第 11 次课后作业

F1603407　516021910080 危国锐

提交截止时间：2019.11.29, 24:00

补交截止时间：2019.11.29, 24:00

第 10 章习题

一、简答题

1. 用 struct 定义类型与用 class 定义类型有什么区别？

答[1]：二者的主要区别在于成员的访问控制。在结构体中，缺省的访问特性是 public；在类中，缺省的访问特性是 private。

2. 构造函数和析构函数的作用是什么？它们各有什么特征？

答[1]：构造函数用于初始化一个对象，析构函数用于销毁一个对象。

构造函数在对象定义时被自动调用，其函数名同类名，不能指定返回类型，允许被重载。析构函数在对象生命周期结束时被自动调用，其函数名为~类名，不能指定返回类型，无形参，不允许被重载。

构造函数和析构函数只能在对象生命周期中被自动调用 1 次，而不能以对象名.函数名(实参表);的方式被程序员主动访问。

5. 在定义一个类时，哪些部分应放在头文件（.h 文件）中？哪些部分应放在实现文件（.cpp 文件）中？

答[1]：一般将类定义放在头文件中，将成员函数的实现和静态数据成员的定义放在实现文件中。

9. 什么是 this 指针？为什么要有 this 指针？

答[1]：类的成员函数在内存中仅有 1 个副本，由所有对象共享，故需引入一种使成员函数在被调用时能访问到正确对象的成员的机制。

注意到在调用成员函数时，函数体中提到的成员都是主调对象中的成员。因此 C++引入了一个指向主调对象的指针 this 作为成员函数的一个隐含参数，这样成员函数就能访问到当前对象的成员。

11. 复制构造函数的参数为什么一定要用引用传递，而不能用值传递？

答[1]：若复制构造函数的参数采用值传递，则在调用复制构造函数时，需要用实参构造其形参，这样又引起一次复制构造函数的调用，造成一个没有终止条件的对复制构造函数的递归调用。

12. 下面哪个类必须定义复制构造函数？

a. 包含 3 个 int 类型的数据成员的 point3 类。

b. 处理动态二维数组的 matrix 类。其中存储二维数组的空间在构造函数中动态分配，在析构函数中释放。

c. 本章定义的有理数类。

d. 处理一段文本的 word 类。所处理的文本存储在一个静态的字符数组中。

答[1]：b。因为若程序员不自行定义复制构造函数，编译器将生成默认复制构造函数，该函数将形参的数据成员对应地赋给正在创建的对象的数据成员，即生成一个与形式参数完全一样的对象。这样会造成两个对象的指向动态二维数组的那个数据成员指向同一块内存空间。如是，对其中一个对象的动态数组的任何操作也是对另一个对象的动态数组的操作，这显然不是程序员的本意。

二、程序设计题

7. 实现一个复数类，编写实现复数的加法、乘法、输入和输出的成员函数，并测试这个类。

- **加法规则：$(a+b_i)+(c+d_i)=(a+c)+(b+d)_i$。**
- **乘法规则：$(a+b_i)\times(c+d_i)=(ac-bd)+(bc+ad)_i$。**
- **输入规则：分别输入实部和虚部。**
- **输出规则：如果a是实部，b是虚部，输出格式为$a+bi$。**

[分析]

本题的主要目的是验证 C++语法。可从以下 3 个方面考虑。

A. 文件组织方式

要创建并测试一个 *complex_T* 类，可设计两个模块 complex_T.cpp, main.cpp 和一个头文件 complex_T.h。各文件内容如下：

①complex_T.h：类的定义。包括友元声明，静态非常量数据成员声明，静态常量成员的声明、定义和初始化，部分成员和友元（全局）函数的类中定义。

②complex_T.cpp：类的实现。在 complex.h 中声明而未定义的实体的定义，包括部分成员和友元函数的类外定义，静态非常量成员的定义和初始化等。

③main.cpp：测试程序（主模块）。定义几个 *complex_T* 类的对象，并对它们进行各种可能的操作。

B. 主要测试内容

①构造函数和析构函数的访问特性是否必须为 public。

②重载构造函数的绑定方式，当存在带参数默认值的重载时。

③对于已存在的对象，能否主动调用其构造函数，从而修改其数据成员。

④通过类名(实参表);方式调用构造函数，能否创建一个临时对象，从而能作

为 return 关键词后的表达式（函数的返回值）和复制/移动构造函数的实参。

⑤一次值返回的函数调用，会引起多少次关于临时对象的构造和析构。

⑥用初始化列表作赋值表达式的右运算对象为对象赋值。（C++11）

⑦用初始化列表初始化一个对象。（C++11）

⑧在类中定义的友元函数是否是全局的，从而其他模块只需 include 其所在的头文件，而无需声明即可调用该函数。

⑨将类的成员或友元的类外定义从源文件移到头文件中会发生什么。

⑩流提取运算符和流插入运算符重载函数的第二个形参能否增加或去除限定符 const。

C. 预期结论

①构造函数和析构函数的访问特性必须为 public，因为这两个函数需要被其他类的对象和函数调用。

②当存在带参数默认值的重载时，可能导致一个编译错误：有多个构造函数 "complex_T::complex_T" 的实例与参数列表匹配。因此，必须保证参数默认值不会引起重载函数调用时的**联编**（绑定/捆绑）二义性。

③作为对象的成员函数的构造函数的访问特性虽然是 public，但它只能在对象创建时被自动调用一次，而不能由程序员自行调用。因此，不能通过对象名.类名(实参表);的方式修改对象的数据成员。

④可以通过类名(实参表);的方式调用类的构造函数。该调用将创建并返回一个临时对象，从而可作为 return 后面的表达式或复制/移动构造函数的实参。

⑤一次值返回的函数调用，会引起一次构造和一次析构。在被调函数返回前，必会构造一个**局部**对象作为返回值。该局部对象在函数返回后就成为了主调函数中的**临时**对象，并替换函数调用。当调用语句执行完毕后，该临时对象被析构。

注意：虽然上述在函数体内定义的临时局部对象能被主调函数访问，但由于它在函数调用语句执行结束后将被立即销毁，所以不能将该函数设计成返回这个临时变量的引用。这样会导致一个编译警告，并使程序在执行时发生 Segmentation fault。

小结：值返回时，return 语句将局部（临时）对象**直接**变为主调函数中的临时对象。该语句本身不会构造临时对象，也不会析构 return 后面表达式的局部（临时）对象。

⑥初始化列表作赋值表达式的右运算对象时，会发生一次以初始化列表作实参表的对赋值表达式的左运算对象所属类的构造函数的调用，从而创建了一个临时对象，再以此临时对象为实参调用赋值运算符重载函数。这样就实现了为对象赋值。也就是可用对象名 ＝{初始化列表};的方式为对象赋值。

⑦可用类名 对象名{初始化列表};或类名 对象名 ＝{初始化列表};的方式定

义并初始化一个对象。该方式与类名 对象名(实参表);的初始化方式等效，即初始化列表相当于构造函数的实参表。当表中只有 1 个参数时，可省略花括号。也就是类名 对象名 = 参数;等价于类名 对象名(参数);。

⑧友元必须在类定义中声明，而友元的定义可在类中，也可以在类外（任一模块中）。

对于友元类外定义的情况，一个模块只需要 include 类定义（友元函数声明）所在的头文件，并将友元定义所在的模块加入编译和链接，即可使主调模块如访问全局实体一样访问友元。

对于友元函数类中定义（即在类定义中声明友元函数的同时定义）的情况，一个模块若要访问该函数，不仅需要 include 声明所在的头文件，还要在模块中作**全局函数声明**（即在**所有**语句块**外**作声明）。[存疑，2019/12/02]

对于类中定义的作为友元的运算符重载函数，主调模块仅需 include 类定义所在的头文件，不必再声明，不允许再定义。

⑨不应将实体（包括变量，函数等）的定义写到头文件中，这样容易引起链接错误[2]。因为若一个头文件中存在某实体的定义，且该头文件被多个模块 include，则每个模块编译产生的目标文件都会有该实体的目标代码。因此在链接时会发生重复定义。

正确的做法是在头文件中只作实体声明，而将实体定义写到一个模块中。这样，实体的目标代码就仅在一个模块的目标文件中出现，从而避免了重复定义。具体来说：

1)全局函数：在头文件写函数原型声明，在其中一个模块写函数定义。不能在头文件中声明的同时定义，因为编译器不会如对待类中定义的成员函数那样把全局函数处理成内联函数，故会产生重复定义的链接错误。

2)全局变量：在头文件中用 extern 关键词作外部变量**声明**，在其中一个模块定义。

3)静态的全局函数：必须在同一个模块（指包含了头文件后的源文件）里声明和定义（二者可分开，也可同时）。因为关键词 static 表示该函数对于每个模块都是专属的，链接时，不同模块的同名静态全局函数不会产生重复定义。若在头文件里只声明，而在另一模块写函数定义，则由于编译器在编译每个模块时不会到其他模块寻找静态函数的定义，在编译非函数定义所在模块时会产生 used but never defined 的编译警告，在链接时会产生 undefined reference to…的链接错误，除非代码中无调用该静态的全局函数的语句。

定义一个静态的全局函数，需要在函数声明前加关键词 static，并在同一模块中写函数定义。其中，在函数定义中的函数原型前**不必**加关键字 static。这样定义的函数是本模块专有的，不会与其他模块的同名函数产生“重复定义”，也不

会作为在其他模块中声明的函数的定义。

不允许将函数声明为[extern]（默认省略此关键字），却在同一模块中定义为static。总之，除非是在声明时同时定义，否则不建议在函数定义前加关键字 static。

静态的全局函数通常是声明并定义在一个模块中，作为该模块的工具函数。虽也可声明且定义在头文件中，但这样就相当于为每个模块都定义完全相同的工具函数。这样就违背了 static 的本意，应考虑把该函数变为 extern 的库函数。

4)静态的全局变量：同静态的全局函数。

5)类的静态非常量数据成员：在类定义中加 static 关键词（注意这只是作为变量声明，并非定义，故不允许指定初值），在类的实现模块中写变量定义。

6)类的静态常量数据成员：必须在类定义的同时指定初值。在 C++11 中，建议用 constexpr 限定符代替 const。

7)类的静态成员函数：可在类中定义，也可在类外定义。特别地，在类外定义时**不允许**加 static 关键词，因为不存在**联编**二义性。

⑩流提取运算符和流插入运算符重载函数的第二个形参的类型必须分别为非常量引用和常量引用，否则将引起编译错误或程序执行错误。

⑪在类定义中声明并同时定义成员或友元函数不会引起重复定义，是因为编译器默认将它们处理成内联函数，从而不会产生事实上的函数调用。因此可以将一些较简单的成员定义写到类定义中。

⑫返回对自身引用的常量成员函数，其返回类型必须是**常量**引用。返回其他引用的常量成员函数，可以是非常量引用，但建议规定为常量返回。

⑬同名、同形参表的常量成员函数和非常量成员函数被视作重载函数。常量对象只能访问常量成员函数。非常量对象可以访问所有成员函数，但有重载时优先**绑定**非常量成员函数。

⑭关于各模块 include 库的头文件和头文件保护符。加了该符后，在多模块编译-链接时，头文件的有效内容似乎只会有效一次。但由于各模块是单独编译生成目标代码，之后再链接成可执行代码，所以每个模块都要 include 库的头文件，而不能试图只在第一个被编译的模块中 include 该头文件。

小结：在多模块 include 同一个头文件时，若该头文件中无头文件保护符，会导致**链接**错误：重复定义。若试图仅在第一个编译的模块中 include 该头文件，则编译那些未 include 该头文件的模块时会发生**编译**错误：缺少声明/定义。

⑮建议实现文件要 include 自己的头文件，原因如下：

如是，当实现文件中出现与头文件中某函数声明原型不一致、且无法区分重载（如仅返回类型不同）的同名函数时，编译器会指出错误：不能区分重载。

否则，当实现文件中出现与头文件中某函数声明原型不一致、且无法区分重载（如仅返回类型不同）的同名函数时，编译器会认为头文件中的函数声明已被

实现，从而通过编译。程序运行时，会执行实现源文件中的函数体，造成提供给用户的函数原型和实际实现的函数原型不一致的情况。

无论实现文件是否 include 自己的头文件，当实现文件中无与在头文件中声明的函数[原型一致/或/不能区分重载]的函数定义时，编译器都会认为头文件中声明的函数缺定义。若有模块调用了已声明却缺定义的函数，编译器会指出错误。

⑯在多模块编译-链接中，一个非静态函数定义必须且只需在一个模块中出现。函数定义必须在所定义模块的所有语句块外，而不能被包含在某一语句块内。函数声明可以在不同模块或一个模块的不同地方反复出现（包括头文件中的声明），编译器（MinGW）会忽略重复的定义。

[源代码]

```
1   // complex_T.h
2   // complex_T 类的定义。该类表示了一个复数，实现复数的输入输出和代数运算。
3   #ifndef _complex_T
4   #define _complex_T
5   #include <iostream>
6
7   class complex_T
8   {
9       friend complex_T operator+(const complex_T &opL, const complex_T &opR);
10      friend complex_T operator*(const complex_T &opL, const complex_T &opR);
11      friend std::ostream &operator<<(std::ostream &os, const complex_T &op);
12      friend std::istream &operator>>(std::istream &is, complex_T &op);
13
14  private:
15      double real; // 实部
16      double imag; // 虚部
17
18  public:
19      complex_T(double r, double i) : real(r), imag(i)
20      {
21          std::cout << "call: complex_T(double r, double i)  : real(r), imag(i)\n";
22      }
23      complex_T(double r);
24      complex_T() = default;
25      ~complex_T()
26      {
27          std::cout << "call: ~complex_T()\n";
28      }
29      complex_T &add(const complex_T &opL, const complex_T &opR);
30      complex_T &mul(const complex_T &opL, const complex_T &opR);
```

```
31      complex_T &setVal(double r, double i);
32      const complex_T &display() const;
33      complex_T &getFromTerm();
34  };
35  #endif
36
37  // complex_T.cpp
38  // complex_T 类的实现。
39  #include "complex_T.h"
40  #include <iostream>
41
42  complex_T::complex_T(double r) : real(r)
43  {
44      std::cout << "call: complex_T(double r) : real(r)\n";
45  }
46
47  complex_T &complex_T::add(const complex_T &opL, const complex_T &opR)
48  {
49      real = opL.real + opR.real;
50      imag = opL.imag + opR.imag;
51      return *this;
52  }
53
54  complex_T &complex_T::mul(const complex_T &opL, const complex_T &opR)
55  {
56      real = opL.real * opR.real - opL.imag * opR.imag;
57      imag = opL.real * opR.imag + opL.imag * opR.real;
58      return *this;
59  }
60
61  complex_T &complex_T::setVal(double r, double i)
62  {
63      // 结论：这样会创建并返回临时对象，而不会修改主调对象的值。
64      /*
65      std::cout << "测试：主动调用构造函数，重设对象的数据成员\n";
66      complex_T(r, i);
67      return *this;
68      */
69      real = r;
70      imag = i;
71      return *this;
72  }
73
74  const complex_T &complex_T::display() const
```

```
75  {
76      if (imag < 0)
77          std::cout << real << " - " << -imag << 'i';
78      else
79          std::cout << real << " + " << imag << 'i';
80
81      return *this;
82  }
83
84  complex_T operator+(const complex_T &opL, const complex_T &opR)
85  {
86      // std::cout << "测试：主动调用构造函数，作为函数的返回值\n";
87      // 类名(形参表) 将调用构造函数，创建并初始化一个临时对象。
88      return complex_T{opL.real + opR.real, opL.imag + opR.imag};
89  }
90
91  complex_T operator*(const complex_T &opL, const complex_T &opR)
92  {
93      double real, imag;
94      real = opL.real * opR.real - opL.imag * opR.imag;
95      imag = opL.real * opR.imag + opL.imag * opR.real;
96      return complex_T(real, imag);
97  }
98
99  std::ostream &operator<<(std::ostream &os, const complex_T &op)
100 {
101     if (op.imag < 0)
102         os << op.real << " - " << -op.imag << 'i';
103     else
104         os << op.real << " + " << op.imag << 'i';
105     return os;
106 }
107
108 complex_T &complex_T::getFromTerm()
109 {
110     std::cout << "(real imag): ";
111     do
112     {
113         std::cin >> real >> imag;
114         if (std::cin.fail())
115         {
116             std::cin.clear();
117             std::cin.sync();
118             std::cout << "ERROR! Please reinput(real imag): ";
```

```
119             continue;
120         }
121         std::cin.sync();
122         break;
123     } while (true);
124
125     return *this;
126 }
127
128 std::istream &operator>>(std::istream &is, complex_T &op)
129 {
130     is >> op.real >> op.imag;
131     return is;
132 }
133
134 // Ex10_7.cpp
135 // 第 10 章_程序设计题_第 7 题
136 #include "complex_T.h"
137 #include <iostream>
138 #include <process.h> /* system */
139 #include <cstdlib>   /* getchar */
140 using namespace std;
141
142 int main(int argc, char const *argv[])
143 {
144     // 测试 1：对象定义、I/O 和赋值
145     cout << "Test1: Defining object, object I/O and assignment.\n";
146     cout << "complex_T r1(1), r2{2, 2}, r3;\n";
147     complex_T r1(1), r2{2, 1}, r3;
148
149     cout << "r1 = ";
150     r1.display();                   // 定义：仅初始化 1 个数据成员
151     cout << "r2 = " << r2 << endl; // 定义：使用初始化列表(C++11).
152
153     // r1.complex_T(1, 1); // 用户不能调用构造函数
154     cout << "\nr1.setVal(1, -1);\n";
155     cout << "r1 = " << r1.setVal(1, -1) << endl;
156
157     cout << "r2 = {2, 2};\n";
158     r2 = {2, 2}; // 赋值：使用初始化列表(C++11)。
159     cout << "r2 = " << r2 << endl;
160
161     cout << "r3 = " << r3 << endl; // 定义：使用默认构造函数
162     cout << "r3.getFromTerm();\n";
```

```
163     r3.getFromTerm(); // 赋值：通过终端
164     cout << "r3 = " << r3 << endl;
165 
166     // 测试 2：类的其他成员函数
167     cout << "\nTest2: Other member functions.\n";
168     cout << "r3.add(r1, r2);\n"
169          << '(' << r1 << ") + (" << r2 << ") = " << r3.add(r1, r2) << endl;
170     cout << "r3.mul(r1, r2);\n"
171          << '(' << r1 << ") * (" << r2 << ") = " << r3.mul(r1, r2) << endl;
172 
173     while (true)
174     {
175         // 重复测试，直到输入 n
176         cout << "Continue?(y/n): ";
177         if (cin.get() == 'n')
178             break;
179         cin.clear();
180         cin.sync();
181 
182         // 测试 3：运算符重载
183         cout << "\nTest3: Operator overloading.\n"
184              << "set r1:";
185         r1.getFromTerm();
186         cout << "set r2:";
187         r2.getFromTerm();
188         cout << '(' << r1 << ") + (" << r2 << ") = " << r1 + r2 << endl;
189         cout << '(' << r1 << ") * (" << r2 << ") = " << r1 * r2 << endl;
190     }
191 
192     // cin.sync();
193     // system("pause");
194     // getchar();
195     cout << "\nEnd of test...\n";
196     return 0;
197 }
```

[测试情况]

Test1: Defining object, object I/O and assignment.

complex_T r1(1), r2{2, 1}, r3;

call: complex_T(double r) : real(r)

call: complex_T(double r, double i) : real(r), imag(i)

r1 = 1 + 1.58101e-322i

r2 = 2 + 1i

r1.setVal(1, -1);
r1 = 1 - 1i
r2 = {2, 2};
call: complex_T(double r, double i) : real(r), imag(i)
call: ~complex_T()
r2 = 2 + 2i
r3 = 7.7134e-317 + 6.95234e-310i
r3.getFromTerm();
(real imag): *2 5↙*
r3 = 2 + 5i

Test2: Other member functions.
r3.add(r1, r2);
(1 - 1i) + (2 + 2i) = 3 + 1i
r3.mul(r1, r2);
(1 - 1i) * (2 + 2i) = 4 + 0i
Continue?(y/n): *y↙*

Test3: Operator overloading.
set r1:(real imag): *1 -1↙*
set r2:(real imag): *-1 1↙*
(1 - 1i) + (-1 + 1i) = call: complex_T(double r, double i) : real(r), imag(i)
0 + 0i
call: ~complex_T()
(1 - 1i) * (-1 + 1i) = call: complex_T(double r, double i) : real(r), imag(i)
0 + 2i
call: ~complex_T()
Continue?(y/n): *n↙*

End of test...
call: ~complex_T()
call: ~complex_T()
call: ~complex_T()

d:\SJTU\Bachelor\2019-2020,1\code\Ch10\Ex10_7.exe

```
Test1: Defining object, object I/O and assignment.
complex_T r1(1), r2{2, 1}, r3;
call: complex_T(double r) : real(r)
call: complex_T(double r, double i)  : real(r), imag(i)
r1 = 1 + 1.58101e-322i
r2 = 2 + 1i
r1.setVal(1, -1);
r1 = 1 - 1i
r2 = {2, 2};
call: complex_T(double r, double i)  : real(r), imag(i)
call: ~complex_T()
r2 = 2 + 2i
r3 = 7.7134e-317 + 6.95234e-310i
r3.getFromTerm();
(real imag): 2 5
r3 = 2 + 5i

Test2: Other member functions.
r3.add(r1, r2);
(1 - 1i) + (2 + 2i) = 3 + 1i
r3.mul(r1, r2);
(1 - 1i) * (2 + 2i) = 4 + 0i
Continue?(y/n): y

Test3: Operator overloading.
set r1:(real imag): 1 -1
set r2:(real imag): -1 1
(1 - 1i) + (-1 + 1i) = call: complex_T(double r, double i)  : real(r), imag(i)
0 + 0i
call: ~complex_T()
(1 - 1i) * (-1 + 1i) = call: complex_T(double r, double i)  : real(r), imag(i)
0 + 2i
call: ~complex_T()
Continue?(y/n): n

End of test...
call: ~complex_T()
call: ~complex_T()
call: ~complex_T()
```

[思考]

本题测试结果支持[分析]中的结论①~⑩。

本题验证了 C++关于类的基本语法特性，通过实例理清了头文件和源文件的关系，从底层角度加深了对程序设计过程之**编码-编译-链接**的理解。

参考资料

[1]翁惠玉. C++程序设计题解与拓展[M]. 北京 ： 清华大学出版社, 2013. ISBN 978-7-302-33891-8.

[2]https://blog.csdn.net/u012332816/article/details/86535333