

CS2311 Computer Programming

LT12 Object Oriented Programming-II

Computer Science, City University of Hong Kong

Semester A 2022-23

Outline

- Operator overloading
- Polymorphism

Operator Overloading

- Enabling C++'s operators to work with class objects
- Using traditional operators with **user-defined objects**
- Examples of already overloaded operators
 - Operator << is both the stream-insertion operator and the bitwise left-shift operator
 - + and -, perform arithmetic on multiple types

Operator Overloading

```
class Triangle {  
private:  
    double s1, s2, s3;  
    double area;  
public:  
    Triangle() {}  
    void setSides();  
    void computeArea();  
    double getArea();  
};
```

```
Triangle a, b;  
  
a.setSides();  
b.setSides();
```

lhs = Left
Hand Side

rhs = Right
Hand Side

```
if (a < b) {  
    cout << "Triangle a is smaller than triangle b\n";  
}
```

Operator Overloading

`+` `-` `*` `/` `%` `^` `&` `|` `~` `!` `,` `=` `<` `>` `<=` `>=`

`++` `--` `<<` `>>` `==` `!=` `&&` `||` `+=` `-=` `*=` `/=`

`%=` `^=` `&=` `|=` `<<=` `>>=` `[]` `()` `->`

`->*` `new` `new[]` `delete` `delete[]`

Operator Overloading

- Overloading an operator
 - Write function definition as normal
 - Function name is keyword **operator** followed by the symbol for the operator being overloaded
 - **operator+** used to overload the addition operator (+)
- Special operators
 - To use an operator on a class object it must be overloaded except the assignment operator(=) or the address operator(&)
 - Assignment operator by default performs member-wise assignment
 - Address operator (&) by default returns the address of an object

Operator Overloading: Member Function

- Add a function called operator _ (e.g., <, +, !) to your class:

```
class Circle {
private:
    int radius;
public:
    Circle(int radius): radius(radius) {};
    bool operator< (Circle& rhs);
    bool operator> (Circle& rhs);
    // lhs (left hand side) of each operator is this.
};

bool Circle::operator<(Circle& rhs) {
    if (radius < rhs.radius) return true;
    else return false;
}
```

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

int main() {
    Circle a(3);
    Circle b(5);
    cout << (a < b);
    return 0;
}

// a < b  $\leftrightarrow$  a.operator<(b)
```

Operator Overloading: Friend Function

- *Friend function*: a special function which is a non-member function of a class but has privilege to access private and protected data of that class
- Friend function can be declared in any section of the class i.e. public or private or protected
- When friend function is called neither name of object nor dot operator is used

Operator Overloading: Friend Function

```
class Triangle {  
private:  
    double s1, s2, s3;  
public:  
    Triangle() { s1=0; s2=0; s3=0; }  
    Triangle(double s1, double s2, double s3): s1(s1), s2(s2), s3(s3) {}  
    double getArea();  
    friend bool operator< (Triangle &lhs, Triangle &rhs);  
    friend bool operator> (Triangle &lhs, Triangle &rhs);  
    friend ostream& operator<< (ostream &outs, Triangle &c);  
};  
double Triangle::getArea() {  
    double s = (s1+s2+s3)/2;  
    return sqrt(s*(s-s1)*(s-s2)*(s-s3));  
}
```

Operator Overloading: Friend Function

```
bool operator<(Triangle &lhs, Triangle &rhs) {
    return lhs.getArea() < rhs.getArea();
}

bool operator>(Triangle &lhs, Triangle &rhs) {
    return lhs.getArea() > rhs.getArea();
}

ostream &operator << (ostream &outs, Triangle &t) {
    outs << "The sides are: ";
    outs << t.s1 << " " << t.s2 << " " << t.s3 << " ";
    outs << "The area is: ";
    outs << t.getArea() << endl;
    return outs;
}
```

```
int main() {
    Triangle t1(3, 4, 5);
    Triangle t2(5, 6, 7);
    cout << t1;
    cout << t2;
    if (t1 < t2) {
        cout << "t1 is smaller\n";
    } else {
        cout << "t2 is smaller\n";
    }
    return 0;
}
```

Outline

- Operator overloading
- Polymorphism

Type Casting in Class Inheritance

```
class Animal {  
    ...  
};  
class Human : public Animal {  
    ...  
};
```

// Only *down-casting* is allowed in class type conversion

// You can say a human is an animal, but not vice versa

```
Animal *a = new Human();    // legal
```

```
Human *b = new Animal();    // illegal
```

Static Type vs Dynamic Type

- *Static type*: the declared type
- *Dynamic type*: the actual type assigned

```
class Animal {  
    ...  
};  
  
class Human : public Animal {  
    ...  
};  
  
class Dog : public Animal {  
    ...  
};
```

```
int main() {  
    Human *human = new Human();  
    Dog *dog = new Dog();  
  
    Animal *a; // the static type of a is Animal  
    a = human; // the dynamic type of a is Human  
    a = dog;   // the dynamic type of a is Dog  
  
    delete human;  
    delete dog;  
    return 0;  
}
```

Override

- To re-implement a base class's member function by writing a **new version** of that function (with the same function prototype) in a **derived class**

```
class Shape {
public:
    void print() { cout << "I am a shape\n"; }
};
class Circle: public Shape {
private:
    double radius;
public:
    Circle(double radius=0):radius(radius) {};
    void print() { cout << "I am a circle and my radius is " << radius << "\n"; }
};
```

Override vs Overload

- Overload

```
double sum(double, double, double);  
double sum(double, double);
```

- Override

```
void Animal::makeSound();  
void Human::makeSound();  
void Dog::makeSound();
```

Polymorphism

- Polymorphism means "many forms"
- In inherited classes, the same function behaves differently depending on types

```
void Animal::makeSound();
```

```
void Human::makeSound();
```

```
void Duck::makeSound();
```

```
void Dog::makeSound();
```

```
void Cat::makeSound();
```



Polymorphism: Static Binding

- The called function is determined by *static type*

```
class Animal {  
    public:  
    void sayHi() {  
        cout << "...\\n";  
    }  
};
```

```
class Human : public Animal {  
    public:  
    void sayHi() {  
        cout << "hi\\n";  
    }  
};
```

```
class Dog : public Animal {  
    public:  
    void sayHi() {  
        cout << "wow wow\\n";  
    }  
};
```

```
int main() {  
    Human *human = new Human();  
    Dog *dog = new Dog();  
    Animal *a;  
  
    // the static type of a is Animal  
    a = human;  
    a->sayHi(); // will print "..."  
    a = dog;  
    a->sayHi(); // will print "..."  
  
    delete human;  
    delete dog;  
    return 0;  
}
```



Polymorphism: Dynamic Binding

- We want the called function to be determined by *dynamic type*

```
int main() {  
    Human *human = new Human();  
    Dog *dog = new Dog();  
  
    Animal *a;           // the static type of a is Animal  
    a = human;           // the dynamic type of a is Human  
    a->sayHi();           // we want it to print "Hi"  
    a = dog;             // the dynamic type of a is Dog  
    a->sayHi();           // we want it to print "wow wow"  
  
    delete human;  
    delete dog;  
    return 0;  
}
```

Dynamic Binding: Virtual Function

- A *virtual function* is declared in the base class using the keyword `virtual` and is re-defined (Overridden) in the derived class
- Virtual functions are Dynamic in nature
- Allows dynamic binding at runtime

Polymorphism: Dynamic Binding

```
class Base {  
public:  
    virtual void print() {  
        cout << "print base\n";  
    }  
    void show() {  
        cout << "show base\n";  
    }  
};
```

```
class Derived : public Base {  
public:  
    void print() {  
        cout << "print derived\n";  
    }  
    void show() {  
        cout << "show derived\n";  
    }  
};
```

```
int main() {  
    Base *base;  
    Derived *derived = new Derived();  
  
    base = derived;  
    base->print(); // dynamic binding  
                  // will print "print derived"  
  
    base->show(); // static binding  
                 // will print "show base"  
  
    delete derived;  
    return 0;  
}
```

Exercise

What is printed?

- (A) "Mammal"
- (B) "Cat"
- (C) "Siamese"
- (D) Gives an error
(identify compiler or crash)
- (E) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};

class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};

class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

Siamese * s = new Siamese();
s->printString();
```

Exercise

What is printed?

- (A) "Mammal"
- (B) "Cat"
- (C) "Siamese"
- (D) Gives an error
(identify compiler or crash)
- (E) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};

class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};

class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

Siamese * s = new Mammal();
s->printString();
```

Exercise

What is printed?

- (A) "Mammal"
- (B) "Cat"
- (C) "Siamese"
- (D) Gives an error
(identify compiler or crash)
- (E) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};

class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};

class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

Mammal * s = new Mammal();
s->printString();
```

Exercise

What is printed?

- (A) "Mammal"
- (B) "Cat"
- (C) "Siamese"
- (D) Gives an error
(identify compiler or crash)
- (E) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};

class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};

class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

Mammal * s = new Siamese();
s->printString();
```