

实验1 现代C++编程初体验

说明

一、实验目的

二、实验准备

三、实验内容

1. 实验任务1

2. 实验任务2

3. 实验任务3

4. 实验任务4

四、实验结论

1. 实验任务1

2. 实验任务2

3. 实验任务3

4. 实验任务4

五、实验总结(选)

六、实验文档提交要求

七、博客园编辑器使用补充说明

说明

• 阅读文档

请阅读文档，明确各项任务的具体要求、提交方式与截止时间。

• 编程建议

随文档准备了部分代码，对测试代码，为避免录入错误，请直接从代码压缩包解压缩后使用。

设计性实验，请先独立思考。缺失主动思考和试错过程，会削弱思维训练与编码能力训练。

• 提交规范

- 一次实验的所有任务写在一篇博客中
- 发布博客后，还需要在作业系统中提交链接
- 晚交的实验，按课程规则处理（分数折半）

一、实验目的

- 加深对OOP概念（**类、对象**）和特性（**封装**）的理解
- 会用C++正确定义、实现、测试类；会创建对象，并基于对象编程
- 加深对C++内存资源管理技术的理解，能够解释构造函数、析构函数的用途，分析它们何时会被调用
- 会用多文件方式组织代码
- 针对具体问题场景，练习运用面向对象思维设计，合理利用C++语言特性（封装与访问权限控制, *static*, *friend*, *const*），在数据共享和保护之间达到平衡

二、实验准备

系统浏览教材第4-5章，熟悉以下内容：

- 类的抽象、设计
- 使用C++定义类、使用类创建对象的语法；构造/析构函数语法、用途及调用时机
- 类的共享与保护机制：访问权限控制、静态成员(*static*)、友元(*friend*)、*const*
- 多文件组织代码

三、实验内容

1. 实验任务1

验证性实验：简单类T的定义和测试。实践、阅读代码，回答问题。

这个简单任务覆盖以下内容：

- 类的定义（封装）
- 类的使用：对象的创建、访问
- 数据共享
 - 在同一个对象的所有操作之间共享 —— 实现基础：封装
 - 在同一个类的所有对象之间共享 —— 实现机制：类的*static*成员
 - 在不同模块（类、函数）之间共享 —— 实现机制：友元
- 数据保护机制
 - *const*对象
 - *const*引用作为形参
 - *const*成员数据/*const*成员函数
- 代码组织方式：多文件结构

代码组织：

- T.h 内容：类T的声明、友元函数声明
- T.cpp 内容：类T的实现、友元函数实现
- task1.cpp 内容：测试模块、*main*函数

T.h

```
1  #pragma once
2
3  #include <string>
4
5  // 类T：声明
```

```

6  class T {
7  // 对象属性、方法
8  public:
9      T(int x = 0, int y = 0);    // 普通构造函数
10     T(const T &t);    // 复制构造函数
11     T(T &&t);        // 移动构造函数
12     ~T();            // 析构函数
13
14     void adjust(int ratio);      // 按系数成倍调整数据
15     void display() const;       // 以(m1, m2)形式显示T类对象信息
16
17 private:
18     int m1, m2;
19
20 // 类属性、方法
21 public:
22     static int get_cnt();        // 显示当前T类对象总数
23
24 public:
25     static const std::string doc;    // 类T的描述信息
26     static const int max_cnt;       // 类T对象上限
27
28 private:
29     static int cnt;               // 当前T类对象数目
30
31 // 类T友元函数声明
32     friend void func();
33 };
34
35 // 普通函数声明
36 void func();

```

T.cpp

```

1  #include "T.h"
2  #include <iostream>
3  #include <string>
4
5  // 类T实现
6
7  // static成员数据类外初始化
8  const std::string T::doc{"a simple class sample"};
9  const int T::max_cnt = 999;
10 int T::cnt = 0;
11
12 // 类方法
13 int T::get_cnt() {
14     return cnt;
15 }
16
17 // 对象方法

```

```

18 T::T(int x, int y): m1{x}, m2{y} {
19     ++cnt;
20     std::cout << "T constructor called.\n";
21 }
22
23 T::T(const T &t): m1{t.m1}, m2{t.m2} {
24     ++cnt;
25     std::cout << "T copy constructor called.\n";
26 }
27
28 T::T(T &&t): m1{t.m1}, m2{t.m2} {
29     ++cnt;
30     std::cout << "T move constructor called.\n";
31 }
32
33 T::~~T() {
34     --cnt;
35     std::cout << "T destructor called.\n";
36 }
37
38 void T::adjust(int ratio) {
39     m1 *= ratio;
40     m2 *= ratio;
41 }
42
43 void T::display() const {
44     std::cout << "(" << m1 << ", " << m2 << ")" ;
45 }
46
47 // 普通函数实现
48 void func() {
49     T t5(42);
50     t5.m2 = 2049;
51     std::cout << "t5 = "; t5.display(); std::cout << '\n';
52 }

```

task1.cpp

```

1  #include "T.h"
2  #include <iostream>
3
4  void test_T();
5
6  int main() {
7      std::cout << "test Class T: \n";
8      test_T();
9
10     std::cout << "\ntest friend func: \n";
11     func();
12 }
13

```

```

14 void test_T() {
15     using std::cout;
16     using std::endl;
17
18     cout << "T info: " << T::doc << endl;
19     cout << "T objects'max count: " << T::max_cnt << endl;
20     cout << "T objects'current count: " << T::get_cnt() << endl << endl;
21
22     T t1;
23     cout << "t1 = "; t1.display(); cout << endl;
24
25     T t2(3, 4);
26     cout << "t2 = "; t2.display(); cout << endl;
27
28     T t3(t2);
29     t3.adjust(2);
30     cout << "t3 = "; t3.display(); cout << endl;
31
32     T t4(std::move(t2));
33     cout << "t4 = "; t4.display(); cout << endl;
34
35     cout << "test: T objects'current count: " << T::get_cnt() << endl;
36 }

```

正确录入后，编译、运行，预期测试结果：

```

test Class T:
T info: a simple class sample
T objects'max count: 999
T objects'current count: 0

T constructor called.
t1 = (0, 0)
T constructor called.
t2 = (3, 4)
T copy constructor called.
t3 = (6, 8)
T move constructor called.
t4 = (3, 4)
test: T objects'current count: 4
T destructor called.
T destructor called.
T destructor called.
T destructor called.

test friend func:
T constructor called.
t5 = (42, 2049)
func: T objects'current count: 1
T destructor called.

```

结合运行结果，理解代码，回答问题：

问题1：

T.h中，在类T内部，已声明 `func` 是T的友元函数。在类外部，去掉line36，重新编译，程序能否正常运行。

如果能，回答**YES**；如果不能，以截图形式提供编译报错信息，说明原因。

```
35 // 普通函数声明
36 void func();
```

问题2:

T.h中, line9-12给出了各种构造函数、析构函数。总结它们各自的功能、调用时机。

```
9 T(int x = 0, int y = 0); // 普通构造函数
10 T(const T &t); // 复制构造函数
11 T(T &&t); // 移动构造函数
12 ~T(); // 析构函数
```

问题3:

T.cpp中, line13-15, 剪切到T.h的末尾, 重新编译, 程序能否正确编译。

如不能, 以截图形式给出报错信息, 分析原因。

```
8 const std::string T::doc{"a simple class sample"};
9 const int T::max_cnt = 999;
10 int T::cnt = 0;
```

2. 实验任务2

不使用C++标准库提供的复数模板类, 设计并实现一个简化版复数类`Complex`。要求如下:

- 类属性
 - doc: 用于类说明, `std::string` 类型, 常量, 公有
 - 说明信息为: a simplified complex class
- 对象属性
 - 用于表示复数的实部`real`和虚部`imag`, 均为小数形式, 私有
- 对象方法
 - 构造函数
 - 要求支持以下方式构造复数对象

```
1 Complex c1; // 构造的复数对象, 对应数学中的0+0i
2 Complex c2(3.5); // 构造的复数对象, 对应数学中的3.5+0i
3 Complex c3(3,-4); // 构造的复数对象, 对应数学中的3-4i
4 Complex c4(c2); // 用已经存在的复数对象c2构造c4
5 Complex c5 = c2; // 用已经存在的复数对象c2构造c5
```

- 接口
 - `get_real()` 返回复数实部
 - `get_imag()` 返回复数虚部
 - `add()` 用于把一个复数加到自身, 用法: `c1.add(c2)`, 相当于 `c1 += c2`
- 友元函数

- `output()` 用于输出一个复数，以 $a+bi$ 的形式，如 $3 + 4i$
- `abs()` 用于对复数取模。比如，`Complex c(3, 4);`调用`abs(c)`结果为5.0
- `add()` 用于实现两个复数相加，返回复数。比如 `c3 = add(c1, c2)`
- `isEqual()` 用于判断两个复数是否相等，相等返回 `true`，否则，返回 `false`
- `isNotEqual()` 用于判断两个复数是否相等，不相等返回 `true`，否则，返回 `false`
- 代码要求

采用多文件组织代码：

- `Complex.h` 类`Complex`声明、友元函数声明
- `Complex.cpp` 类`Complex`实现、友元函数实现
- `task2.cpp` 测试代码、`main()`函数 **(测试代码已经在task2.cpp给出)**
- 类设计及编码风格要求
 - 设计并实现类时，合理利用数据的共享和保护机制，尽可能在数据共享和保护之间达到平衡
 - 关注编码风格、可读性、易维护性

task2.cpp源码

```

1  // 待补足头文件
2  // xxx
3  #include <iostream>
4  #include <iomanip>
5  #include <complex>
6
7  void test_Complex();
8  void test_std_complex();
9
10 int main() {
11     std::cout << "*****测试1：自定义类Complex*****\n";
12     test_Complex();
13
14     std::cout << "\n*****测试2：标准库模板类complex*****\n";
15     test_std_complex();
16 }
17
18 void test_Complex() {
19     using std::cout;
20     using std::endl;
21     using std::boolalpha;
22
23     cout << "类成员测试： " << endl;
24     cout << Complex::doc << endl << endl;
25
26     cout << "Complex对象测试： " << endl;
27     Complex c1;
28     Complex c2(3, -4);
29     Complex c3(c2);
30     Complex c4 = c2;
31     const Complex c5(3.5);
32

```

```

33     cout << "c1 = "; output(c1); cout << endl;
34     cout << "c2 = "; output(c2); cout << endl;
35     cout << "c3 = "; output(c3); cout << endl;
36     cout << "c4 = "; output(c4); cout << endl;
37     cout << "c5.real = " << c5.get_real()
38         << ", c5.imag = " << c5.get_imag() << endl << endl;
39
40     cout << "复数运算测试: " << endl;
41     cout << "abs(c2) = " << abs(c2) << endl;
42     c1.add(c2);
43     cout << "c1 += c2, c1 = "; output(c1); cout << endl;
44     cout << boolalpha;
45     cout << "c1 == c2 : " << is_equal(c1, c2) << endl;
46     cout << "c1 != c2 : " << is_not_equal(c1, c2) << endl;
47     c4 = add(c2, c3);
48     cout << "c4 = c2 + c3, c4 = "; output(c4); cout << endl;
49 }
50
51 void test_std_complex() {
52     using std::cout;
53     using std::endl;
54     using std::boolalpha;
55
56     cout << "std::complex<double>对象测试: " << endl;
57     std::complex<double> c1;
58     std::complex<double> c2(3, -4);
59     std::complex<double> c3(c2);
60     std::complex<double> c4 = c2;
61     const std::complex<double> c5(3.5);
62
63     cout << "c1 = " << c1 << endl;
64     cout << "c2 = " << c2 << endl;
65     cout << "c3 = " << c3 << endl;
66     cout << "c4 = " << c4 << endl;
67
68     cout << "c5.real = " << c5.real()
69         << ", c5.imag = " << c5.imag() << endl << endl;
70
71     cout << "复数运算测试: " << endl;
72     cout << "abs(c2) = " << abs(c2) << endl;
73     c1 += c2;
74     cout << "c1 += c2, c1 = " << c1 << endl;
75     cout << boolalpha;
76     cout << "c1 == c2 : " << (c1 == c2) << endl;
77     cout << "c1 != c2 : " << (c1 != c2) << endl;
78     c4 = c2 + c3;
79     cout << "c4 = c2 + c3, c4 = " << c4 << endl;
80 }

```

合理设计并实现自定义类**Complex**，使得代码编写完成后，测试代码预期输出结果如下：

```

*****测试1: 自定义类Complex*****
类成员测试:
a simplified Complex class

Complex对象测试:
c1 = 0 + 0i
c2 = 3 - 4i
c3 = 3 - 4i
c4 = 3 - 4i
c5.real = 3.5, c5.imag = 0

复数运算测试:
abs(c2) = 5
c1 += c2, c1 = 3 - 4i
c1 == c2 : true
c1 != c2 : false
c4 = c2 + c3, c4 = 6 - 8i

```

```

*****测试2: 标准库模板类complex*****
std::complex<double>对象测试:
c1 = (0,0)
c2 = (3,-4)
c3 = (3,-4)
c4 = (3,-4)
c5.real = 3.5, c5.imag = 0

复数运算测试:
abs(c2) = 5
c1 += c2, c1 = (3,-4)
c1 == c2 : true
c1 != c2 : false
c4 = c2 + c3, c4 = (6,-8)

```

比较自定义类 `Complex` 和标准库模板类 `complex` 的使用，回答问题：

问题1：

比较自定义类 `Complex` 和标准库模板类 `complex` 的用法，在使用形式上，哪一种更简洁？函数和运算内在有关联吗？

自定义类 <code>Complex</code>	标准库模板类 <code>complex</code>
<code>c1.add(c2)</code>	<code>c1 += c2</code>
<code>c4 = add(c2, c3)</code>	<code>c4 = c2 + c3</code>
<code>is_equal(c1, c2)</code>	<code>c1 == c2</code>
<code>is_not_equal(c1, c2)</code>	<code>c1 != c2</code>
<code>output(c1)</code>	<code>cout << c1</code>
<code>abs(c2)</code>	<code>abc(c2)</code>

问题2：

2-1：自定义 `Complex` 中，`output/abs/add/` 等均设为友元，它们真的需要访问 **私有数据** 吗？（回答“是/否”并给出理由）

2-2：标准库 `std::complex` 是否把 `abs` 设为友元？（查阅 [cppreference](#) 后回答）

2-3：什么时候才考虑使用 `friend`？总结你的思考。

问题3：

如果构造对象时禁用=形式，即遇到 `Complex c4 = c2`；编译报错，类 `Complex` 的设计应如何调整？

3. 实验任务3

设计并实现播放器控制类 `PlayerControl`，模拟音频播放器中控制操作：**播放、暂停、下一首、上一首、结束、退出**。

(仅模拟播放控制器行为，暂不实现真实音频播放控制)

- 播放控制，用枚举类型表示，限定以下枚举值:

```
enum class ControlType {Play, Pause, Next, Prev, Stop, Unknown};
```

- 类属性

```
total_cnt, int类型, 私有, 用于记录播放控制操作总次数
```

- 类方法

```
int get_cnt(), 公有, 用于获取当前播放控制操作总数
```

- 接口

```
ControlType parse(const std::string& control_str);
```

负责将用户输入的控制命令串转换成枚举值
解析转换时，忽略大小写。例如，用户输入字符串"play", "Play", "PLAY", 均会被解析成枚举值Play

```
void execute(ControlType cmd) const;
```

负责模拟执行控制命令

例如，当cmd是枚举值Play时，屏幕上输出"播放"（暂不实现实际播放行为）

- 代码组织

多文件方式

PlayerControl.h 播放控制类PlayerControl声明

PlayerControl.cpp 播放控制类PlayerControl实现

task3.cpp 测试模块 + main

待补足代码如下：

PlayerControl.h

```
1  #pragma once
2  #include <string>
3
4  enum class ControlType {Play, Pause, Next, Prev, Stop, Unknown};
5
6  class PlayerControl {
7  public:
8      PlayerControl();
9
10     ControlType parse(const std::string& control_str);    // 实现std::string -->
ControlType转换
11     void execute(ControlType cmd) const;    // 执行控制操作（以打印输出模拟）
12
13     static int get_cnt();
14
15 private:
16     static int total_cnt;
17 };
```

PlayerControl.cpp

```
1 #include "PlayerControl.h"
2 #include <iostream>
3 #include <algorithm>
4
5 int PlayerControl::total_cnt = 0;
6
7 PlayerControl::PlayerControl() {}
8
9 // 待补足
10 // 1. 将输入字符串转为小写，实现大小写不敏感
11 // 2. 匹配"play"/"pause"/"next"/"prev"/"stop"并返回对应枚举
12 // 3. 未匹配的字符串返回ControlType::Unknown
13 // 4. 每次成功调用parse时递增total_cnt
14 ControlType PlayerControl::parse(const std::string& control_str) {
15     // xxx
16 }
17
18 void PlayerControl::execute(ControlType cmd) const {
19     switch (cmd) {
20     case ControlType::Play: std::cout << "[play] Playing music...\n"; break;
21     case ControlType::Pause: std::cout << "[Pause] Music paused\n"; break;
22     case ControlType::Next: std::cout << "[Next] Skipping to next track\n"; break;
23     case ControlType::Prev: std::cout << "[Prev] Back to previous track\n"; break;
24     case ControlType::Stop: std::cout << "[Stop] Music stopped\n"; break;
25     default: std::cout << "[Error] unknown control\n"; break;
26     }
27 }
28
29 int PlayerControl::get_cnt() {
30     return total_cnt;
31 }
```

task3.cpp

```
1 #include "PlayerControl.h"
2 #include <iostream>
3
4 void test() {
5     PlayerControl controller;
6     std::string control_str;
7     std::cout << "Enter Control: (play/pause/next/prev/stop/quit):\n";
8
9     while(std::cin >> control_str) {
10         if(control_str == "quit")
11             break;
12
13         ControlType cmd = controller.parse(control_str);
14         controller.execute(cmd);
15         std::cout << "Current Player control: " << PlayerControl::get_cnt() << "\n\n";
16     }
```

```
17 }  
18  
19 int main() {  
20     test();  
21 }
```

PlayerControl1.cpp 正确补足后，预期测试效果如下：

(注意，无论用户输入的控制命令是小写、大写还是混合，最后都转换成对应枚举值。需要在 `parse` 接口实现对此的支持)

```
Enter Control: (play/pause/next/prev/stop/quit):  
play  
[play] Playing music...  
Current Player control: 1  
  
Play  
[play] Playing music...  
Current Player control: 2  
  
next  
[Next] Skipping to next track  
Current Player control: 3  
  
STOP  
[Stop] Music stopped  
Current Player control: 4  
  
quit
```

思考 (选做*)

如果希望输模拟播放控制时，输出控制更现代 (使用 [emoji](#))，如下测试截图所示。如何调整代码实现？

```
Enter Control: (play/pause/next/prev/stop/quit):  
play  
🎵 Playing music...  
Current Player control: 1  
  
next  
▶▶ Skipping to next track  
Current Player control: 2  
  
prev  
◀◀ Back to previous track  
Current Player control: 3  
  
stop  
■ Music stopped  
Current Player control: 4  
  
quit
```

4. 实验任务4

设计并实现一个分数类`Fraction`. 要求如下:

- 类属性

doc: 用于类说明, `std::string` 类型, 常量, 公有

说明信息为:

`Fraction`类 v 0.01版.

目前仅支持分数对象的构造、输出、加/减/乘/除运算.

- 对象属性

用于表示分数的分子`up`和分母`down`, 均为整数形式, 私有

- 对象方法

- 构造函数

要求支持以下方式构造分数对象

```
1 Fraction f1(2);           // 构造的分数对象, 对应数学中的分数2/1, 分母为1
2 Fraction f2(2, -3);       // 构造的分数对象, 对应数学中的分数-2/3
3 Fraction f3(f2);          // 用已经存在的分数对象, 构造新的分数对象
```

- 接口

- `get_up()` 返回分子
- `get_down()` 返回分母
- `negative()` 用于求负, 支持 `Fraction f2 = f1.negative()`, 形如 `f2 = -f1`

求负运算, 分数对象`f1`本身不变, 返回值是求负后的分数对象

- 工具函数

这些工具函数以何种方案实现 (友元/命名空间+自由函数/类+static函数), 请自行设计。

- `output()` 用于输出一个分数, 以形如2/3或-2/3这样的形式
(输出化简后的形式, 比如-4/10, 输出时化简后的-2/5)
- `add()` 用于实现两个分数相加, 返回分数。比如 `f3 = add(f1, f2)`
- `sub()` 用于实现两个分数相减, 返回分数。比如 `f3 = sub(f1, f2)`
- `mul()` 用于实现两个分数相乘, 返回分数。比如 `f3 = mul(f1, f2)`
- `div()` 用于实现两个分数相除法, 返回分数。比如 `f3 = div(f1, f2)`

- 代码组织

要求采用多文件方式

- `Fraction.h` 类`Fraction`声明、其他声明
- `Fraction.cpp` 类`Fraction`实现、其他实现
- `task4.cpp` 测试代码、`main()`函数 (测试代码已经在`task4.cpp`给出)

(如确定方案时, 需新增文件或调整`task4.cpp`内运算调用方式, 请根据方案做微调)

- 类设计及编码风格要求

- 设计并实现类时，可设计必要的内部工具函数辅助分数计算；合理利用数据的共享和保护机制
- 关注编码风格、可读性、易维护性

task4.cpp

```
1  #include "Fraction.h"
2  #include <iostream>
3
4  void test1();
5  void test2();
6
7  int main() {
8      std::cout << "测试1: Fraction类基础功能测试\n";
9      test1();
10
11      std::cout << "\n测试2: 分母为0测试: \n";
12      test2();
13  }
14
15  void test1() {
16      using std::cout;
17      using std::endl;
18
19      cout << "Fraction类测试: " << endl;
20      cout << Fraction::doc << endl << endl;
21
22      Fraction f1(5);
23      Fraction f2(3, -4), f3(-18, 12);
24      Fraction f4(f3);
25      cout << "f1 = "; output(f1); cout << endl;
26      cout << "f2 = "; output(f2); cout << endl;
27      cout << "f3 = "; output(f3); cout << endl;
28      cout << "f4 = "; output(f4); cout << endl;
29
30      const Fraction f5(f4.negative());
31      cout << "f5 = "; output(f5); cout << endl;
32      cout << "f5.get_up() = " << f5.get_up()
33          << ", f5.get_down() = " << f5.get_down() << endl;
34
35      cout << "f1 + f2 = "; output(add(f1, f2)); cout << endl;
36      cout << "f1 - f2 = "; output(sub(f1, f2)); cout << endl;
37      cout << "f1 * f2 = "; output(mul(f1, f2)); cout << endl;
38      cout << "f1 / f2 = "; output(div(f1, f2)); cout << endl;
39      cout << "f4 + f5 = "; output(add(f4, f5)); cout << endl;
40  }
41
42  void test2() {
43      using std::cout;
44      using std::endl;
45
46      Fraction f6(42, 55), f7(0, 3);
```

```

47     cout << "f6 = "; output(f6); cout << endl;
48     cout << "f7 = "; output(f7); cout << endl;
49     cout << "f6 / f7 = "; output(div(f6, f7)); cout << endl;
50 }

```

Fraction类正确合计、实现后，预期测试结果如下：

```

测试1: Fraction类基础功能测试
Fraction类测试:
Fraction类 v 0.01版.
目前仅支持分数对象的构造、输出、加/减/乘/除运算.

f1 = 5
f2 = -3/4
f3 = -3/2
f4 = -3/2
f5 = 3/2
f5.get_up() = 3, f5.get_down() = 2
f1 + f2 = 17/4
f1 - f2 = 23/4
f1 * f2 = -15/4
f1 / f2 = -20/3
f4 + f5 = 0

测试2: 分母为0测试:
f6 = 42/55
f7 = 0
f6 / f7 = 分母不能为0

```

问题回答：

分数的输出和计算，`output/add/sub/mul/div`，你选择的是哪一种设计方案？（友元/自由函数/命名空间+自由函数/类+static）

你的决策理由？如友元方案的优缺点、静态成员函数方案的适用场景、命名空间方案的考虑因素等。

四、实验结论

1. 实验任务1

此部分书写内容：

- 分别给出T.h, T.cpp, task1.cpp源码，以及，运行测试结果截图
- 回答问题

问题1：

T.h中，在类T内部，已声明 `func` 是T的友元函数。在类外部，去掉line36，重新编译，程序能否正常运行。

如果能，回答**YES**；如果不能，以截图形式提供编译报错信息，说明原因。

```

35     // 普通函数声明
36     void func();

```

问题2:

T.h中, line9-12给出了各种构造函数、析构函数。总结它们各自的功能、调用时机。

```
9      T(int x = 0, int y = 0);    // 普通构造函数
10     T(const T &t);             // 复制构造函数
11     T(T &&t);                   // 移动构造函数
12     ~T();                       // 析构函数
```

问题3:

T.cpp中, line13-15, 剪切到T.h的末尾, 重新编译, 程序能否正确编译。

如不能, 以截图形式给出报错信息, 分析原因。

```
8  const std::string T::doc{"a simple class sample"};
9  const int T::max_cnt = 999;
10 int T::cnt = 0;
```

2. 实验任务2

此部分书写内容:

- 分别给出Complex.h, Complex.cpp, task2.cpp源码, 及, 运行测试结果截图
- 回答问题

问题1:

比较自定义类 `Complex` 和标准库模板类 `complex` 的用法, 在使用形式上, 哪一种更简洁? 函数和运算内在有关联吗?

自定义类 <code>Complex</code>	标准库模板类 <code>complex</code>
<code>c1.add(c2)</code>	<code>c1 += c2</code>
<code>c4 = add(c2, c3)</code>	<code>c4 = c2 + c3</code>
<code>is_equal(c1, c2)</code>	<code>c1 == c2</code>
<code>is_not_equal(c1, c2)</code>	<code>c1 != c2</code>
<code>output(c1)</code>	<code>cout << c1</code>
<code>abs(c2)</code>	<code>abc(c2)</code>

问题2:

2-1: 自定义 `Complex` 中, `output/abs/add/` 等均设为友元, 它们真的需要访问 **私有数据** 吗? (回答“是/否”并给出理由)

2-2: 标准库 `std::complex` 是否把 `abs` 设为友元? (查阅 [cppreference](#)后回答)

2-3: 什么时候才考虑使用 friend? 总结你的思考。

问题3:

如果构造对象时禁用=形式, 即遇到 `Complex c4 = c2;` 编译报错, 类Complex的设计应如何调整?

3. 实验任务3

此部分书写内容：

- 分别给出PlayerControl.h, PlayerControl.cpp, task3.cpp源码，及，运行测试结果截图
- 思考（选做*）

如果希望输模拟播放控制时，输出控制更现代（使用emoji），如下测试截图所示。如何调整代码实现？

如选做，给出实现emoji输出的PlayerControl.h, PlayerControl.cpp, task3.cpp源码，及，运行测试结果截图

4. 实验任务4

此部分书写内容：

- 分别给出程序Fraction.h, Fraction.cpp, task4.cpp源码，及，运行测试结果截图
- 问题回答

分数的输出和计算，`output/add/sub/mul/div`，你选择的是哪一种设计方案？（友元/自由函数/命名空间+自由函数/类+static）

你的决策理由？如友元方案的优缺点、静态成员函数方案的适用场景、命名空间方案的考虑因素等。

五、实验总结(选)

此部分书写内容：（可以包括但不限于以下内容）

- 提炼本次实验的核心收获与思考
- 记录新发现、真实感受与待解疑问
- 任意想补充的反馈或分享

六、实验文档提交要求

- 提交形式

请将实验文档以**在线技术博客**形式提交，发布后请在**相应实验作业**页面提交链接。

博客开通与提交方法详见：「[博客园博客开通及发布步骤_2024级_OOP.pdf](#)」

- 撰写内容

必写：「四、实验结论」

选写：「五、实验总结」

要求：内容完整，表述简练，逻辑清晰，图文整洁

- 截止时间

通常为实验课后一周内，具体**截止时间**以博客园发布的作业通知为准。（**提交时间**以博客园文章页面显示的日

期为准)

截止后，欢迎通过[教学班级博客主页](#)相互评阅。

代码正确、风格规范、图文清晰、总结到位的博客，欢迎点赞推荐，一起进步！

七、博客园编辑器使用补充说明

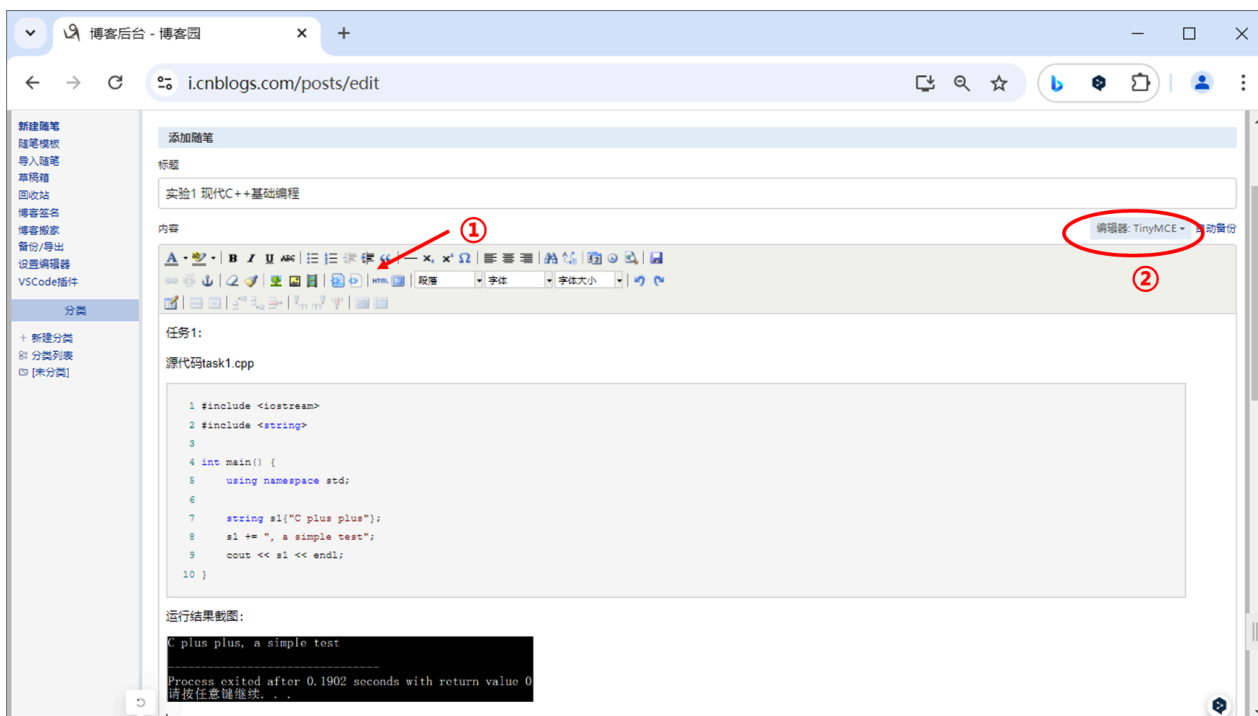
针对博客园平台使用的默认编辑器，插入代码和图片后，文字难以插入编辑的状况：

- 解决方案1

在插入代码和图片前，自己先编辑好文字，或者，用回车，预留好空白行

- 解决方案2

在编辑界面，点开页面对应的html源代码，在你想插入文字的地方加上html标签 `<p></p>`（图中①位置），然后再回到编辑界面，就会看到空白行。可以在那里编辑。



另，博客园平台设置页面支持更改默认编辑器（图中②位置）。目前，默认编辑器是markdown编辑器。

如果你暂不熟悉markdown基础用法，可切换成TinyMCE编辑器，其界面和使用风格接近office系列。

markdown编辑器使用参考： <https://www.runoob.com/markdown/md-tutorial.html>