C++ Programming

Chapter 10 Operator Overloading

运算符重载

Zheng Guibin
（郑贵滨）

哈爾濱工業大學
Harbin Institute of Technology

Operator overloading...

- **运算符重载Operator Overloading**
  - 当定义一个新的类时，可以**重新定义/重载**已经存在的运算符。
  - 运算符重载—— 运算符函数

    在类中，使用运算符定义的**特殊成员函数。**

Operator overloading...

```
6  class time24 // A simple 24-hour time class.
7  {
8       public:
9          time24( int h = 0, int m = 0, int s = 0 ) ;
10         void set_time( int h, int m, int s ) ;
11         void get_time( int& h, int& m, int& s ) const ;
12         time24 operator+( int secs ) const ;
13      private:
14         int hours ; // 0 to 23
15         int minutes ; // 0 to 59
16         Int seconds ; // 0 to 59
17 };
```

10.2 Overloading the addition operator +

```
32 // Overloaded + operator.
33 time24  time24::operator+( int secs ) const
34 {
35       // Add secs to class member seconds and calculate the new
time.
36        time24 temp ;
37        temp.seconds = seconds + secs ;
38        temp.minutes = minutes + temp.seconds / 60 ;
39        temp.seconds %= 60 ;
40        temp.hours = hours + temp.minutes / 60 ;
41       temp.minutes %= 60 ;
42       temp.hours %= 24 ;
43       return temp ; // Return the new time.
44 }
```

10.2 Overloading the addition operator +

```
45 main()
46 {
47        int h, m, s ;
48         time24 t1( 23, 59, 57 ) ; // t1 represents 23:59:57
49         time24 t2 ;
50         t2 = t1 + 4 ; // t2 should now be 0:0:1
51         t2.get_time ( h, m, s ) ;
52        cout << "Time t2 is " << h << ":" << m << ":" << s << endl ;
53 }
```

它可以像其他函数那样被调用; 第50行也可以写成:

```
t2 = t1.operator+( 4 ) ;
```

10.2 Overloading the addition operator +

Program Example P10A+B+C

```
1  // Program Example P10C
2  // Demonstration of an overloaded + operator.
3  #include <iostream>
4  using namespace std ;
5  class time24 // A simple 24-hour time class.
6  {
7       public:
8          time24( int h = 0, int m = 0, int s = 0 ) ;
9          void set_time( int h, int m, int s ) ;
10         void get_time( int& h, int& m, int& s ) const ;
11         time24 operator+( int secs ) const ;//重载运算符+的函数原型
12         time24 operator+( const time24& t ) const ;
13         private:
14         int hours ; // 0 to 23
15         int minutes ; // 0 to 59
16         Int seconds ; // 0 to 59
17 }
```

10.2 Overloading the addition operator +

```
32 // Overloaded + operator.
33 time24 time24::operator+( int secs ) const
34 {//重载+运算符：将time24对象和一个整型的秒数相加并返回结果
35       // Add secs to class member seconds and calculate the new time.
36        time24 temp ;
37        temp.seconds = seconds + secs ;
38        temp.minutes = minutes + temp.seconds / 60 ;
39        temp.seconds %= 60 ;
40        temp.hours = hours + temp.minutes / 60 ;
41       temp.minutes %= 60 ;
42       temp.hours %= 24 ;
43       return temp ; // Return the new time.
44 }
```

```
45 // Overloaded + operator.
46 time24 time24::operator+( const time24& t ) const
47 {
48       // Add total seconds in t to seconds and calculate the new time.
49       time24 temp ;
50       int secs = t.hours * 3600 + t.minutes * 60 + t.seconds ;
51       temp.seconds = seconds + secs ;
52       temp.minutes = minutes + temp.seconds / 60 ;
53       temp.seconds %= 60 ;
54       temp.hours = hours + temp.minutes / 60 ;
55       temp.minutes %= 60 ;
56       temp.hours %= 24 ;
57       return temp ; // Return the new time.
58 }
```

10.2 Overloading the addition operator +

```
// Non-member overloaded + operator.
59 time24 operator+( int secs, const time24& t )
60 {
61       // Add secs to t to calculate the new time.
62        time24 temp ;
63       temp = t + secs ; // Uses the member function operator+( int ).
64       return temp ; // Return the new time.
65 }
```

10.2 Overloading the addition operator +

//Program Example P10B

```
66 main()
67 {
68        int h, m, s ;
69        time24 start_time( 23, 0, 0 ) ;
70         time24 elapsed_time( 1, 2, 3 ) ;
71         time24 finish_time ;
72        finish_time = start_time + elapsed_time ;
73         finish_time.get_time( h, m, s ) ;
74        cout << "Finish Time is "
75         << h << ":" << m << ":" << s << endl ;
```

finish_time = start_time.operator+(elapsed_time) ;

10.2 Overloading the addition operator +

Program Example P10C

```
76        time24 t1( 23, 59, 57 ) ; // t1 represents 23:59:57
77        time24 t2 ;
78        t2 = t1 + 4 ;
79        t2.get_time ( h, m, s ) ;
80        cout << "Time t2 is " << h << ":" << m << ":" << s << endl ;
81        t2 = 4 + t1 ;
82        t2.get_time ( h, m, s ) ;
83        cout << "Time t2 is " << h << ":" << m << ":" << s << endl ;
84 }
```

t2 = t1.operator+(4) ;

t2 = operator+(4, t1)

t2 = operator+(t1, 4) ;//?

10.4.1 Overloading prefix and postfix forms of ++

- C++编译器如何区分重载的后置++ 前置++
  - 若d为自定义类 MyClass的对象

自增表达式	编译器自动生成的成员函数调用	对应的函数原型
++d	d.operator++()	MyClass &operator++();
d++	d.operator++(0)	MyClass operator++(int);

自增表达式	编译器自动生成的非成员函数调用	对应的函数原型
++d	operator++(d)	friend MyClass &operator++(MyClass &);
d++	operator++(d,0)	friend MyClass operator++(MyClass &, int);

10.5 Overloading relational operators

```
1 // Program Example P10F，Overloading relational operator ==
...
6 class time24 // A simple 24-hour time class.
7 {
8 public:
9     time24( int h = 0, int m = 0, int s = 0 ) ;
10    void set_time( int h, int m, int s ) ;
11    void get_time( int& h, int& m, int& s ) const ;
12    time24 operator+( int secs ) const ;
13    time24 operator+( const time24& t ) const ;
14    time24& operator++() ; // prefix.
15    time24 operator++( int ) ; // postfix.
16    bool operator == ( const time24& t ) const ;
17 private:
18    int hours ; // 0 to 23
19    int minutes ; // 0 to 59
20    int seconds ; // 0 to 59
21 } ;
```

10.5 Overloading relational operators

```
87 // Overloaded equality operator ==.
88 bool time24::operator == ( const time24& t ) const
89 {
90   if ( hours == t.hours &&
91        minutes == t.minutes &&
92        seconds == t.seconds )
93        return true ;
94   else
95        return false ;
96 }
```

```
107 main()
108 {
109   int h, m, s ;     time24 t1( 23, 59, 57 ) ;   time24 t2 ;
...
113   t2 = ++t1 ; // t1 and t2 should be equal.
114   t1.get_time ( h, m, s ) ;
115   cout << "t1 is "
116        << h << ":" << m << ":" << s ;
117   t2.get_time ( h, m, s ) ;
118   cout << ", t2 is "
119        << h << ":" << m << ":" << s << endl ;
120
121   if ( t1 == t2 ) // Test equality operator ==.
122     cout << "t1 and t2 are equal. Prefix ++ is working."
123          << endl ;
124   else
125     cout << "t1 and t2 are not equal. Prefix ++ is not working."
126          << endl ;
127 }
```

10.6 Overloading << and >>

- 重载运算符<< 和 >>的实现与非成员函数一样.
- cout 是类ostream的对象，cin是类istream的对象。
- 这些对象和类实在程序第3行包含的头文件 iostream 中定义的。
- 任何使用输出流的函数都会改变cout，因此向函数传递的必须是输出流的引用，而不能传值。

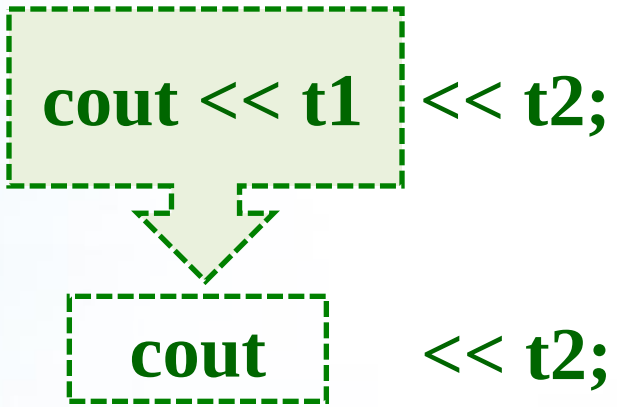
10.6 Overloading << and >>

```
108 // Non-member overloaded << operator.
109 ostream& operator<<( ostream& os, const time24& t )
110 {
111   // Format time24 object, precede single digits with a 0.
112   int h, m, s ;
113   t.get_time( h, m, s ) ;
114   os << setfill( '0' )
115      << setw( 2 ) << h << ":"
116      << setw( 2 ) << m << ":"
117      << setw( 2 )<< s << endl ;
118   return os ;
119 }
```

10.6 Overloading << and >>

- 第109行函数的返回类型是由第118行返回的输出流对象的引用。

```
109 ostream& operator<<( ostream& os, const time24& t )
```



- 语句的第一部分（cout<<t1）输出t1的值，并返回一个cout的引用，使得语句的后面部分（<<t2）可以继续使用cout。
- istream& operator>>(istream& is, time24& t);

10.6 Overloading << and >>

```
121 // Non-member overloaded >> operator.
122 istream& operator>>( istream& is, time24& t )
123 {
124   //Input a time24 object data in the format h:m:s.
125   int h, m, s ;
126   do
127   {
128     is >> h ;
129   }
130   while ( h < 0 || h > 23 ) ;
131   //Ignore the separator.
132   is.ignore( 1 ) ;
133   do
134   {
135     is >> m ;
136   }
137  while ( m < 0 || m > 60 ) ;
```

```
138 // Ignore the separator.
139   is.ignore( 1 ) ;
140   do
141   {
142     is >> s ;
143   }
144   while ( s < 0 || s > 60 ) ;
145   t.set_time ( h, m, s ) ;
146   return is ;
147 }
```

10.6 Overloading << and >>

```
149 main()
150 {
151   time24 t1( 1, 2, 3 ) ;
152   time24 t2 ( 10, 10, 10 ) ;
153
154   cout << t1 << t2 ;
155
156   time24 t3 ;
157   cin >> t3 ;
158   cout << t3 ;
159 }
```

```
01:02:03
10:10:10
4:5:6
04:05:06
```

10.7 Conversion operators（转换运算符）

- 转换运算符函数用于将一个类对象转换成内置数据类型或其他类对象。转换运算符成员函数与它要转换成的数据类型具有相同的名字。
- 转换运算符函数与其他重载运算符函数有两点不同：
  - 第一，转换运算符函数无需实参。
  - 第二，转换运算符函数没有返回类型，甚至void也不行。可以根据转换运算符函数的名字推出函数的返回类型，例如，如果要将一个time24对象转换为int类型，则在类中定义该转换运算符函数的名字为operator int。

```
1 // Program Example P10H
2 // Demonstration of a class conversion operator.
…
7 class time24 // A simple 24 hour time class.
8 {
9    public:
10   time24( int h = 0, int m = 0, int s = 0 ) ;
11   void set_time( int h, int m, int s ) ;
12   void get_time( int& h, int& m, int& s ) const ;
13   time24 operator+( int secs ) const ;
14   time24 operator+( const time24& t ) const ;
15   time24& operator++() ; // prefix.
16   time24 operator++( int ) ; // postfix.
17   bool operator == ( const time24& t ) const ;
18   operator int() ;
19 private:
20   int hours ; // 0 to 23
21   int minutes ; // 0 to 59
22   int seconds ; // 0 to 59
23 } ;
```

10.7 Conversion operators（转换运算符）

```
101 time24::operator int()
102 {
103   int no_of_seconds = hours * 3600 + minutes * 60 + seconds ;
104   return no_of_seconds ;
105 }
```

```
151 main()
152 {
153   time24 t( 1, 2, 3 ) ;
154   int s ;
155
156   s = t ; // Conversion from a time24 data type to an int data type.
157   cout << "Time = " << t
158        << "Equivalent number of seconds = " << s << endl ;
159 }
```

```
Time = 01:02:03
Equivalent number of seconds = 3723
```

10.8 Use of friend functions（使用友元函数）

- 使用类的成员函数可以访问类的私有成员的值。
  - time24类的私有成员hours,minutes和seconds的值可以使用类成员函数get_time()来读取，使用set_time()来赋值。
  - 在一个非成员函数中，要想直接访问类的私有数据成员，该函数必须声明为这个类的*友元函数（friend of the class）*。
- 为了将非成员函数operator>>和operator<<声明为time24类的友元函数，在类中应加入下列声明：

  friend 返回类型 operator 运算符(形参表)
- 这些声明可以放在类中的任何地方，但通常将其放在public部分。

10.8 Use of friend functions（使用友元函数）

- friend ostream& operator<<(ostream& os, const time24& t);
  friend istream& operator>>(istream& is, const time24& t);

```
108 // Non-member overloaded << operator.
109 ostream& operator<<( ostream& os, const time24& t )
110 {
111   //
112   in
113   t.g
114   os
115
116
117
118   re
119 }
```

```
108 // Non-member overloaded << operator.
109 ostream& operator<<( ostream& os, const time24& t )
110 {// 声明为time24的友元函数后
111   // Format time24 object, precede single digits with a 0.
112   //int h, m, s ;
113   //t.get_time( h, m, s ) ;
114   os << setfill( '0' )
115      << setw( 2 ) << t.hours << ":"
116      << setw( 2 ) << t.minutes << ":"
117      << setw( 2 )<< t.seconds << endl ;
118   return os ;
119 }
```

10.8 Use of friend functions（使用友元函数）

- Friend functions override a basic principle of object-oriented programming - that of data hiding.
- 友元函数破坏了面向对象程序的基本原则：数据隐藏。
- Friends of a class have access to all the private data of a class and their use should be minimised where possible.
- 类的友元可以访问类的所有私有数据，慎重使用！

10.9 Overloading the assignment operator =
（重载赋值运算符=）

- **10.9.1 A class with a pointer data member**

  含有指针数据成员的类

- **10.9.2 Assigning one object to another**

  对象之间的赋值

10.9.1 A class with a pointer data member
含有指针数据成员的类

- 考虑一个记录银行客户交易的类
- 为简化问题，类的私有数据成员只有三个：银行账号、交易的次数和每笔交易额。
- 每个银行账户都有一个不同的交易次数，交易的最大次数均设置为100。
- 每个transactions对象可以存储100个浮点型交易额。

固定数值：100、200 、 ...？

- 小的数值—— 对于大多数客户不够用
- 大的数值—— 大多数情况下会浪费内存空间
- 更好的方法是使用动态内存分配为每个账户分配恰好满足其大小需求的存储空间。这需要一个含有指针数据成员的类。

10.9.1 A class with a pointer data member

```
class transactions
{
public:
  transactions () ; // Constructor.
  ...
private:
  int account_number ;
  int number_of_transactions ;
  float transaction_amounts[100] ;
} ;
```

```
1 // Program Example P10I
2 // Demonstration of a class with a pointer data member.
3 #include <iostream>
4 using namespace std ;
5
6 class transactions // A class containing a pointer data member.
7 {
8 public:
9     transactions( int ac_num, int num_transactions,
10    const float transaction_values[] ) ;
11    // Purpose: Constructor.
12    ~transactions() ;
13    // Purpose: Destructor.
14    void display() const ;
15    // Purpose: Display transactions.
16 private:
17    int account_number ;
18    int number_of_transactions ;
19    float *transaction_amounts ; // Pointer to array of transactions.
20 } ;
```

10.9.1 A class with a pointer data member

```
22 // Constructor.
23 transactions::transactions( int ac_num, int num_transactions,
24                                const float transaction_values[] )
25 {
26   account_number = ac_num ;
27   number_of_transactions = num_transactions ;
28   // Allocate storage for the transactions.
29   transaction_amounts = new float[number_of_transactions] ;
30   for ( int i = 0 ; i < number_of_transactions ; i++ )
31     transaction_amounts[i] = transaction_values[i] ;
32 }
```

10.9.1 A class with a pointer data member

```
34 // Destructor.
35 transactions::~transactions()
36 {
37   delete[] transaction_amounts ;
38 }
39
```

```
40 void transactions::display() const
41 {
42   cout << " Account Number " << account_number << "
Transactions: ";
43   for ( int i = 0 ; i < number_of_transactions ; i++ )
44     cout << transaction_amounts[i] << ' ' ;
45   cout << endl ;
46 }
47
48 main()
49 {
50   // Construct a transactions object
51   // with account number 1 and 5 transactions.
52   float trans_amounts1[] = { 25.67, 6.12, 19.86, 23.41, 1.21 } ;
53   transactions trans_object1( 1, 5, trans_amounts1 ) ;
54   trans_object1.display() ;
55 }
```

10.9.2 Assigning one object to another (对象之间的赋值)

- 当两个相同类型的对象之间赋值时，默认情形是成员逐项赋值。
- 例如, t1和t2是time24对象

  t1 = t2 ;
- 上面的赋值语句可以一个接一个地复制每个成员：

  t1.hours = t2.hours;

  t1.minutes = t2.minutes;

  t1.seconds = t2.seconds;

- 对于大多数的类而言，默认的赋值运算符都能正确赋值
- 当类为它的一个或多个数据成员动态分配内存后，会有问题。

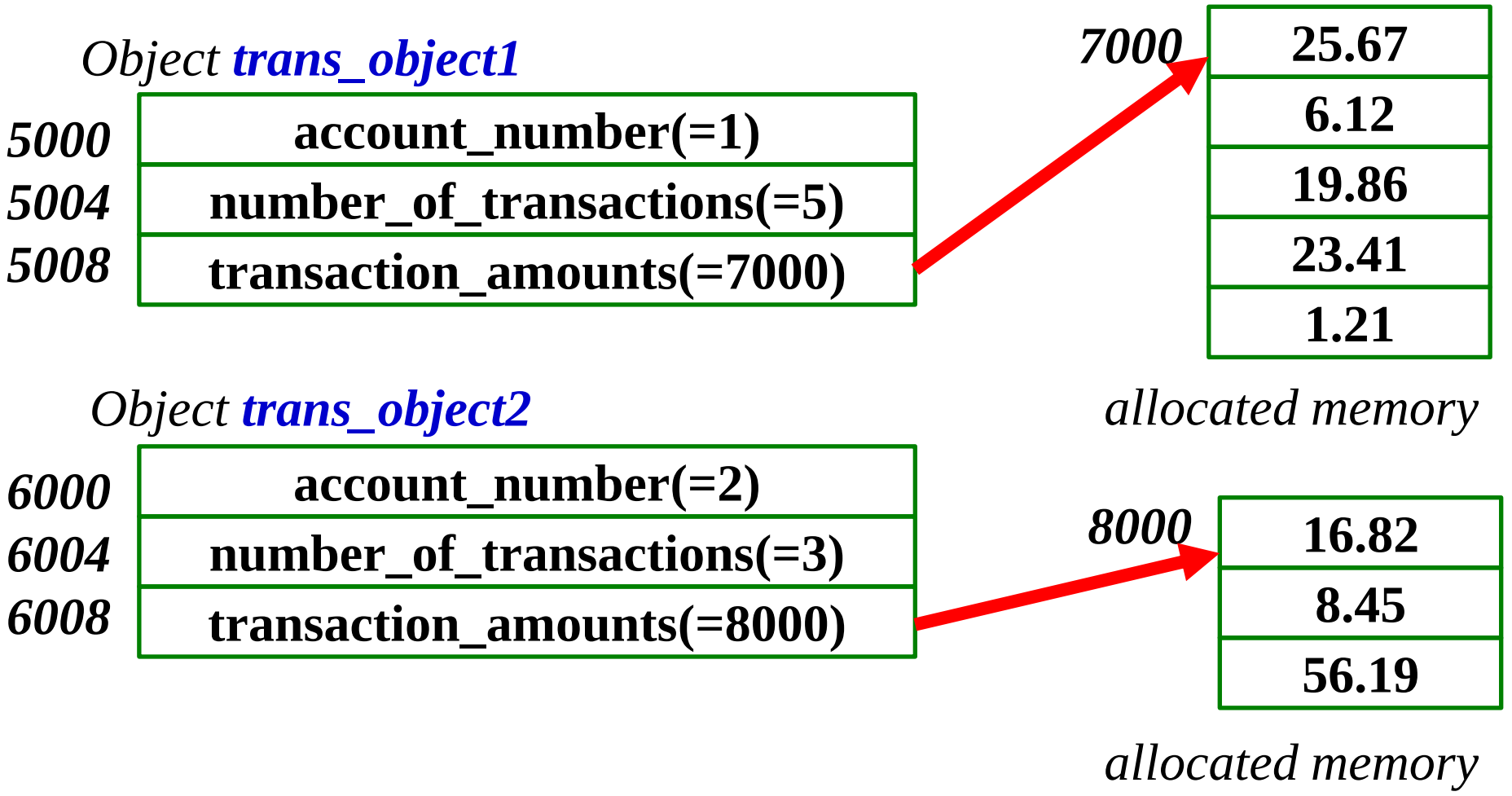
10.9.2 Assigning one object to another

- **Program Example P10J**

```
63 main()
64 {
65   float trans_amounts1[] = { 25.67, 6.12, 19.86, 23.41, 1.21 }
;
66   transactions trans_object1( 1, 5, trans_amounts1 ) ;
67   float trans_amounts2[] = { 16.82, 8.45, 56.19 } ;
68   transactions trans_object2( 2, 3, trans_amounts2 ) ;
```

10.9.2 Assigning one object to another

- 第68行之后，一个内存的简图可能如下所示。

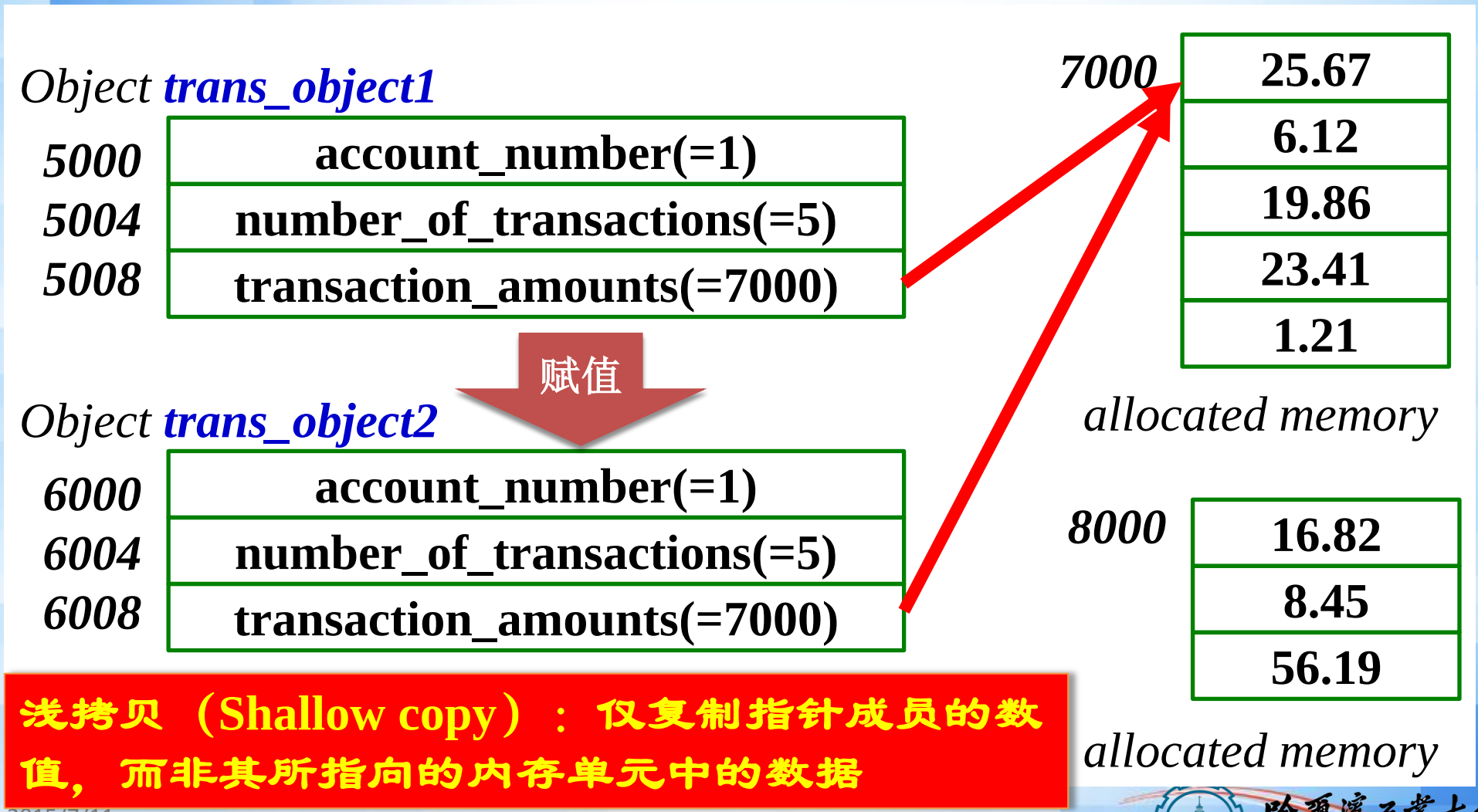


10.9.2 Assigning one object to another

```
70   cout << "Before assignment:" << endl ;
71   cout << "trans_object1:" << endl ;
72   trans_object1.display() ;
73   cout << "trans_object2:" << endl ;
74   trans_object2.display() ;
75
76   trans_object2 = trans_object1 ;
77
78 cout << endl << "After assignment:" << endl ;
79 cout << "trans_object1:" << endl ;
80 trans_object1.display() ;
81 cout << "trans_object2:" << endl ;
82 trans_object2.display() ;
```

•默认情况下，赋值运算将成员逐项复制。

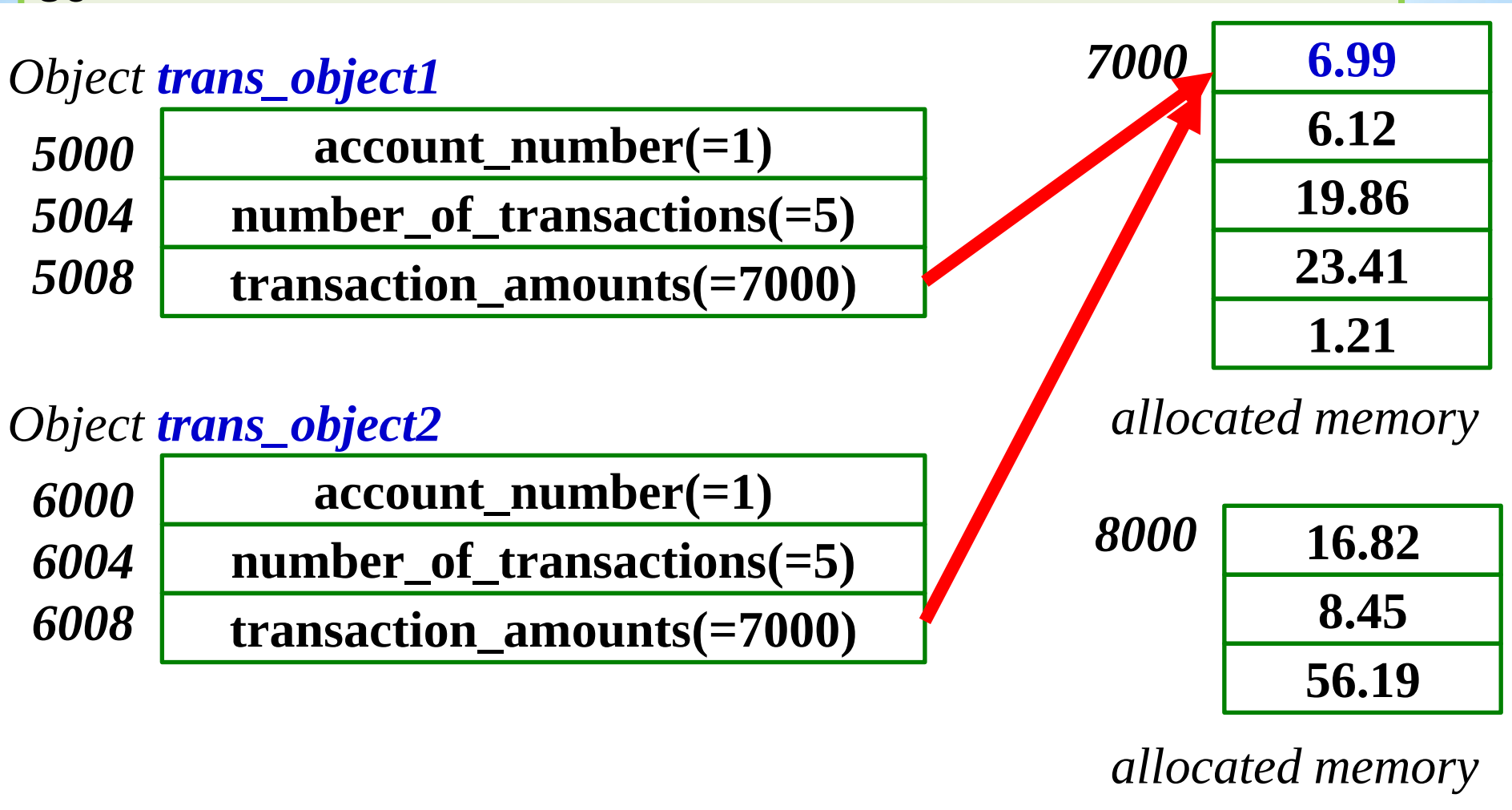
10.9.2 Assigning one object to another



浅拷贝（Shallow copy）：仅复制指针成员的数值，而非其所指向的内存单元中的数据

10.9.2 Assigning one object to another

```
84 // Change first transaction amount of trans_object1 to 9.99
85 trans_object1.modify_transaction( 0, 9.99 ) ;
86
```



10.9.2 Assigning one object to another

```
40 transactions::~transactions()
41 {
42   delete[] transaction_amounts ;
43 }
```

- 浅拷贝在对象析构时会有大麻烦！
  - 其中一个对象调用析构函数来释放指针所指向的内存区，同时也将另一个对象使用的内存区给释放了。
  - 当第二个对象调用析构函数时，就会出错：试图释放已经被释放完毕的内存
- 解决办法——深拷贝（deep copy）
  - 重载赋值运算符 =
  - 复制指针成员所指向的实际内存单元的值
  - 设计指针成员的内存重新申请、拷贝

```
1 // Program Example P10K
7 class transactions  // A class containing a pointer data member.
8 {//deep copy
9   public:
10   transactions( int ac_num, int num_transactions,
11   const float transaction_values[] ) ;
12   // Purpose: Constructor.
13   ~transactions() ;
14   // Purpose: Destructor.
15   void modify_transaction( int transaction_index,float amount ) ;
16   // Purpose: Modify a transaction amount.
17   // Arguments: transaction_index - transaction to change.
18   // amount - new transaction amount.
19   void display() const ;
20   // Purpose: Display transactions.
21    const transactions& operator=( const transactions& t ) ;
22   // Purpose: Overloaded assignment operator.
23   private:
24     int account_number ;
25     int number_of_transactions ;
26     float *transaction_amounts ; // Pointer to array of transactions.
27 } ;
```

10.9.2 Assigning one object to another

```
66 const transactions& transactions::operator=(const transactions& t)
67 {
68   if ( this != &t) // Avoid self-assignment.
69   { // Copy each member from t.
70     account_number = t.account_number ;
71     number_of_transactions = t.number_of_transactions ;
72     // Delete old array and allocate memory for new array.
73     delete[] transaction_amounts ;
74     transaction_amounts = new float[number_of_transactions] ;
75     // Copy transaction amounts from t.
76     for ( int i = 0 ; i < number_of_transactions ; i++ )
77         transaction_amounts[i] = t.transaction_amounts[i] ;
78   }
79     return *this ;
80 }
```

10.9.2 Assigning one object to another

```
82 main()
83 {
84    float trans_amounts1[] = { 25.67, 6.12, 19.86, 23.41, 1.21 } ;
85    transactions trans_object1( 1, 5, trans_amounts1 ) ;
86    float trans_amounts2[] = { 16.82, 8.45, 56.19 } ;
87    transactions trans_object2( 2, 3, trans_amounts2 ) ;
88
89    cout << "Before assignment:" << endl ;
90    cout << "trans_object1:" << endl ;
91    trans_object1.display() ;
92    cout << "trans_object2:" << endl ;
93    trans_object2.display() ;
94
95    trans_object2 = trans_object1 ;
```

第95行的等价语句：

```
trans_object2.operator=( trans_object1 ) ;
```

10.10 The copy constructor（拷贝构造函数）

- 拷贝构造函数(copy constructor)是一个特殊的构造函数，下列任何一种情况发生时，都会自动调用此函数:
  - (1) 创建并初始化一个新对象时。

```
float trans_amounts[] = { 56.88, 89.65, 15.76, 63.91 } ;
transactions trans_object1( 1, 4, trans_amounts ) ;
transactions trans_object2 = trans_object1 ;
// trans_object2 is created and initialised with
// trans_object1 member values by the copy constructor.
```

10.10 The copy constructor（拷贝构造函数）

- (2) 按传值方式将一个对象传给函数时。

```
main()
{
  void any_function( transaction tr ) ;
  float trans_amounts[] = { 39.87, 6.43, 7.34, 75.12, 9.65 } ;
  transactions trans( 3, 5, trans_amounts ) ;
  any_function ( trans ) ; // trans is passed by value to  the
function
                                  // i.e. a copy of trans is passed to tr.
...
}
void any_function( transactions tr )
{  // tr receives a copy of trans. The copy constructor does this.
  ...
}
```

10.10 The copy constructor（拷贝构造函数）

- (3) 从函数返回对象本身时。

```
main()
{
  ...
  transactions another_function() ;
  trans = another_function() ; // A copy of tr is passed back to trans.
                               // The copy constructor does this.
...
}
transactions another_function()
{
  float trans_amounts[] = { 70.00, 68.47} ;
  transactions tr( 4, 2, trans_amounts ) ;
  ...
  return tr ; // tr is returned by value.
}
```

10.10 The copy constructor（拷贝构造函数）

- 每个类都有一个默认的拷贝构造函数，它与默认的赋值运算符类似，逐个成员进行复制。
- 对于大多数的类而言，默认的拷贝构造函数和默认的赋值运算符都足以满足应用要求。
- 但是，对于需要动态分配和释放内存的类，默认的拷贝构造函数和默认的赋值运算符存在同样的问题。
- 一个定义了自己的赋值运算符的类也必须定义它自己的拷贝构造函数。

10.10 The copy constructor（拷贝构造函数）

```
1 // Program Example P10L
2 // Program to demonstrate the use of the copy constructor for
3 // a class with a pointer data member.
4 #include <iostream>
5 using namespace std ;
6
7 class transactions // A class containing a pointer data member.
8 {
9    public:
10    transactions( int ac_num, int num_transactions,
11    const float transaction_values[] ) ;
12    // Purpose: Constructor.
13    transactions( const transactions& tr ) ;
14    // Purpose: Copy Constructor.
15    ~transactions() ;
16   / / Purpose: Destructor.
```

10.10 The copy constructor（拷贝构造函数）

```
43 // Copy constructor.
44 transactions::transactions( const transactions& tr )
45 {
46   account_number = tr.account_number ;
47   number_of_transactions = tr.number_of_transactions ;
48   transaction_amounts = new float[number_of_transactions] ;
49   for ( int i = 0 ; i < number_of_transactions ; i++ )
50     transaction_amounts[i] = tr.transaction_amounts[i] ;
51 }
```

拷贝构造函数只能有一个形参，它是指向同一类的对象的引用。这个形参是一个const引用，拷贝构造函数不能通过这个引用来修改传递给它的对象。

```
69 void transactions::display() const
70 {
71   cout << " Account Number " << account_number << "
Transactions: " ;
72   for ( int i = 0 ; i < number_of_transactions ; i++ )
73      cout << transaction_amounts[i] << ' ' ;
74    cout << endl ;
75 }
```

The copy constructor is called

```
94 main()
95 {
96    float trans_amounts1[] = { 25.6     .12, 19.86, 23.41, 1.21 } ;
97    transactions trans_object1( 1, 5    trans_amounts1 ) ;
98    transactions trans_object2 = trans_object1 ;
99
```

```
trans_object1:
Account Number 1 Transactions: 25.67 6.12 19.86 23.41 1.21
trans_object2:
Account Number 1 Transactions: 25.67 6.12 19.86 23.41 1.21
```

```
1 // Program Example P10M
2 // Program to demonstrate the overloading of index operator [].
3 #include <iostream>
4 using namespace std ;
5
6 class int_array // A smart integer array.
7 {
8  public:
9     int_array( int number_of_elements = 10 ) ;
10    int_array( int_array const &array ) ;
11    ~int_array() ;
12    int_array const& operator=( int_array const &array ) ;
13    int &operator[]( int index ) ;
14  private:
15    int number_of_elements ;
16    int* data ;
17    void check_index( int index) const;
18    void copy_array( int_array const &array ) ;
19 } ;
```

总结：拷贝构造函数

- 作用
  - 默认拷贝构造函数可以完成对象的数据成员值简单的复制；
  - 自定义拷贝构造函数可以实现与默认拷贝构造函数不同的功能。
- 调用时机

  拷贝构造函数只适用于初始化：
  - 一个对象需要通过另一个对象进行初始化。
  - 一个对象以值传递的方式传入函数体。
  - 一个对象以值传递的方式从函数返回。

习题...

1. 设A为test类的对象且赋有初值,则语句test B=A;表示( )

A. 语法错误　　B. 为对象A定义一个别名

C. 利用对象A拷贝构造对象B　D. 仅说明B和A属于同一个类

2. 假定AB为一个类，则该类的拷贝构造函数的声明语句为_____。

A. AB &(AB x)　　B. AB(AB x)

C. AB(AB &)　　D. AB(AB * x)

3. 假定AB为一个类，则执行“AB a(4)，b[3]，* p[2] ;”语句时，自动调用该类构造函数的次数为_____。

A. 3　　B. 4　　C. 6　　D. 9

4. 假定AB为一个类，则执行“AB a[10];”语句时，系统自动调用该类的构造函数的次数为_____。

10.11 Overloading the index operator[ ]

```
21 // Constructor.
22 int_array::int_array( int n )
23 {
24   if ( n < 1 )
25   {
26     cerr << "number of elements cannot be " << n
27          << ", must be >= 1" << endl ;
28     exit( 1 ) ;
29   }
30   number_of_elements = n ;
31   data = new int [number_of_elements] ;
32   for (int i = 0 ; i < number_of_elements ; i++ )
33     data[ i ] = 0 ;// void *memset(void *s, int ch, size_t n);
34 }
```

10.11 Overloading the index operator[ ]

```
36 // Copy constructor.
37 int_array::int_array(int_array const &array)
38 {
39   copy_array(array) ;
40 }
41
42 // Destructor.
43 i
44 {
45
46 }
```

```
48 // Assignment operator.
49 int_array const& int_array::operator=( int_array const &array )
50 {
51   if ( this != &array ) // Avoid self assignment.
52   {
53     delete[] data ;
54     copy_array( array ) ;
55   }
56   return *this ;
57 }
```

10.11 Overloading the index operator[ ]

重载下标运算符[ ]

```
59 // Overloaded index operator.
60 int& int_array::operator[]( int index )
61 {
62   check_index( index ) ;
63   return data[ index ] ;
64 }

66 void int_array::copy_array( int_array const &array )
67 {
68   number_of_elements = array.number_of_elements ;
69   data = new int [ number_of_elements ] ;
70   for ( int i = 0 ; i < number_of_elements ; i++ )
71     data[ i ] = array.data[ i ] ;
72 }
```

10.11 Overloading the index operator[ ]

```
74 void int_array::check_index( int index ) const
75 {
76   if ( index < 0 || index >= number_of_elements )
77   {
78     cerr << "invalid index " << index
79          << ", range is 0 to " << number_of_elements - 1 << endl ;
80     exit( 1 ) ;
81   }
82 }
```

•如果下标的值是无效的，程序将会给出一个错误信息，并终止程序。

```
84 main()
85 {
86
87
88
89    // Display the contents of the array.
90    for ( i = 0 ; i < 15 ; i++ )
91      cout << a[i] << ' ' ;
92    cout << endl ;
93
94    // Assign some values to the elements of the array.
95    for ( i = 0 ; i < 15 ; i++ )
96      a[i] = i * 10 ;
97
98    // Display the new contents of the array.
99    for ( i = 0 ; i < 15 ; i++ )
100       cout << a[i] << ' ' ;
101   cout << endl ;
102
103    exit( 0 ) ;
104 }
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140
```

Programming Pitfalls (编程误区)

- 1.重载算术运算符（如+、-）并没有重载相应的复合运算符（如+=、-=），如果要使用，必须对它们单独进行重载。
- 2.使用一个像time24这样的类，以下将会正常工作:

```
time24 t( 1, 2, 3 ) ;
++( ++t ) ; // t is 1:02:05
```

最后一句等同于:

```
( t.operator++() ).operator++() ;
```

表达式t.operator++() 将t的值增1，然后返回t的引用用于第二次调用，于是t的值被增1两次

Programming Pitfalls

- 使用重载后置运算符++来做同样的测试

  time24 t(1, 2, 3) ;

  (t++)++ ; // t is now 1:02:04

  最后一句等同于:

  (t.operator++(0)).operator++(0) ;//0: dummy int argument

- 表达式**t.operator++(0)**将**t**的值增1，同时返回一个**临时对象**（不是t）的副本用于第二次调用。结果是**t**增1了一次，**临时对象**的副本也增1了一次。

- 解决方法：将**(t++)++;** 拆分成两个语句**t++;t++;** 能使**t**的值增**1**两次。

Programming Pitfalls

- 3. 如果一个类使用了指针数据成员，并动态分配内存，则需如下三项内容:
  - 重载的赋值运算符
  - 析构函数
  - 拷贝构造函数

如果一个类需要上述三者之一，那么必须会三个都需要。

Q & A