运算符重载

主讲：陈笑沙

目录

10.1 运算符的需要性

10.1 运算符的需要性

- C++ 中，我们不仅要使用基本数据类型，还要设计新的数据类型—类类型。

10.1 运算符的需要性

- C++ 中，我们不仅要使用基本数据类型，还要设计新的数据类型—类类型。
- 一般情况下，基本数据类型的运算都是用运算符来表达，这很直观，语义也简单。

10.1 运算符的需要性

- C++ 中，我们不仅要使用基本数据类型，还要设计新的数据类型—类类型。
- 一般情况下，基本数据类型的运算都是用运算符来表达，这很直观，语义也简单。
  - `a = b + c`

10.1 运算符的需要性

- C++ 中，我们不仅要使用基本数据类型，还要设计新的数据类型—类类型。
- 一般情况下，基本数据类型的运算都是用运算符来表达，这很直观，语义也简单。
  - a = b + c
  - 内部数据类型都预定义了运算符(操作符)

10.1 运算符的需要性

10.1 运算符的需要性

- C++ 擅长自定义数据类型,以成员函数的方式定义其操作

10.1 运算符的需要性

- C++ 擅长自定义数据类型,以成员函数的方式定义其操作
  - 矩阵 `a + b` 实现调用为 `a.add(b)`

10.1 运算符的需要性

- C++ 擅长自定义数据类型,以成员函数的方式定义其操作
  - 矩阵 `a + b` 实现调用为 `a.add(b)`
  - 或者 `Matrix::add(a, b)`

10.1 运算符的需要性

- C++ 擅长自定义数据类型,以成员函数的方式定义其操作
  - 矩阵 `a + b` 实现调用为 `a.add(b)`
  - 或者 `Matrix::add(a, b)`
- C++ 能在自定义类中定义操作符

10.1 运算符的需要性

- C++ 擅长自定义数据类型,以成员函数的方式定义其操作
  - 矩阵 `a + b` 实现调用为 `a.add(b)`
  - 或者 `Matrix::add(a, b)`
- C++ 能在自定义类中定义操作符
  - 为矩阵定义 `+` 运算符

10.1 运算符的需要性

10.1 运算符的需要性

- 如果直接将运算符作用在类类型之上，情况又如何呢?

10.1 运算符的需要性

- 如果直接将运算符作用在类类型之上，情况又如何呢?
  - `Complex ret, c1,c2; ret=c1+c2;`

10.1 运算符的需要性

- 如果直接将运算符作用在类类型之上，情况又如何呢?
  - `Complex ret, c1,c2; ret=c1+c2;`
- 编译器将不能识别运算符的语义。

10.1 运算符的需要性

- 如果直接将运算符作用在类类型之上，情况又如何呢?
  - `Complex ret, c1,c2; ret=c1+c2;`
- 编译器将不能识别运算符的语义。
- 需要一种机制来重新定义运算符作用在类类型上的含义。

10.1 运算符的需要性

- 如果直接将运算符作用在类类型之上，情况又如何呢?
  - `Complex ret, c1,c2; ret=c1+c2;`
- 编译器将不能识别运算符的语义。
- 需要一种机制来重新定义运算符作用在类类型上的含义。
- 这种机制就是运算符重载。

10.2 如何重载运算符

```
 1 #include <sstream>
 2 #include <iostream>
 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;
```

10.2 如何重载运算符

```
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    std::cout << c3.to_string() << std::endl;
      std::ostringstream oss;
```

10.2 如何重载运算符

```
#include <sstream>
#include <iostream>

class Complex {
public:
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  static Complex add(const Complex &c1, const Complex &c2) {
    Complex r;
    r.re = c1.re + c2.re;
    r.im = c1.im + c2.im;
    return r;
  }

  std::string to_string() const {
    std::ostringstream oss;
```

10.2 如何重载运算符

```
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  private:
```

10.2 如何重载运算符

```
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 如何重载运算符

示例解读：example/lec10/complex

10.2 如何重载运算符

10.2 如何重载运算符

- 实际上还可以：

10.2 如何重载运算符

- 实际上还可以：
  - 将 `Complex operator+(Complex& c1, Complex &c2)` 声明为类成员函数。

10.2 如何重载运算符

- 实际上还可以：
  - 将 `Complex operator+(Complex& c1, Complex &c2)` 声明为类成员函数。
  - `c2 = c1 + 27` 相当于 `c2 = c1.operator+(Complex{27})`

10.2 如何重载运算符

- 实际上还可以：
  - 将 `Complex operator+(Complex& c1, Complex &c2)` 声明为类成员函数。
  - `c2 = c1 + 27` 相当于 `c2 = c1.operator+(Complex{27})`
  - 此时，`c2 = 27 + c1` 会报错

10.2 如何重载运算符

也可以采用友元：

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

10.3 返回值与返回引用

10.3 返回值与返回引用

- 如果想输出复数对象，该怎么办?

10.3 返回值与返回引用

- 如果想输出复数对象，该怎么办?
  - `cout << c.to_string() << endl`

10.3 返回值与返回引用

- 如果想输出复数对象，该怎么办?
  - `cout << c.to_string() << endl`
- 如果想要直接 `cout << c << endl` 呢?

10.3 返回值与返回引用

```
 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 &
11
12   friend ostream &operator<<(ostream &out, const Complex &c)
13
14 private:
15   double re, im;
```

10.3 返回值与返回引用

```
 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 &
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) {
   double re, im;
```

10.3 返回值与返回引用

```
 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 &
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) {
```

10.3 返回值与返回引用

```
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      if (c.re ≠ 0)
        double re, im;
```

10.3 返回值与返回引用

```
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 }
35
   double re, im;
```

10.3 返回值与返回引用

```
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  }
    double re, im;
```

10.3 返回值与返回引用

思考：为何 `std::ostream` 需要返回引用？为何函数参数不能加 `const`？

10.4 重载增量运算符

10.4 重载增量运算符

- 类对象要实现自增，自减操作，也要进行运算符重载。

10.4 重载增量运算符

- 类对象要实现自增，自减操作，也要进行运算符重载。
- 那么，如何区别前置和后置呢?

10.4 重载增量运算符

- 类对象要实现自增，自减操作，也要进行运算符重载。
- 那么，如何区别前置和后置呢?
- 自增自减的函数原型是什么?

10.4 重载增量运算符

10.4 重载增量运算符

- 增量运算符分前增量与后增量,前减量与后减量

10.4 重载增量运算符

- 增量运算符分前增量与后增量,前减量与后减量
  - `int a=3;`

10.4 重载增量运算符

- 增量运算符分前增量与后增量,前减量与后减量
  - `int a=3;`
  - `a++;  //后增量`

10.4 重载增量运算符

- 增量运算符分前增量与后增量,前减量与后减量
  - `int a=3;`
  - `a++;  //后增量`
  - `++a;  //前增量`

10.4 重载增量运算符

- 增量运算符分前增量与后增量,前减量与后减量
  - `int a=3;`
  - `a++;  //后增量`
  - `++a;  //前增量`
  - `--a;  //前减量`

10.4 重载增量运算符

- 增量运算符分前增量与后增量,前减量与后减量
  - `int a=3;`
  - `a++;  //后增量`
  - `++a;  //前增量`
  - `--a;  //前减量`
  - `a--;  //后减量`

10.4 重载增量运算符

- 增量运算符分前增量与后增量,前减量与后减量
  - `int a=3;`
  - `a++;  //后增量`
  - `++a;  //前增量`
  - `--a;  //前减量`
  - `a--;  //后减量`
- 施行前增量运算成为左值表达式

10.4 重载增量运算符

- 增量运算符分前增量与后增量,前减量与后减量
  - `int a=3;`
  - `a++;  //后增量`
  - `++a;  //前增量`
  - `--a;  //前减量`
  - `a--;  //后减量`
- 施行前增量运算成为左值表达式
- 施行后增量运算成为右值表达式

10.4 重载增量运算符

- 增量运算符分前增量与后增量,前减量与后减量
  - `int a=3;`
  - `a++;  //后增量`
  - `++a;  //前增量`
  - `--a;  //前减量`
  - `a--;  //后减量`
- 施行前增量运算成为左值表达式
- 施行后增量运算成为右值表达式
- 重载前++与后++,都只有一个操作数,故对应一个参数。

10.4 重载增量运算符

10.4 重载增量运算符

- 运算符都是 `operator++`

10.4 重载增量运算符

- 运算符都是 `operator++`
- 重载前++时返回引用,重载后++时返回值

10.4 重载增量运算符

- 运算符都是 operator++
- 重载前++时返回引用,重载后++时返回值
- C++对前++和后++的重载,区分为：后++增设一整型参数

10.4 重载增量运算符

- 运算符都是 operator++
- 重载前++时返回引用,重载后++时返回值
- C++对前++和后++的重载,区分为：后++增设一整型参数
  - `T &operator++(T& a); // 前++`

10.4 重载增量运算符

- 运算符都是 operator++
- 重载前++时返回引用,重载后++时返回值
- C++对前++和后++的重载,区分为：后++增设一整型参数
  - `T &operator++(T& a); // 前++`
  - `T &operator++(T& a, int); // 后++`

10.4 重载增量运算符

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

class Increase {
  int value;

public:
  Increase(int x) : value(x) {}
  Increase &operator++() { // 前增量(无参数)
    value++;               // 先增量
    return *this;          // 返回原对象
  }

  Increase operator++(int) {
    // 后增量(仅一标记参数int)
```

10.4 重载增量运算符

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

class Increase {
  int value;

public:
  Increase(int x) : value(x) {}
  Increase &operator++() { // 前增量(无参数)
    value++;               // 先增量
    return *this;          // 返回原对象
  }

  Increase operator++(int) {
    // 后增量(仅一标记参数int)
```

10.4 重载增量运算符

```
 4 class Increase {
 5   int value;
 6
 7 public:
 8   Increase(int x) : value(x) {}
 9   Increase &operator++() { // 前增量(无参数)
10     value++;               // 先增量
11     return *this;          // 返回原对象
12   }
13
14   Increase operator++(int) {
15     // 后增量(仅一标记参数int)
16     Increase temp(value);
17     // 构造临时对象存放原对象值
18     // 后增量运算，然后原对象值改变
```

10.4 重载增量运算符

```
10      value++;                    // 先增量
11      return *this;               // 返回原对象
12    }
13
14    Increase operator++(int) {
15      // 后增量(仅一标记参数int)
16      Increase temp(value);
17      // 构造临时对象存放原对象值
18      value++;        // 原有对象值改变
19      return temp; // 返回原对象值
20    }
21
22    void display() { cout << "the value is " << value << endl;
23  };
24
```

10.4 重载增量运算符

```
15      // 后增量(仅一标记参数int)
16      Increase temp(value);
17      // 构造临时对象存放原对象值
18      value++;       // 原有对象值改变
19      return temp; // 返回原对象值
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();
      // 后增量(仅一标记参数int)
```

10.4 重载增量运算符

```
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 重载增量运算符

this 指针的用处

10.4 重载增量运算符

this 指针的用处

- 一个对象的 this 指针并不是对象本身的一部分，不会影响 sizeof(对象) 的结果。

10.4 重载增量运算符

this 指针的用处

- 一个对象的 this 指针并不是对象本身的一部分，不会影响 sizeof(对象) 的结果。
- this 作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员的时候，编译器会自动将对象本身的地址作为一个隐含参数传递给函数。

10.4 重载增量运算符

this 指针的用处

- 一个对象的 this 指针并不是对象本身的一部分，不会影响 sizeof(对象) 的结果。
- this 作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员的时候，编译器会自动将对象本身的地址作为一个隐含参数传递给函数。
- 也就是说，即使你没有写上 this 指针，编译器在编译的时候也是加上 this 的，它作为非静态成员函数的隐含形参，对各成员的访问均通过 this 进行。

10.4 重载增量运算符

this 指针的使用

10.4 重载增量运算符

this 指针的使用

- 一种情况就是，在类的非静态成员函数中返回类对象本身的时候，直接使用 `return *this;`

10.4 重载增量运算符

this 指针的使用

- 一种情况就是，在类的非静态成员函数中返回类对象本身的时候，直接使用 `return *this;`
- 另外一种情况是当参数与成员变量名相同时，如 `this→n = n`

10.4 重载增量运算符

普通函数形式

```
#include<iostream>
using namespace std;
class Increase{
  int value;
public:
  Increase(int x) : value(x) {}
  friend Increase &operator++(Increase& a);
  friend Increase operator++(Increase &a, int);
  void display() { cout<< "the value is " << value << endl;
};

Increase& operator++(Increase& a){
  a.value++;
  return a;
}
```

10.4 重载增量运算符

普通函数形式

```
#include<iostream>
using namespace std;
class Increase{
  int value;
public:
  Increase(int x) : value(x) {}
  friend Increase &operator++(Increase& a);
  friend Increase operator++(Increase &a, int);
  void display() { cout<< "the value is " << value << endl;
};

Increase& operator++(Increase& a){
  a.value++;
  return a;
}
```

10.4 重载增量运算符

普通函数形式

```
 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 重载增量运算符

普通函数形式

```
 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 转换运算符

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

class RMB {
  unsigned int yuan, jf;  //元角分
public:
  RMB(double d=0) : yuan(d) , jf(int(d*100+0.5)%100){}

  RMB(int y, int f):yuan(y), jf(f) {}
  friend RMB operator+(const RMB&, const RMB&);
  friend RMB& operator++(RMB&);
  void display(){ cout<<(yuan + jf / 100.0)<<endl; }
};

RMB operator+(const RMB& s1, const RMB& s2){
```

10.5 转换运算符

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

class RMB {
  unsigned int yuan, jf;  //元角分
public:
  RMB(double d=0) : yuan(d) , jf(int(d*100+0.5)%100){}

  RMB(int y, int f):yuan(y), jf(f) {}
  friend RMB operator+(const RMB&, const RMB&);
  friend RMB& operator++(RMB&);
  void display(){ cout<<(yuan + jf / 100.0)<<endl; }
};

RMB operator+(const RMB& s1, const RMB& s2){
RMB operator++(const RMB& s1,int const RMB& s2){
```

10.5 转换运算符

```
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    yuan += ++s.jf/100;
23    s.jf %= 100;
24    return s;
```

10.5 转换运算符

```
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    yuan += ++s.jf/100;
23    s.jf %= 100;
24    return s;
25  }
26
27  int main(){
28    RMB w(12.567);   //将浮点数转为RMB对象(浮点类转为RMB类)
29    ++w;
30    w.display();
```

10.5 转换运算符

```
17    unsigned int yuan = s1.yuan + s2.yuan + x/100;
18    return RMB(yuan, x % 100);
19  }
20
21  RMB& operator++(RMB& s){
22    yuan += ++s.jf/100;
23    s.jf %= 100;
24    return s;
25  }
26
27  int main(){
28    RMB w(12.567);   //将浮点数转为RMB对象(浮点类转为RMB类)
29    ++w;
30    w.display();
31  }
```

10.5 转换运算符

```
1  class RMB{
2    unsigned int yuan, jf;   //元角分
3  public:
4    RMB(double value = 0.0) : yuan(value) {
5      jf = (value - yuan) * 100 + 0.5;
6    }
7    operator double(){   //转换运算符行使转出去
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; //构造函数转成RMB
```

10.5 转换运算符

```
class RMB{
  unsigned int yuan, jf;  //元角分
public:
  RMB(double value = 0.0) : yuan(value) {
    jf = (value - yuan) * 100 + 0.5;
  }
  operator double(){  //转换运算符行使转出去
    return yuan + jf/100.0;
  }

  void display(){ cout<<(yuan + jf/100.0)<<endl; }
};

int main(){
  RMB d1(2.0), d2(1.5), d3; //构造函数转成RMB
```

10.5 转换运算符

```
class RMB{
  unsigned int yuan, jf;  //元角分
public:
  RMB(double value = 0.0) : yuan(value) {
    jf = (value - yuan) * 100 + 0.5;
  }
  operator double(){  //转换运算符行使转出去
    return yuan + jf/100.0;
  }

  void display(){ cout<<(yuan + jf/100.0)<<endl; }
};

int main(){
  RMB d1(2.0), d2(1.5), d3; //构造函数转成RMB
```

10.5 转换运算符

```
 7    operator double(){   //转换运算符打使转出去
 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; //构造函数转成RMB
16    //(显式)转成浮点数做+运算
17    d3 = RMB((double)d1 + (double)d2);
18    //(隐式)d1和d2没有重载+,却有转换运算符而转成浮点数
19    d3 = d1 + d2;
20    d3.display();
21  }
```

10.6 赋值运算符

10.6 赋值运算符

- 类中总是有默认赋值运算符，一般情况下无须重载赋值运算符

10.6 赋值运算符

- 类中总是有默认赋值运算符，一般情况下无须重载赋值运算符
- 当类对象拷贝为深拷贝性质时，需要自定义：

10.6 赋值运算符

- 类中总是有默认赋值运算符，一般情况下无须重载赋值运算符
- 当类对象拷贝为深拷贝性质时，需要自定义：
  - `ClassName(const ClassName &other)`

10.6 赋值运算符

- 类中总是有默认赋值运算符，一般情况下无须重载赋值运算符
- 当类对象拷贝为深拷贝性质时，需要自定义：
  - `ClassName(const ClassName &other)`
  - `ClassName& operator=(const ClassName &other)`

10.6 赋值运算符

- 类中总是有默认赋值运算符，一般情况下无须重载赋值运算符
- 当类对象拷贝为深拷贝性质时，需要自定义：
  - `ClassName(const ClassName &other)`
  - `ClassName& operator=(const ClassName &other)`
  - 构造函数

10.6 赋值运算符

- 类中总是有默认赋值运算符，一般情况下无须重载赋值运算符
- 当类对象拷贝为深拷贝性质时，需要自定义：
  - `ClassName(const ClassName &other)`
  - `ClassName& operator=(const ClassName &other)`
  - 构造函数
  - 析构函数

10.6 赋值运算符

- 类中总是有默认赋值运算符，一般情况下无须重载赋值运算符
- 当类对象拷贝为深拷贝性质时，需要自定义：
  - `ClassName(const ClassName &other)`
  - `ClassName& operator=(const ClassName &other)`
  - 构造函数
  - 析构函数
- 赋值运算符第一参数一般为对象,所以其总是设计为成员函数

10.6 赋值运算符

问题：`class A{};`有几个默认函数?

10.6 赋值运算符

问题：`class A{};`有几个默认函数?

- `A()`默认构造函数

10.6 赋值运算符

问题：`class A{};`有几个默认函数?

- `A()` 默认构造函数
- `A(const A&)` 拷贝构造函数

10.6 赋值运算符

问题：`class A{};` 有几个默认函数?

- `A()` 默认构造函数
- `A(const A&)` 拷贝构造函数
- `~A()` 析构函数

10.6 赋值运算符

问题：`class A{};` 有几个默认函数?

- `A()` 默认构造函数
- `A(const A&)` 拷贝构造函数
- `~A()` 析构函数
- `A& operator=(const A&)` 拷贝赋值运算符

10.6 赋值运算符

问题：class A{};有几个默认函数?

- A() 默认构造函数
- A(const A&) 拷贝构造函数
- ~A() 析构函数
- A& operator=(const A&) 拷贝赋值运算符
- A* operator&() 取址运算符

10.6 赋值运算符

问题：`class A{};` 有几个默认函数?

- `A()` 默认构造函数
- `A(const A&)` 拷贝构造函数
- `~A()` 析构函数
- `A& operator=(const A&)` 拷贝赋值运算符
- `A* operator&()` 取址运算符
- `const A* operator&()` const取址运算符

10.6 赋值运算符

问题：`class A{};` 有几个默认函数?

- `A()` 默认构造函数
- `A(const A&)` 拷贝构造函数
- `~A()` 析构函数
- `A& operator=(const A&)` 拷贝赋值运算符
- `A* operator&()` 取址运算符
- `const A* operator&()` const取址运算符
- `A(A&&)` 移动构造函数

10.6 赋值运算符

问题：`class A{};` 有几个默认函数?

- `A()` 默认构造函数
- `A(const A&)` 拷贝构造函数
- `~A()` 析构函数
- `A& operator=(const A&)` 拷贝赋值运算符
- `A* operator&()` 取址运算符
- `const A* operator&()` const取址运算符
- `A(A&&)` 移动构造函数
- `A& operator=(A&&)` 移动赋值运算符

10.6 赋值运算符

示例：example/lec10/myvector

总结

总结

- 运算符可以重载为成员函数，也可以重载为普通函数

总结

- 运算符可以重载为成员函数，也可以重载为普通函数
- 运算符重载后，结合性、优先级等不变

总结

- 运算符可以重载为成员函数，也可以重载为普通函数
- 运算符重载后，结合性、优先级等不变
- 大部分运算可以重载，但是少量运算符无法重载

总结

- 运算符可以重载为成员函数，也可以重载为普通函数
- 运算符重载后，结合性、优先级等不变
- 大部分运算可以重载，但是少量运算符无法重载
- 不存在的运算符无法重载

总结

- 运算符可以重载为成员函数，也可以重载为普通函数
- 运算符重载后，结合性、优先级等不变
- 大部分运算可以重载，但是少量运算符无法重载
- 不存在的运算符无法重载
- 好的编程风格：尽量不要重载运算符，除非是约定俗成的。