

OPERATOR OVERLOADING

Lecturer: 陈笑沙

TABLE OF CONTENTS

- 10.1 Necessity of Operators
- 10.2 How to Overload Operators
- 10.3 Return Value vs. Return Reference
- 10.4 Overloading Increment Operators
- 10.5 Conversion Operators
- 10.6 Assignment Operators

10.1 NECESSITY OF OPERATORS

10.1 NECESSITY OF OPERATORS

- In C++, we not only need to use basic data types, but also to design new data types—class types.

10.1 NECESSITY OF OPERATORS

- In C++, we not only need to use basic data types, but also to design new data types—class types.
- In general, operations on basic data types are expressed using operators, which are intuitive and have simple semantics.

10.1 NECESSITY OF OPERATORS

- In C++, we not only need to use basic data types, but also to design new data types—class types.
- In general, operations on basic data types are expressed using operators, which are intuitive and have simple semantics.
 - $a = b + c$

10.1 NECESSITY OF OPERATORS

- In C++, we not only need to use basic data types, but also to design new data types—class types.
- In general, operations on basic data types are expressed using operators, which are intuitive and have simple semantics.
 - $a = b + c$
 - Internal data types all have predefined operators (operands)

10.1 NECESSITY OF OPERATORS

10.1 NECESSITY OF OPERATORS

- C++ can define custom data types and their operations through member functions.

10.1 NECESSITY OF OPERATORS

- C++ can define custom data types and their operations through member functions.
 - Matrix $a + b$ implementation calls `a.add(b)`.

10.1 NECESSITY OF OPERATORS

- C++ can define custom data types and their operations through member functions.
 - Matrix $a + b$ implementation calls `a.add(b)`.
 - Alternatively, `Matrix::add(a, b)`.

10.1 NECESSITY OF OPERATORS

- C++ can define custom data types and their operations through member functions.
 - Matrix $a + b$ implementation calls `a.add(b)`.
 - Alternatively, `Matrix::add(a, b)`.
- C++ allows defining operators within custom classes.

10.1 NECESSITY OF OPERATORS

- C++ can define custom data types and their operations through member functions.
 - Matrix $a + b$ implementation calls `a.add(b)`.
 - Alternatively, `Matrix::add(a, b)`.
- C++ allows defining operators within custom classes.
 - Define the `+` operator for matrices.

10.1 NECESSITY OF OPERATORS

10.1 NECESSITY OF OPERATORS

- If the operator is directly applied to a class type, what happens?

10.1 NECESSITY OF OPERATORS

- If the operator is directly applied to a class type, what happens?
 - Complex ret, c1,c2; $\text{ret} = \text{c1} + \text{c2};$

10.1 NECESSITY OF OPERATORS

- If the operator is directly applied to a class type, what happens?
 - Complex `ret, c1,c2; ret=c1+c2;`
- The compiler will not recognize the semantics of the operator.

10.1 NECESSITY OF OPERATORS

- If the operator is directly applied to a class type, what happens?
 - Complex `ret, c1,c2; ret=c1+c2;`
- The compiler will not recognize the semantics of the operator.
- A mechanism is needed to redefine the meaning of the operator acting on class types.

10.1 NECESSITY OF OPERATORS

- If the operator is directly applied to a class type, what happens?
 - Complex `ret, c1,c2; ret=c1+c2;`
- The compiler will not recognize the semantics of the operator.
- A mechanism is needed to redefine the meaning of the operator acting on class types.
- This mechanism is operator overloading.

10.2 HOW TO OVERLOAD OPERATORS

```
1 #include <iostream>
2 #include <iomanip>
3
4 class Complex {
5 public:
6     Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
7     static Complex add(const Complex &c1, const Complex &c2) {
8         Complex r;
9         r.re = c1.re + c2.re;
10        r.im = c1.im + c2.im;
11        return r;
12    }
13
14    std::string to_string() const {
15        std::ostringstream oss;
16        oss << "(" << re << " + " << im << "i)";
17        return oss.str();
18    }
19}
```

10.2 HOW TO OVERLOAD OPERATORS

```
19     oss << im;
20 }
21 oss << "i";
22 }
23 return oss.str();
24 }
25
26 private:
27 double re, im;
28 };
29
30 int main() {
31     Complex c1{1, 2};
32     Complex c2{3, 4};
33     auto c3 = Complex::add(c1, c2);
34     cout << c3 << endl;
```

10.2 HOW TO OVERLOAD OPERATORS

```
1 #include <iostream>
2 #include <iomanip>
3
4 class Complex {
5 public:
6     Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
7     static Complex add(const Complex &c1, const Complex &c2) {
8         Complex r;
9         r.re = c1.re + c2.re;
10        r.im = c1.im + c2.im;
11        return r;
12    }
13
14    std::string to_string() const {
15        std::ostringstream oss;
16        oss << "(" << re << ", " << im << ")";
17        return oss.str();
18    }
19}
```

10.2 HOW TO OVERLOAD OPERATORS

```
11     return r;
12 }
13
14 std::string to_string() const {
15     std::ostringstream oss;
16     oss << re << " + ";
17     if (im != 0) {
18         if (im != 1) {
19             oss << im;
20         }
21         oss << "i";
22     }
23     return oss.str();
24 }
25
26     . . .
27 }
```

10.2 HOW TO OVERLOAD OPERATORS

```
21     oss << "i";
22 }
23 return oss.str();
24 }
25
26 private:
27     double re, im;
28 };
29
30 int main() {
31     Complex c1{1, 2};
32     Complex c2{3, 4};
33     auto c3 = Complex::add(c1, c2);
34     std::cout << c3.to_string() << std::endl;
35     return 0;
36 }
```

10.2 HOW TO OVERLOAD OPERATORS

Example Interpretation: example/lec10/complex

10.2 HOW TO OVERLOAD OPERATORS

10.2 HOW TO OVERLOAD OPERATORS

- Actually, you can also:

10.2 HOW TO OVERLOAD OPERATORS

- Actually, you can also:
 - Declare Complex operator+(Complex& c1, Complex &c2) as a class member function.

10.2 HOW TO OVERLOAD OPERATORS

- Actually, you can also:
 - Declare Complex operator+(Complex& c1, Complex &c2) as a class member function.
 - $c2 = c1 + 27$ is equivalent to $c2 = c1.operator+(Complex\{27\})$

10.2 HOW TO OVERLOAD OPERATORS

- Actually, you can also:
 - Declare Complex operator+(Complex& c1, Complex &c2) as a class member function.
 - $c2 = c1 + 27$ is equivalent to $c2 = c1.operator+(Complex\{27\})$
 - At this point, $c2 = 27 + c1$ will result in an error

10.2 HOW TO OVERLOAD OPERATORS

You can also use friends:

```
1 class Complex{  
2     double re, im;  
3     friend Complex operator+(const Complex& c1, const Complex& c2);  
4 };
```

10.3 RETURN VALUE VS. RETURN REFERENCE

10.3 RETURN VALUE VS. RETURN REFERENCE

- If you want to output a complex number object, what should you do?

10.3 RETURN VALUE VS. RETURN REFERENCE

- If you want to output a complex number object, what should you do?
 - `cout << c.to_string() << endl`

10.3 RETURN VALUE VS. RETURN REFERENCE

- If you want to output a complex number object, what should you do?
 - `cout << c.to_string() << endl`
- Can we directly `cout << c << endl`?

10.3 RETURN VALUE VS. RETURN REFERENCE

```
1 #include <iostream>
2 #include <ostream>
3
4 using namespace std;
5
6 class Complex {
7 public:
8     Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
9
10    friend Complex operator+(const Complex &c1, const Complex &c2);
11
12    friend ostream &operator<<(ostream &out, const Complex &c);
13
14 private:
15     double re, im;
16 }
```

10.3 RETURN VALUE VS. RETURN REFERENCE

```
4 using namespace std;  
5  
6 class Complex {  
7 public:  
8     Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}  
9  
10    friend Complex operator+(const Complex &c1, const Complex &c2);  
11  
12    friend ostream &operator<<(ostream &out, const Complex &c);  
13  
14 private:  
15     double re, im;  
16 };  
17  
18 Complex operator+(const Complex &c1, const Complex &c2) {  
19 }
```

10.3 RETURN VALUE VS. RETURN REFERENCE

```
4 using namespace std;  
5  
6 class Complex {  
7 public:  
8     Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}  
9  
10    friend Complex operator+(const Complex &c1, const Complex &c2);  
11  
12    friend ostream &operator<<(ostream &out, const Complex &c);  
13  
14 private:  
15     double re, im;  
16 };  
17  
18 Complex operator+(const Complex &c1, const Complex &c2) {  
19 }
```

10.3 RETURN VALUE VS. RETURN REFERENCE

```
11
12     friend ostream &operator<<(ostream &out, const Complex &c);
13
14 private:
15     double re, im;
16 };
17
18 Complex operator+(const Complex &c1, const Complex &c2) {
19     return {c1.re + c2.re, c1.im + c2.im};
20 }
21
22 ostream &operator<<(ostream &out, const Complex &c) {
23     if (c.re != 0)
24         out << c.re;
25     if (c.im != 0) {
26         out << " + ";
27         if (c.im > 0)
28             out << c.im;
29         else
30             out << c.im << "i";
31     }
32 }
```

10.3 RETURN VALUE VS. RETURN REFERENCE

```
20  }
21
22 ostream &operator<<(ostream &out, const Complex &c) {
23     if (c.re != 0)
24         out << c.re;
25     if (c.im != 0) {
26         if (c.re != 0)
27             out << " + ";
28         if (c.im != 1) {
29             out << c.im;
30         }
31         out << "i";
32     }
33     return out;
34 }
```

10.3 RETURN VALUE VS. RETURN REFERENCE

```
28     if (c.im != 1) {  
29         out << c.im;  
30     }  
31     out << "i";  
32 }  
33 return out;  
34 }  
35  
36 int main(int argc, char **argv) {  
37     cout << Complex{1, 2} + Complex{-1, 2} << endl;  
38     cout << Complex{1, 0} << endl;  
39     cout << Complex{1, 1} << endl;  
40     cout << Complex{0, 1} << endl;  
41     cout << Complex{0, 4} << endl;  
42     return 0;  
43 }
```

10.3 RETURN VALUE VS. RETURN REFERENCE

Thinking: Why does std::ostream need to return a reference? Why can't the function parameter be const?

10.4 OVERLOADING INCREMENT OPERATORS

10.4 OVERLOADING INCREMENT OPERATORS

- Class objects need to implement self-increment, self-decrement operations, and also need to perform operator overloading.

10.4 OVERLOADING INCREMENT OPERATORS

- Class objects need to implement self-increment, self-decrement operations, and also need to perform operator overloading.
- How to distinguish between prefix and postfix?

10.4 OVERLOADING INCREMENT OPERATORS

- Class objects need to implement self-increment, self-decrement operations, and also need to perform operator overloading.
- How to distinguish between prefix and postfix?
- What are the function prototypes for self-increment and self-decrement?

10.4 OVERLOADING INCREMENT OPERATORS

10.4 OVERLOADING INCREMENT OPERATORS

- Increment operators include prefix increment, postfix increment, prefix decrement, and postfix decrement

10.4 OVERLOADING INCREMENT OPERATORS

- Increment operators include prefix increment, postfix increment, prefix decrement, and postfix decrement
 - `int a=3;`

10.4 OVERLOADING INCREMENT OPERATORS

- Increment operators include prefix increment, postfix increment, prefix decrement, and postfix decrement
 - `int a=3;`
 - `a++;` //postfix increment

10.4 OVERLOADING INCREMENT OPERATORS

- Increment operators include prefix increment, postfix increment, prefix decrement, and postfix decrement
 - `int a=3;`
 - `a++;` //postfix increment
 - `++a;` //prefix increment

10.4 OVERLOADING INCREMENT OPERATORS

- Increment operators include prefix increment, postfix increment, prefix decrement, and postfix decrement
 - `int a=3;`
 - `a++;` //postfix increment
 - `++a;` //prefix increment
 - `--a;` //prefix decrement

10.4 OVERLOADING INCREMENT OPERATORS

- Increment operators include prefix increment, postfix increment, prefix decrement, and postfix decrement
 - int a=3;
 - a++; //postfix increment
 - ++a; //prefix increment
 - --a; //prefix decrement
 - a--; //postfix decrement

10.4 OVERLOADING INCREMENT OPERATORS

- Increment operators include prefix increment, postfix increment, prefix decrement, and postfix decrement
 - `int a=3;`
 - `a++;` //postfix increment
 - `++a;` //prefix increment
 - `--a;` //prefix decrement
 - `a--;` //postfix decrement
- Applying prefix increment results in a left-value expression

10.4 OVERLOADING INCREMENT OPERATORS

- Increment operators include prefix increment, postfix increment, prefix decrement, and postfix decrement
 - `int a=3;`
 - `a++;` //postfix increment
 - `++a;` //prefix increment
 - `--a;` //prefix decrement
 - `a--;` //postfix decrement
- Applying prefix increment results in a left-value expression
- Applying postfix increment results in a right-value expression

10.4 OVERLOADING INCREMENT OPERATORS

- Increment operators include prefix increment, postfix increment, prefix decrement, and postfix decrement
 - int a=3;
 - a++; //postfix increment
 - ++a; //prefix increment
 - --a; //prefix decrement
 - a--; //postfix decrement
- Applying prefix increment results in a left-value expression
- Applying postfix increment results in a right-value expression
- Overloading prefix++ and postfix++, both have only one operand, hence corresponding to one parameter.

10.4 OVERLOADING INCREMENT OPERATORS

10.4 OVERLOADING INCREMENT OPERATORS

- Operators are all operator++

10.4 OVERLOADING INCREMENT OPERATORS

- Operators are all operator++
- Overload prefix++ to return reference, overload postfix++ to return value

10.4 OVERLOADING INCREMENT OPERATORS

- Operators are all operator++
- Overload prefix++ to return reference, overload postfix++ to return value
- C++ distinguishes between prefix++ and postfix++ overloads by adding an integer parameter for postfix++

10.4 OVERLOADING INCREMENT OPERATORS

- Operators are all operator++
- Overload prefix++ to return reference, overload postfix++ to return value
- C++ distinguishes between prefix++ and postfix++ overloads by adding an integer parameter for postfix++
 - `T &operator++(T& a); // prefix++`

10.4 OVERLOADING INCREMENT OPERATORS

- Operators are all operator++
- Overload prefix++ to return reference, overload postfix++ to return value
- C++ distinguishes between prefix++ and postfix++ overloads by adding an integer parameter for postfix++
 - `T &operator++(T& a); // prefix++`
 - `T operator++(T& a, int); // postfix++`

10.4 OVERLOADING INCREMENT OPERATORS

```
1 #include <iostream>
2 using namespace std;
3
4 class Increase {
5     int value;
6
7 public:
8     Increase(int x) : value(x) {}
9     Increase &operator++() { // Prefix increment (no parameters)
10        value++;           // Increment first
11        return *this;       // Return the original object
12    }
13
14    Increase operator++(int) {
15        // Post-increment (only one marker parameter int)
16        Increase temp(*this);
```

10.4 OVERLOADING INCREMENT OPERATORS

```
1 #include <iostream>
2 using namespace std;
3
4 class Increase {
5     int value;
6
7 public:
8     Increase(int x) : value(x) {}
9     Increase &operator++() { // Prefix increment (no parameters)
10        value++;           // Increment first
11        return *this;       // Return the original object
12    }
13
14    Increase operator++(int) {
15        // Post-increment (only one marker parameter int)
16        Increase temp(*this);
```

10.4 OVERLOADING INCREMENT OPERATORS

```
3  
4 class Increase {  
5     int value;  
6  
7 public:  
8     Increase(int x) : value(x) {}  
9     Increase &operator++() { // Prefix increment (no parameters)  
10        value++;           // Increment first  
11        return *this;       // Return the original object  
12    }  
13  
14    Increase operator++(int) {  
15        // Post-increment (only one marker parameter int)  
16        Increase temp(value);  
17        // Construct a temporary object to store the original object value  
18        value++;           // Change the original object value  
19        return temp;        // Return the temporary object  
20    }
```

10.4 OVERLOADING INCREMENT OPERATORS

```
10    value++;           // Increment first
11    return *this;     // Return the original object
12 }
13
14 Increase operator++(int) {
15     // Post-increment (only one marker parameter int)
16     Increase temp(value);
17     // Construct a temporary object to store the original object value
18     value++;    // Change the original object value
19     return temp; // Return the original object value
20 }
21
22 void display() { cout << "the value is " << value << endl; }
23 };
24
25 // Test program
26 int main() {
```

10.4 OVERLOADING INCREMENT OPERATORS

```
14 Increase operator++(int) {  
15     // Post-increment (only one marker parameter int)  
16     Increase temp(value);  
17     // Construct a temporary object to store the original object value  
18     value++;    // Change the original object value  
19     return temp; // Return the original object value  
20 }  
21  
22 void display() { cout << "the value is " << value << endl; }  
23 };  
24  
25 int main() {  
26     Increase n(20);  
27     n.display();  
28  
29     // Enter the value 10  
30 }
```

10.4 OVERLOADING INCREMENT OPERATORS

```
20  }
21
22 void display() { cout << "the value is " << value << endl; }
23 };
24
25 int main() {
26     Increase n(20);
27     n.display();
28
29     (n++++).display();
30     n.display();
31
32     ++(++n);
33     n.display();
34     return 0;
35 }
```

10.4 OVERLOADING INCREMENT OPERATORS

THE USE OF THE `this` POINTER

10.4 OVERLOADING INCREMENT OPERATORS

THE USE OF THE `this` POINTER

- The `this` pointer of an object is not part of the object itself and does not affect the result of `sizeof(object)`.

10.4 OVERLOADING INCREMENT OPERATORS

THE USE OF THE `this` POINTER

- The `this` pointer of an object is not part of the object itself and does not affect the result of `sizeof(object)`.
- The scope of `this` is within the class. When accessing non-static members of the class in non-static member functions, the compiler automatically passes the address of the object itself as an implicit parameter to the function.

10.4 OVERLOADING INCREMENT OPERATORS

THE USE OF THE `this` POINTER

- The `this` pointer of an object is not part of the object itself and does not affect the result of `sizeof(object)`.
- The scope of `this` is within the class. When accessing non-static members of the class in non-static member functions, the compiler automatically passes the address of the object itself as an implicit parameter to the function.
- In other words, even if you don't write the `this` pointer, the compiler adds it during compilation. It serves as an implicit formal parameter for non-static member functions and accesses all members through `this`.

10.4 OVERLOADING INCREMENT OPERATORS

THE USE OF THE `this` POINTER

10.4 OVERLOADING INCREMENT OPERATORS

THE USE OF THE `this` POINTER

- One case is when returning a class object itself in a non-static member function of the class, using `return *this;` directly

10.4 OVERLOADING INCREMENT OPERATORS

THE USE OF THE `this` POINTER

- One case is when returning a class object itself in a non-static member function of the class, using `return *this;` directly
- Another case is when the parameter name is the same as the member variable name, such as `this->n = n`

10.4 OVERLOADING INCREMENT OPERATORS

GENERAL FUNCTION FORM

```
1 #include<iostream>
2 using namespace std;
3 class Increase{
4     int value;
5 public:
6     Increase(int x) : value(x) {}
7     friend Increase &operator++(Increase& a);
8     friend Increase operator++(Increase &a, int);
9     void display() { cout<< "the value is " << value << endl; }
10 };
11
12 Increase& operator++(Increase& a){
13     a.value++;
14     return a;
15 }
16
17 Increase operator++(Increase& a, int){
18     Increase temp(a);
19     a.value++;
20     return temp;
21 }
```

10.4 OVERLOADING INCREMENT OPERATORS

GENERAL FUNCTION FORM

```
1 #include<iostream>
2 using namespace std;
3 class Increase{
4     int value;
5 public:
6     Increase(int x) : value(x) {}
7     friend Increase &operator++(Increase& a);
8     friend Increase operator++(Increase &a, int);
9     void display() { cout<< "the value is " << value << endl; }
10 };
11
12 Increase& operator++(Increase& a){
13     a.value++;
14     return a;
15 }
16
17 Increase operator++(Increase& a, int){
18     Increase temp(a);
19     a.value++;
20     return temp;
21 }
```

10.4 OVERLOADING INCREMENT OPERATORS

GENERAL FUNCTION FORM

```
1 #include<iostream>
2 using namespace std;
3 class Increase{
4     int value;
5 public:
6     Increase(int x) : value(x) {}
7     friend Increase &operator++(Increase& a);
8     friend Increase operator++(Increase &a, int);
9     void display() { cout<< "the value is " << value << endl; }
10 };
11
12 Increase& operator++(Increase& a){
13     a.value++;
14     return a;
15 }
16
17 Increase operator++(Increase& a, int){
18     Increase temp(a);
19     a.value++;
20     return temp;
21 }
```

10.4 OVERLOADING INCREMENT OPERATORS

GENERAL FUNCTION FORM

```
1 #include<iostream>
2 using namespace std;
3 class Increase{
4     int value;
5 public:
6     Increase(int x) : value(x) {}
7     friend Increase &operator++(Increase& a);
8     friend Increase operator++(Increase &a, int);
9     void display() { cout<< "the value is " << value << endl; }
10 };
11
12 Increase& operator++(Increase& a){
13     a.value++;
14     return a;
15 }
16
17 Increase operator++(Increase& a, int){
18     Increase temp(a);
19     a.value++;
20     return temp;
21 }
```

10.5 CONVERSION OPERATORS

```
1 #include<iostream>
2 using namespace std;
3
4 class RMB {
5     unsigned int yuan, jf; // yuan jiao fen
6 public:
7     RMB(double d=0) : yuan(d) , jf(int(d*100+0.5)%100){}
8
9     RMB(int y, int f):yuan(y), jf(f) {}
10    friend RMB operator+(const RMB&, const RMB&);
11    friend RMB& operator++(RMB&);
12    void display(){ cout<<(yuan + jf / 100.0)<<endl; }
13 };
14
15 RMB operator+(const RMB& s1, const RMB& s2){
```

10.5 CONVERSION OPERATORS

```
1 #include<iostream>
2 using namespace std;
3
4 class RMB {
5     unsigned int yuan, jf; // yuan jiao fen
6 public:
7     RMB(double d=0) : yuan(d) , jf(int(d*100+0.5)%100){}
8
9     RMB(int y, int f):yuan(y), jf(f) {}
10    friend RMB operator+(const RMB&, const RMB&);
11    friend RMB& operator++(RMB&);
12    void display(){ cout<<(yuan + jf / 100.0)<<endl; }
13 };
14
15 RMB operator+(const RMB& s1, const RMB& s2){
16     unsigned int x = s1.yuan + s2.yuan;
```

10.5 CONVERSION OPERATORS

```
9     RMB(y, s1.yuan, s1.jf);
10    friend RMB operator+(const RMB&, const RMB&);
11    friend RMB& operator++(RMB&);
12    void display(){ cout<<(yuan + jf / 100.0)<<endl; }
13 };
14
15 RMB operator+(const RMB& s1, const RMB& s2){
16     unsigned int x = s1.jf + s2.jf;
17     unsigned int yuan = s1.yuan + s2.yuan + x/100;
18     return RMB(yuan, x % 100);
19 }
20
21 RMB& operator++(RMB& s){
22     s.yuan += ++s.jf/100;
23     s.jf %= 100;
24     return s;
```

10.5 CONVERSION OPERATORS

```
15 RMB operator+(const RMB& s1, const RMB& s2) {
16     unsigned int x = s1.jf + s2.jf;
17     unsigned int yuan = s1.yuan + s2.yuan + x/100;
18     return RMB(yuan, x % 100);
19 }
20
21 RMB& operator++(RMB& s){
22     s.yuan += ++s.jf/100;
23     s.jf %= 100;
24     return s;
25 }
26
27 int main(){
28     //Convert floating point number to RMB object
29     // (floating point type to RMB class)
30     RMB w(12.567);
31 }
```

10.5 CONVERSION OPERATORS

```
18     return RMB(yuan, (s.jf * 100),
19 }
20
21 RMB& operator++(RMB& s){
22     s.yuan += ++s.jf/100;
23     s.jf %= 100;
24     return s;
25 }
26
27 int main(){
28     //Convert floating point number to RMB object
29     // (floating point type to RMB class)
30     RMB w(12.567);
31     ++w;
32     w.display();
33 }
```

10.5 CONVERSION OPERATORS

```
1 class RMB{  
2     unsigned int yuan, jf; // yuan jiao fen  
3 public:  
4     RMB(double value = 0.0) : yuan(value) {  
5         jf = (value - yuan) * 100 + 0.5;  
6     }  
7     operator double(){ //conversion operator converts to  
8         return yuan + jf/100.0;  
9     }  
10  
11    void display(){ cout<<(yuan + jf/100.0)<<endl; }  
12 };  
13  
14 int main(){  
15     RMB d1(2.0), d2(1.5), d3; //constructor converts to RMB
```

10.5 CONVERSION OPERATORS

```
1 class RMB{  
2     unsigned int yuan, jf; // yuan jiao fen  
3 public:  
4     RMB(double value = 0.0) : yuan(value) {  
5         jf = (value - yuan) * 100 + 0.5;  
6     }  
7     operator double(){ //conversion operator converts to  
8         return yuan + jf/100.0;  
9     }  
10  
11    void display(){ cout<<(yuan + jf/100.0)<<endl; }  
12 };  
13  
14 int main(){  
15     RMB d1(2.0), d2(1.5), d3; //constructor converts to RMB  
16 }
```

10.5 CONVERSION OPERATORS

```
1 class RMB{  
2     unsigned int yuan, jf; // yuan jiao fen  
3 public:  
4     RMB(double value = 0.0) : yuan(value) {  
5         jf = (value - yuan) * 100 + 0.5;  
6     }  
7     operator double(){ //conversion operator converts to  
8         return yuan + jf/100.0;  
9     }  
10    void display(){ cout<<(yuan + jf/100.0)<<endl; }  
11};  
12  
13  
14 int main(){  
15     RMB d1(2.0), d2(1.5), d3; //constructor converts to RMB  
16 }
```

10.5 CONVERSION OPERATORS

```
8     return yuan + jf/100.0;
9 }
10
11 void display(){ cout<<(yuan + jf/100.0)<<endl; }
12 };
13
14 int main(){
15     RMB d1(2.0), d2(1.5), d3; //constructor converts to RMB
16     // (explicit) convert to floating point number for + operation
17     d3 = RMB((double)d1 + (double)d2);
18     // (implicit) d1 and d2 do not overload +,
19     // but have conversion operator to convert to floating point number
20     d3 = d1 + d2;
21     d3.display();
22 }
```

10.6 ASSIGNMENT OPERATORS

10.6 ASSIGNMENT OPERATORS

- Classes always have a default assignment operator, which usually does not need to be overloaded

10.6 ASSIGNMENT OPERATORS

- Classes always have a default assignment operator, which usually does not need to be overloaded
- When class objects are copied with deep copy properties, you need to customize:

10.6 ASSIGNMENT OPERATORS

- Classes always have a default assignment operator, which usually does not need to be overloaded
- When class objects are copied with deep copy properties, you need to customize:
 - `ClassName(const ClassName &other)`

10.6 ASSIGNMENT OPERATORS

- Classes always have a default assignment operator, which usually does not need to be overloaded
- When class objects are copied with deep copy properties, you need to customize:
 - `ClassName(const ClassName &other)`
 - `ClassName& operator=(const ClassName &other)`

10.6 ASSIGNMENT OPERATORS

- Classes always have a default assignment operator, which usually does not need to be overloaded
- When class objects are copied with deep copy properties, you need to customize:
 - `ClassName(const ClassName &other)`
 - `ClassName& operator=(const ClassName &other)`
 - Constructor

10.6 ASSIGNMENT OPERATORS

- Classes always have a default assignment operator, which usually does not need to be overloaded
- When class objects are copied with deep copy properties, you need to customize:
 - `ClassName(const ClassName &other)`
 - `ClassName& operator=(const ClassName &other)`
 - Constructor
 - Destructor

10.6 ASSIGNMENT OPERATORS

- Classes always have a default assignment operator, which usually does not need to be overloaded
- When class objects are copied with deep copy properties, you need to customize:
 - `ClassName(const ClassName &other)`
 - `ClassName& operator=(const ClassName &other)`
 - Constructor
 - Destructor
- The first parameter of the assignment operator is usually an object, so it is always designed as a member function

10.6 ASSIGNMENT OPERATORS

Question: How many default functions does class A {};
have?

10.6 ASSIGNMENT OPERATORS

Question: How many default functions does class A {};
have?

- A() Default constructor

10.6 ASSIGNMENT OPERATORS

Question: How many default functions does class A {}; have?

- A() Default constructor
- A(const A&) Copy constructor

10.6 ASSIGNMENT OPERATORS

Question: How many default functions does class A {}; have?

- A() Default constructor
- A(const A&) Copy constructor
- ~A() Destructor

10.6 ASSIGNMENT OPERATORS

Question: How many default functions does class A {}; have?

- A() Default constructor
- A(const A&) Copy constructor
- ~A() Destructor
- A& operator=(const A&) Copy assignment operator

10.6 ASSIGNMENT OPERATORS

Question: How many default functions does class A {}; have?

- A() Default constructor
- A(const A&) Copy constructor
- ~A() Destructor
- A& operator=(const A&) Copy assignment operator
- A* operator&() Address-of operator

10.6 ASSIGNMENT OPERATORS

Question: How many default functions does class A {}; have?

- A() Default constructor
- A(const A&) Copy constructor
- ~A() Destructor
- A& operator=(const A&) Copy assignment operator
- A* operator&() Address-of operator
- const A* operator&() Const address-of operator

10.6 ASSIGNMENT OPERATORS

Question: How many default functions does class A {}; have?

- A() Default constructor
- A(const A&) Copy constructor
- ~A() Destructor
- A& operator=(const A&) Copy assignment operator
- A* operator&() Address-of operator
- const A* operator&() Const address-of operator
- A(A&&) Move constructor

10.6 ASSIGNMENT OPERATORS

Question: How many default functions does class A {}; have?

- A() Default constructor
- A(const A&) Copy constructor
- ~A() Destructor
- A& operator=(const A&) Copy assignment operator
- A* operator&() Address-of operator
- const A* operator&() Const address-of operator
- A(A&&) Move constructor
- A& operator=(A&&) Move assignment operator

10.6 ASSIGNMENT OPERATORS

Example: example/lec10/myvector

Example: example/lec10/smart_ptr

SUMMARY

SUMMARY

- Operators can be overloaded as member functions or as ordinary functions

SUMMARY

- Operators can be overloaded as member functions or as ordinary functions
- After operator overloading, associativity, precedence, etc. remain unchanged

SUMMARY

- Operators can be overloaded as member functions or as ordinary functions
- After operator overloading, associativity, precedence, etc. remain unchanged
- Most operations can be overloaded, but a few operators cannot be overloaded

SUMMARY

- Operators can be overloaded as member functions or as ordinary functions
- After operator overloading, associativity, precedence, etc. remain unchanged
- Most operations can be overloaded, but a few operators cannot be overloaded
- Inexistent operators cannot be overloaded

SUMMARY

- Operators can be overloaded as member functions or as ordinary functions
- After operator overloading, associativity, precedence, etc. remain unchanged
- Most operations can be overloaded, but a few operators cannot be overloaded
- Inexistent operators cannot be overloaded
- Good programming style: try not to overload operators unless they are conventional.