y++; return *this; } // prefix
    P operator--() { x--; y--; return *this; }
    P operator++(int) { P orig = *this; x++; y++; return orig;} //postfix
    P operator--(int) { P orig = *this; x--; y--; return orig; }
    void show() { cout << x << ' ' << y << '\t'; }
};
```

Overloading ++ / -- Operators Example (2)

```
int main() {
    P p00, q;
    q = ++p00; q.show(); p00.show(); cout << endl;
    q = p00--; q.show(); p00.show(); cout << endl;
    q = --p00; q.show(); p00.show(); cout << endl;
    q = p00++; q.show(); p00.show(); cout << endl;
    return 0;
}
```

```
$ g++ pp.cpp
$ a.out
1 1       1 1
1 1       0 0
-1 -1     -1 -1
-1 -1     0 0
```

Binary Operator Overloading

- Let OP be a binary operator and obj1, obj2 be objects (not necessarily of the same class).
- C++ considers the following expressions equivalent

  obj1 OP obj2;

  obj1.operatorOP(obj2);
- That is, binary operator overloading is simply a 1-parameter member function that can be invoked with a binary operation syntax.
- The binary operator overloading can be implemented as a 1-parameter member function of obj1 or as a 2-parameter global function

Binary Operator Overloading Example (1)

```
$ cat bpp1.cpp
#include <iostream>
using namespace std;
class P {
    int x, y;
public:
    P(int X=0, int Y=0): x(X), y(Y) {}
```

Binary Operator Overloading Example (2)

```
    P operator+(const P & rhs) {
        P temp( x+rhs.x, y+rhs.y );
        return temp;
    }
    P & operator=(const P & rhs) {
        x = rhs.x; y = rhs.y;
        return *this;
    }
    ostream & operator<<(ostream & out) {
        return out << x << ' ' << y;
    }
};
```

Binary Operator Overloading Example (3)

```
int main() {
    P a, b(12,0), c(0,80);
    a.operator=(b.operator+(c)); // instead of writing
    a.operator<<(cout) << endl;
    a = b + c;                   // we can write
    a << cout << endl;           // Very weird!
    return 0;
}
```

```
$ g++ bpp1.cpp
$ a.out
12 80
12 80
```

Operator Overloading with Global Functions

- When the ouput operator << is overloaded as a 1-parameter member function, the call and operation expressions are:

  obj.operator<<(cout);

  obj << cout; // very weird!

- To be consistent with the established usage of <<, we would like to have the operation expression to look like

  cout << obj;

- This can be achieved by overloading operator<< with a global function

Overloading << with a Global Function (1)

```
#include <iostream>
using namespace std;
class P {
    int x, y;
public:
    P(int X=0, int Y=0): x(X), y(Y) {}
```

Overloading << with a Global Function (2)

```
    P operator+(const P & rhs) {
        P temp( x+rhs.x, y+rhs.y ); return temp;
    }
    P & operator=(const P & rhs) {
        x = rhs.x; y = rhs.y; return *this;
    }
    ostream & operator<<(ostream & out) const {
        return out << x << ' ' << y;
    }
};
```

Overloading << with a Global Function (3)

```
ostream & operator<<( ostream & out, const P & rhs ) {
    return rhs << out;
}
int main() {
    P a, b(12,0), c(0,80);
    a.operator=(b.operator+(c)); // instead of writing
    a.operator<<(cout) << endl;
    a = b + c;                   // we can write
    a << cout << endl; // using the member function
    cout << a << endl; // or using the global function
    return 0;
}
```

```
$ g++ bpp2.cpp
$ a.out
12 80
12 80
12 80
```

Operator Overloading with Global Functions Revisited

- In the above example, the operator<< is overloaded twice: as a member function and as a global function.
- These functions have different signatures so the code compiles.
- The member operator function could be replaced by an ordinary member function to forbid the operation expression

  obj << cout;

Overloading << with a Global Function Revisited (1)

```
$ cat out2.cpp
#include <iostream>
using namespace std;
class P {
      int x, y;
public:
      P(int X=0, int Y=0): x(X), y(Y) {}
```

Overloading << with a Global Function Revisited (2)

```
    P operator+(const P & rhs) {
        P temp( x+rhs.x, y+rhs.y );
        return temp;
    }
    P & operator=(const P & rhs) {
        x = rhs.x; y = rhs.y;
        return *this;
    }
    ostream & print (ostream & out) const {
        return out << x << ' ' << y;
    }
};
```

Overloading << with a Global Function Revisited (3)

```
ostream & operator<<( ostream & out, const P & rhs ) {
    return rhs.print(out);
}
int main() {
    P a, b(12,0), c(0,80);
    a.operator=(b.operator+(c)); // instead of writing
    a.print(cout) << endl;
    a = b + c;                   // we can write
    cout << a << endl;
    // a << cout << endl;        // invalid
    return 0;
}
```

```
$ g++ out2.cpp
$ a.out
12 80
12 80
```

Operator Overloading with Friend Functions

- When overloading an operator with a global function, the global function has to call a member function in order to access the private data of the object.
- To allow the global function access to private data, the global function can be made a friend of the class of the object.

Overloading << with a Friend Function (1)

```
$ cat out3.cpp
#include <iostream>
using namespace std;
class P {
    int x, y;
public:
    P(int X=0, int Y=0): x(X), y(Y) {}
```

Overloading << with a Friend Function (2)

```
    P operator+(const P & rhs) {
        P temp( x+rhs.x, y+rhs.y );
        return temp;
    }
    P & operator=(const P & rhs) {
        x = rhs.x; y = rhs.y;
        return *this;
    }
    friend ostream & operator<<(ostream &, const P &);
};
```

Overloading << with a Friend Function (3)

```
ostream & operator<<( ostream & out, const P & rhs ) {
    return out << rhs.x << ' ' << rhs.y;
}
int main() {
    P a, b(12,0), c(0,80);
    a = b + c;
    cout << a << endl;
    return 0;
}
```

```
$ g++ out3.cpp
$ a.out
12 80
```

Operator Overloading Restrictions

- All operators can be overloaded except four (., .*, ?:, sizeof) .
- The precedence, association, and arity (number of operands) of operators cannot be changed as they are fixed by the syntax of C++ .
- The overloaded operator can have any semantics determined by the programmer.
- But it is bad to overload an operator to have unexpected semantics. For example, we should not overload the comparison operator == to test for inequality though this is allowed.

Operator Overloading Restrictions Example

```
$ cat bad.cpp
#include <iostream>
using namespace std;
class P {
	int x, y;
public:
	P(int X=0, int Y=0): x(X), y(Y) {}
	bool operator==(const P & rhs) { return x != rhs.x || y != rhs.y;}
};                                  // quite puzzling
int main() {
	P a, b;
	if(a==b) cout << "unequal"; else cout << "equal";
	cout << endl;
	return 0;
}
```

```
$ g++ bad.cpp
$ a.out
equal
```

Chapter 6: Inheritance

Outline:

- Introduction
- Basics
- Polymorphism, virtual functions, dynamic dispatch
- Pure virtual functions, abstract base classes
- Multiple inheritance

Introduction

- C++ provides inheritance to model the “is-a” relationship. For example, a square is a shape, so is a circle.
- The commonality of objects is described as a base (aka super) class, and each specialization is described as a derived (aka sub) class that extends the base class.
- C++ supports multiple inheritance. That is, a class may derive from multiple base classes.

Syntax and Accessibility

- The basic syntax is

  class Sub : public Super { /* specialization */ };

- While Sub inherits all the members of Super, the accessibility of members of Super by member functions of Sub and the world is as follows:

Super Members	Private	Protected	Public
Sub Functions	No	Yes	Yes
World	No	No	Yes

Construction and Destruction

- Construction of a derived class object leads to the construction of a base class object automatically.
- Destruction of a derived class object leads to the destruction of a base class object automatically.

Accessibility, Construction, and Destruction Example (1)

```
$ cat inherit1.cpp
#include <iostream>
using namespace std;
class A {
        int x;
protected:
        int y;
        A(int X=1, int Y=2, int Z=80): x(X),y(Y),z(Z) {
                cout << "A()" << endl;
        }
        ~A() { cout << "~A()" << endl; }
public:
        int z;
};
```

Accessibility, Construction, and Destruction Example (2)

```
class B : public A {
public:
    B() { cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
    int getPro() { return y; }
    int getPub() { return z; }
};
int main() {
    B b;
    cout << b.getPro() << endl;
    cout << b.getPub() << endl;
    cout << b.z << endl;
    return 0;
}
```

```
$ g++ inherit1.cpp
$ a.out
A()
B()
2
80
80
~B()
~A()
```

A Quick Preview of Inheritance and Polymorphism (1)

```
$ cat inhpoly1.cpp
#include <iostream>
using namespace std;
class animal { public: void say(){cout << "I am an animal." << endl; } };
class cat : public animal { public: void say() { cout << "I am a cat." << endl; } };                // Subclass, inheritance
class dog : public animal { public: void say() { cout << "I am a dog." << endl; } };
class rat : public animal { public: void say() { cout << "I am a rat." << endl; } };
```

A Quick Preview of Inheritance and Polymorphism (2)

```
int main() {
    animal *p;
    cat c; dog d; rat r;
    p = &c; p->say();
    p = &d; p->say();
    p = &r; p->say();
    return 0;
}
```

```
$ g++ inhpoly1.cpp
$ a.out
I am an animal.
I am an animal.
I am an animal.
```

Polymorphism, Virtual Functions, Dynamic Dispatch

- Consider several derived classes of the same base class.
- Objects of these derived classes can be referred to as objects of the base class.
- To exhibit the specialization of a derived class, relevant functions of the base class should be declared as virtual.
- The virtual functions should be overriden in each of the derived classes.
- The ability to select the appropriate implementations of the virtual functions during run time is known as dynamic dispatch.

Example on Dynamic Dispatch(1)

```
$ cat inhpoly2.cpp
#include <iostream>
using namespace std;
class animal { public: virtual void say(){cout << "I am an animal."
    << endl; } };                          // virtual: polymorphic
class cat : public animal { public: void say() { cout << "I am a
    cat." << endl; } };                    // Subclass, inheritance
class dog : public animal { public: void say() { cout << "I am a
    dog." << endl; } };
class rat : public animal { public: void say() { cout << "I am a rat."
    << endl; } };
```

Example on Dynamic Dispatch(2)

```
int main() {
    animal *p;
    cat c; dog d; rat r;
    p = &c; p->say();
    p = &d; p->say();
    p = &r; p->say();
    return 0;
}
```

```
$ g++ inhpoly2.cpp
$ a.out
I am a cat.
I am a dog.
I am a rat.
```

Polymorphism Example (1)

```
$ cat poly.cpp
#include <iostream>
using namespace std;
class Poly {
protected:
    int u, v;                  // width, height or X, Y
    Poly(int U, int V): u(U), v(V) {}
public:
    virtual int getArea() { return 0; }
};
```

Polymorphism Example (2)

```
class Rect : public Poly {    // a rectangle is a polygon
public:
      Rect(int W=0, int H=0) : Poly(W,H) {}
      int getArea() { return u*v; }
};
class Tri : public Poly {     // a triangle is a polygon
public:
      Tri(int W=0, int H=0) : Poly(W,H) {}
      int getArea() { return u*v/2; }
};
class Pt : public Poly {      // a point is a degenerate polygon
public:
      Pt(int X=0, int Y=0) : Poly(X,Y) {}
};
```

Polymorphism Example (3)

```
int main() {
    Poly * shapes[6];
    shapes[0] = new Rect(12,80);
    shapes[1] = new Tri(12,80);
    shapes[2] = new Pt;
    shapes[3] = new Rect(80,12);
    shapes[4] = new Tri(80,12);
    shapes[5] = new Pt(12,80);
    for(int i=0; i<6; i++)
        cout << shapes[i]->getArea() << endl;
    return 0;
}
```

```
$ g++ poly.cpp
$ a.out
960
480
0
960
480
0
```

Pure Virtual Functions and Abstract Base Classes

- A pure virtual function is a virtual function that is undefined.
- To be undefined the body of the virtual function is set to zero:

  virtual typeName functionName(parameters) = 0;
- A class containing one or more pure virtual functions is an abstract base class.
- An abstract base class can be extended but cannot instantiate objects.

Abstract Base Classes Example (1)

```
$ cat abs.cpp
#include <iostream>
using namespace std;
class Poly {
protected:
    int u, v;                    // width, height or X, Y
    Poly(int U, int V): u(U), v(V) {}
public:
    virtual int getArea() = 0;
};
```

Abstract Base Classes Example (2)

```
class Rect : public Poly {                    // Rectangular
public:
    Rect(int W=0, int H=0) : Poly(W,H) {}
    int getArea() { return u*v; }
};
class Tri : public Poly {                     // Triangular
public:
    Tri(int W=0, int H=0) : Poly(W,H) {}
    int getArea() { return u*v/2; }
};
class Pt : public Poly {                      // Point
public:
    Pt(int X=0, int Y=0) : Poly(X,Y) {}
    int getArea() { return 0; }
};
```

Abstract Base Classes Example (3)

```
int main() {
    Poly * shapes[6];
    shapes[0] = new Rect(12,80);
    shapes[1] = new Tri(12,80);
    shapes[2] = new Pt;
    shapes[3] = new Rect(80,12);
    shapes[4] = new Tri(80,12);
    shapes[5] = new Pt(12,80);
    for(int i=0; i<6; i++)
        cout << shapes[i]->getArea() << endl;
    return 0;
}
```

```
$ g++ abs.cpp
$ a.out
960
480
0
960
480
0
```

Multiple Inheritance

- A class may inherit members from more than one class.
- To do so, simply separate the various base classes with comma in the declaration of the derived class.

Multiple Inheritance Example (1)

```
$ cat multi.cpp
#include <iostream>
#include <string>
using namespace std;
class Name {
        string name;
        protected: Name(string Who="unknown"): name(Who) {}
};
class Poly {
protected:
        int u, v;                       // width, height or X, Y
        Poly(int U=0, int V=0): u(U), v(V) {}
        public: virtual int getArea() { return 0; }
};
```

Multiple Inheritance Example (2)

```
class Rect : public Poly, public Name {          // Rectangular
public:
    Rect(int W, int H, string Who) : Poly(W,H), Name(Who) {}
    int getArea() { return u*v; }
};
class Tri : public Poly, public Name {           // Triangular
public:
    Tri(int W, int H, string Who) : Poly(W,H), Name(Who) {}
    int getArea() { return u*v/2; }
};
class Pt : public Poly, public Name {            // Point
public:
    Pt(int X, int Y, string Who) : Poly(X,Y), Name(Who) {}
};
```

Multiple Inheritance Example (3)

```
int main() {
    Poly * shapes[6];
    shapes[0] = new Rect(12,80,"Tall");
    shapes[1] = new Tri(12,80,"Delta");
    shapes[2] = new Pt(12,80,"Period");
    shapes[3] = new Rect(80,12,"Flat");
    shapes[4] = new Tri(80,12,"delta");
    shapes[5] = new Pt(80,12,"Dot");
    for(int i=0; i<6; i++)
        cout << shapes[i]->getArea() << endl;
    return 0;
}
```

```
$ g++ multi.cpp
$ a.out
960
480
0
960
480
0
```

Multiple Inheritance Example (4)

```
$ cat multi2.cpp
#include <iostream>
#include <string>
using namespace std;
class Name {
    string name;
protected:
    Name(string Who="unknown"): name(Who) {}
public:
    virtual string getName() { return name; }
};
```

Multiple Inheritance Example (5)

```
class Poly {
protected:
      int u, v;                    // width, height or X, Y
      Poly(int U=0, int V=0): u(U), v(V) {}
public:
      virtual int getArea() { return 0; }
};
class PolyName: public Poly, public Name {
public:
      PolyName (int W, int H, string Who) : Poly(W,H), Name(Who) {}
};
```

Multiple Inheritance Example (6)

```
class Rect : public PolyName {                    // Rectangular
public:
    Rect(int W, int H, string Who) : PolyName(W,H,Who) {}
    int getArea() { return u*v; }
};
class Tri : public PolyName {                     // Triangular
public:
    Tri(int W, int H, string Who) : PolyName(W,H,Who) {}
    int getArea() { return u*v/2; }
};
class Pt : public PolyName {                      // Point
public:
    Pt(int X, int Y, string Who) : PolyName(X,Y,Who) {}
    // int getArea() ?
};
```

Multiple Inheritance Example (7)

```
int main() {
    PolyName * shapes[6];       // What if Poly is used instead?
    shapes[0] = new Rect(12,80,"Tall");
    shapes[1] = new Tri(12,80,"Delta");
    shapes[2] = new Pt(12,80,"Period");
    shapes[3] = new Rect(80,12,"Flat");
    shapes[4] = new Tri(80,12,"delta");
    shapes[5] = new Pt(80,12,"Dot");
    for(int i=0; i<6; i++)
        cout << shapes[i]->getArea() << “\t"
        << shapes[i]->getName() << endl;
    return 0;
}
```

```
$ g++ multi2.cpp
$ a.out
960     Tall
480     Delta
0       Period
960     Flat
480     delta
0       Dot
```