



§ 14. C++知识补充

7. 类的共用数据的保护

7.1. 引入及含义

一个数据可以通过不同的方式进行共享访问，因此可能导致数据因为误操作而改变，为了达到既能共享，又不会因误操作而改变，引入共用数据保护的概念

```
void fun(int *p)
```

```
{ *p=10;  
}
```

```
int main()
```

```
{ int k=15;  
  fun(&k);  
}
```

通过**指针/引用**，在fun中
改变了main的局部变量k的值

```
void fun(int &p)
```

```
{ p=10;  
}
```

```
int main()
```

```
{ int k=15;  
  fun(k);  
}
```

§ 2. 基础知识

2.7. 变量

2.7.6. 常变量

含义：在程序执行过程中值不能改变的变量

形式：

```
const 数据类型 变量名=初值;
```

```
数据类型 const 变量名=初值;
```

```
const int a=10;
```

```
const double pi=3.14159;
```

★ 常变量必须在**定义时赋初值**，且在执行过程中**不能再次赋值**，否则编译错误

§ 12. 指针进阶

10. const指针

10.1. 共用数据的保护

10.2. 指向常量的指针变量

形式：const 数据类型 *指针变量名;

数据类型 const *指针变量名;

10.3. 常指针变量(常指针)

形式：数据类型 *const 指针变量名;

10.4. 指向常量的常指针变量(同时满足10.2+10.3)

形式：const 数据类型 *const 指针变量名;

	能否指向 不同变量	能否通过指针 变量修改变量值	是否需要定义时初始化
指向常量的指针变量	✓	✗	✗
常指针	✗	✓	✓
指向常量的常指针	✗	✗	✓



§ 14. C++知识补充

7. 类的共用数据的保护

7.2. 常对象

常对象:

const 类名 对象名(初始化实参表)

或 类名 const 对象名(初始化实参表)

const Time t1(15);

T1始终是15:00:00

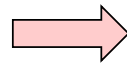
Time const t2(16, 30, 0);

T2始终是16:30:00

★ 与常变量相同, 在整个程序的执行过程中值不可再变化

★ 与常变量相同, 必须在定义时进行初始化

★ 不能调用普通成员函数 (即使不改变数据成员的值)



```
demo.cpp
demo-cpp
1  #include <iostream>
2  using namespace std;
3  class Time {
4  public:
5      int hour, minute, sec;
6      Time(int h = 0, int m = 0, int s = 0)
7      { hour = h; minute = m; sec = s; }
8      void display()
9      { cout << hour << minute << sec; }
10 };
11 int main()
12 {
13     const Time t1(15);
14     Time const t2(16, 30, 0);
15     t1.minute = 12; //编译报错
16     t2.sec = 27;    //编译报错
17     t1.display();  //编译报错(虽然不改)
18 }
```

```
1>demo.cpp
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(15,7): error C3892: "t1": 不能给常量赋值
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(16,7): error C3892: "t2": 不能给常量赋值
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(17,8): error C2662: "void Time::display(void)": 不能将"this"指针从"const Time"转换为"Time &"
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(17,5):
1> 转换丢失限定符
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(8,10):
1> 参见"Time::display"的声明
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(17,8):
1> 尝试匹配参数列表"()"时
1>已完成生成项目"demo-cpp.vcxproj"的操作 - 失败。
```



§ 14. C++知识补充

7. 类的共用数据的保护

7.3. 常对象成员

常对象成员：

常对象中的所有数据成员在程序执行过程中值均不可变，如果只需要限制部分成员的值在执行过程中不可变，则需要引入常对象成员的概念

- 常数据成员：该数据成员的值在执行中不可变
- 常成员函数：该函数只能引用成员的值，不能修改

常数据成员：

```
class 类名 {  
    const 数据类型 数据成员名  
    或 数据类型 const 数据成员名  
};  
  
class Time {  
    private:  
        const int hour;  
        int const minute;  
        int sec;  
};
```

常成员函数：

```
class 类名 {  
    返回类型 成员函数名(形参表) const;  
};  
  
class Time {  
    public:  
        void display() const;  
};
```



§ 14. C++知识补充

7. 类的共用数据的保护

7.3. 常对象成员

使用:

★ 常数据成员要在构造函数中初始化, 使用中值不变, 在构造函数中进行初始化时, 必须用参数初始化表

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

class Time {
public:
    const int hour, minute, sec;
    Time(int h=0, int m=0, int s=0)
    {
        hour    = h;
        minute  = m;
        sec     = s;
    }
};
```

错误

```
int main()
{
    Time t1;
    t1.hour = 10; //错误
}
```

```
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(7,9): error C2789: "Time::hour": 必须初始化常量限定类型的对象
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(6,15):
1>参见 "Time::hour" 的声明
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(7,9): error C2789: "Time::minute": 必须初始化常量限定类型的对象
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(6,21):
1>参见 "Time::minute" 的声明
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(7,9): error C2789: "Time::sec": 必须初始化常量限定类型的对象
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(6,29):
1>参见 "Time::sec" 的声明
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(9,9): error C2166: 左值指定 const 对象
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(10,9): error C2166: 左值指定 const 对象
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(11,9): error C2166: 左值指定 const 对象
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(18,7): error C3892: "t1": 不能给常量赋值
```

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

class Time {
public:
    const int hour;
    int minute, sec;
    Time(int h=0, int m=0, int s=0) : hour(h)
    {
        minute = m;
        sec    = s;
    }
};
```

正确

```
int main()
{
    Time t1;
    t1.hour = 10; //错误
}
```

```
demo.cpp(17,7): error C3892: "t1": 不能给常量赋值
```



§ 14. C++知识补充

7. 类的共用数据的保护

7.3. 常对象成员

使用:

★ 常数据成员要在构造函数中初始化, 使用中值不变, 在构造函数中进行初始化时, 必须用参数初始化表

★ 常成员函数只能引用类的数据成员 (无论是常数据成员) 的值, 而不能修改数据成员的值

普通成员函数	常成员函数
<pre>#include <iostream> using namespace std; class Time { public: int hour, minute, sec; void set(int h=0,int m=0,int s=0) { hour = h; minute = m; sec = s; } }; int main() { Time t1; t1.set(14, 15, 23); }</pre> <div data-bbox="488 933 598 1048">正确</div>	<pre>#include <iostream> using namespace std; class Time { public: int hour, minute, sec; void set(int h=0,int m=0,int s=0) const { hour = h; minute = m; sec = s; } }; int main() { Time t1; t1.set(14, 15, 23); }</pre> <div data-bbox="1214 933 1323 1031">错误</div> <div data-bbox="1214 1048 2027 1129"><p>demo.cpp(8,9): error C3490: 由于正在通过常里对象访问“hour”, 因此无法对其进行修改 demo.cpp(9,9): error C3490: 由于正在通过常里对象访问“minute”, 因此无法对其进行修改 demo.cpp(10,9): error C3490: 由于正在通过常里对象访问“sec”, 因此无法对其进行修改</p></div>



§ 14. C++知识补充

7. 类的共用数据的保护

7.3. 常对象成员

使用:

★ 常成员函数写成下面形式, 编译不报错但不起作用

const 返回类型 成员函数名(形参表)

或 返回类型 const 成员函数名(形参表)

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

赋值正确, 说明
set不是常成员函数

```
class Time {
public:
    int hour, minute, sec;
    const void set(int h, int m, int s)
    {   hour    = h;
        minute = m;
        sec     = s;
    }
};

int main()
{   Time t1;
    t1.set(14, 15, 23);
}
```

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

赋值正确, 说明
set不是常成员函数

```
class Time {
public:
    int hour, minute, sec;
    void const set(int h, int m, int s)
    {   hour    = h;
        minute = m;
        sec     = s;
    }
};

int main()
{   Time t1;
    t1.set(14, 15, 23);
}
```



§ 14. C++知识补充

7. 类的共用数据的保护

7.3. 常对象成员

使用:

★ 常成员函数可以调用本类的另一个常成员函数, 但不能调用本类的非常成员函数 (即使该非常成员函数不修改数据成员的值)

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

```
class Time {
public:
    int hour, minute, sec;
    void display()
    { cout << hour << endl;
    }
    void fun() const
    { display();
    }
```

错误

```
};
```

```
int main()
{
    Time t1;
    t1.fun();
}
```

```
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(9,9): error C2662: "void Time::display(void)": 不能将 "this" 指针从 "const Time" 转换为 "Time &"
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(9,9):
1> 转换丢失限定符
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(6,10):
1> 参见 "Time::display" 的声明
1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(9,9):
1> 尝试匹配参数列表 "()" 时
```

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

```
class Time {
public:
    int hour, minute, sec;
    void display() const
    { cout << hour << endl;
    }
    void fun() const
    { display();
    }
```

正确

```
};
```

```
int main()
{
    Time t1;
    t1.fun();
}
```



§ 14. C++知识补充

7. 类的共用数据的保护

7.3. 常对象成员

使用:

- ★ 若希望常成员函数能强制修改数据成员，则要将数据成员定义为mutable
(适用场景: 一个复杂大类，希望绝大多数成员函数不修改该值)

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

class Time {
public:
    int hour, minute, sec;
    void set(int h=0, int m=0, int s=0) const
    {
        hour = h;
        minute = m;
        sec = s;
    }
};

int main()
{
    Time t1;
    t1.set(14, 15, 23);
}
```

错误

demo.cpp(8,9): error C3490: 由于正在通过常里对象访问“hour”，因此无法对其进行修改
demo.cpp(9,9): error C3490: 由于正在通过常里对象访问“minute”，因此无法对其进行修改
demo.cpp(10,9): error C3490: 由于正在通过常里对象访问“sec”，因此无法对其进行修改

```
#include <iostream>
using namespace std;
class Time {
public:
    mutable int hour, minute, sec;
    void set(int h=0, int m=0, int s=0) const
    {
        hour = h;
        minute = m;
        sec = s;
    }
};

int main()
{
    Time t1;
    t1.set(14, 15, 23);
}
```

正确

```
#include <iostream>
using namespace std;
class Time {
public:
    int mutable hour, minute, sec;
    void set(int h=0, int m=0, int s=0) const
    {
        hour = h;
        minute = m;
        sec = s;
    }
};

int main()
{
    Time t1;
    t1.set(14, 15, 23);
}
```

正确



§ 14. C++知识补充

7. 类的共用数据的保护

7.3. 常对象成员

使用:

★ 若定义对象为常对象, 则只能调用其中的常成员函数 (不能修改数据成员的值), 而不能调用其中的普通成员函数 (即使该成员不修改数据成员的值)

本节开始的例子

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

class Time {
public:
    int hour, minute, sec;
    Time(int h=0, int m=0, int s=0) { hour = h; minute = m; sec = s; }
    void display() { cout << hour << minute << sec; }
};

int main()
{
    const Time t1(15);
    Time const t2(16, 30, 0);
    t1.minute = 12; //编译错误
    t2.sec = 27;    //编译错误
    t1.display();  //编译错误
}
```

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

class Time {
public:
    int hour, minute, sec;
    Time(int h=0, int m=0, int s=0) { hour = h; minute = m; sec = s; }
    void display() const { cout << hour << minute << sec; }
};

int main()
{
    const Time t1(15);
    Time const t2(16, 30, 0);
    t1.minute = 12; //编译错误
    t2.sec = 27;    //编译错误
    t1.display();  //编译正确
}
```



§ 14. C++知识补充

7. 类的共用数据的保护

7.3. 常对象成员

使用:

★ 不能定义构造/析构函数为常成员函数(常识性问题)

★ 全局函数不能定义const

```
void fun() const //编译报错
{
    return;
}
int main()
{
    fun();
}
```

```
demo.cpp (全局范围)
1  #include <iostream>
2  using namespace std;
3
4  void fun() const //编译报错
5  {
6      return;
7  }
8  int main()
9  {
10     fun();
11 }
```

demo.cpp(5,1): error C2270: "fun": 非成员函数上不允许修饰符

	普通数据成员	const数据成员	mutable数据成员	普通成员函数	const成员函数
普通对象	读写	读	读写	可调用	可调用
const对象	读	读	读写	不可调用	可调用
普通成员函数	读写	读	读写	可调用	可调用
const成员函数	读	读	读写	不能调用	可调用



§ 14. C++知识补充

7. 类的共用数据的保护

7. 4. 指向常对象的指针变量

⇔ 指向常量的指针变量

7. 5. 指向对象的常指针

⇔ 常指针

7. 6. 指向常对象的常指针

⇔ 指向常量的常指针

§ 12. 指针进阶

10. const指针

10. 1. 共用数据的保护

10. 2. 指向常量的指针变量

形式: `const 数据类型 *指针变量名;`

`数据类型 const *指针变量名;`

10. 3. 常指针变量(常指针)

形式: `数据类型 *const 指针变量名;`

10. 4. 指向常量的常指针变量(同时满足10. 2+10. 3)

形式: `const 数据类型 *const 指针变量名;`

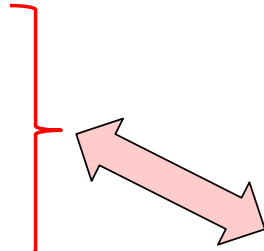


§ 14. C++知识补充

7. 类的共用数据的保护

7.7. 指向变量/对象的常引用

```
const int k;  
int const &ref2 = k;  
  
const Time t1;  
const Time &ref1 = t1;
```



- ① 指向常对象的指针变量
⇔ 指向常量的指针变量
- ② 指向对象的常指针
⇔ 常指针
- ③ 指向常对象的常指针
⇔ 指向常量的常指针

★ 常引用与常指针的使用基本类似

第06模块:

★ 引用必须在声明时进行初始化, 指向同类型变量, 整个生存期内不能再指向其它变量

=> 普通引用已符合②且不可能是①



§ 14. C++知识补充

8. 类的静态成员

8.1. 引入

希望在同一个类的多个对象间实现数据共享

★ 一个对象修改，另一个对象访问得到修改后的值

★ 类似与全局变量的概念，但属于类，仅供该类的不同对象间共享数据

8.2. 静态数据成员

定义：class 类名 {

private/public:

static 数据类型 成员名;

...

};



§ 14. C++知识补充

8. 类的静态成员

8.2. 静态数据成员

使用:

- ★ 静态数据成员不属于任何一个对象，不在对象中占用空间，单独在静态数据区分配空间(初值为0，不随对象的释放而释放)，一个静态数据成员只占有一个空间，所有对象均可共享访问
- ★ 静态数据成员同样受类的作用域限制
- ★ 静态数据成员必须进行初始化，初始化位置在类定义体后，函数体外进行(此时不受类的作用域限制)
数据类型 类名::静态数据成员名=初值;
- ★ 虽然可以通过this指针访问，但是数据不在一起

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

```
int a=10;
class Time {
private:
    static int hour;
    int minute, sec;
public:
    Time(int h=0, int m=0, int s=0) //未对hour赋值
    { minute = m;
      sec = s;
    }
    void display() //打印三个成员的地址，目的?
    { cout << hour << ':' << minute << ':' << sec << endl;
      cout << &hour << endl;
      cout << &minute << endl;
      cout << &sec << endl;
    }
};
```

将display函数中的成员访问
均加上this->，观察结果
(例: this->hour/&this->minute)

```
int Time::hour = 5; //虽然是private，可以
```

```
int main()
{ Time t1;
  cout << sizeof(t1) << endl;
  cout << &a << endl; //目的?
  t1.display();
}
```

8
a的地址
5:0:0
hour的地址(与a相邻，与下面两个差别很大)
minute的地址
sec的地址

Microsoft Visual Studio 调试控制台

```
8
0038C000
5:0:0
0038C004
00D6FDE0
00D6FDE4
```



§ 14. C++知识补充

8. 类的静态成员

8.2. 静态数据成员

使用:

★ 不能通过参数初始化表进行初始化, 但可以通过赋值方式初始化

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

```
class Time {
private:
    static int hour;
    int minute, sec;
public:
    Time(int h=0, int m=0, int s=0) : hour(h)
    {
        minute = m; sec = s;
    }
    void display()
    {
        cout << hour << minute << sec << endl;
    }
};
```

error C2438: "hour": 无法通过构造函数初始化静态类数据

错误

int Time::hour = 5; //必须要有, 否则编译错

```
int main()
{
    Time t1;
    cout << sizeof(Time) << endl;
    t1.display();
}
```

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

```
class Time {
private:
    static int hour;
    int minute, sec;
public:
    Time(int h=0, int m=0, int s=0)
    {
        hour = h; minute = m; sec = s;
    }
    void display()
    {
        cout << hour << minute << sec << endl;
    }
};
```

问1: hour初始化为5, 在构造函数中又被赋值为h, 最终是几?

问2: 哪个先执行?为什么?

正确

int Time::hour = 5; //必须要有, 否则编译错

```
int main()
{
    Time t1;
    cout << sizeof(Time) << endl;
    t1.display();
}
```

Microsoft Visual Studio 调试控制台

```
8
0:0:0
```



§ 14. C++知识补充

8. 类的静态成员

8.2. 静态数据成员

使用:

★ 不能通过参数初始化表进行初始化, 但可以通过赋值方式初始化

★ 既可以通过类型引用, 也可以通过对象名引用

★ 结论: 静态数据成员不是面向对象的概念, 它破坏了数据的封装性, 但方便使用, 提高了运行效率

Microsoft Visual Studio 调试控制台

```
0:15:23
0:0:0
8:15:23
8:0:0
19:15:23
19:0:0
```

```
int main()
{
    Time t1(14, 15, 23), t2;
    t1.display(); 0:15:23
    t2.display(); 0:0:0
    t1.hour = 8; //对象名. 成员名
    t1.display(); 8:15:23
    t2.display(); 8:0:0
    Time::hour = 19; //类名::成员名
    t1.display(); 19:15:23
    t2.display(); 19:0:0

    return 0;
}
```

为什么既不是14:15:23, 也不是10:15:23?

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

class Time {
    public: //不符合规范, 为了在main中直接访问
        static int hour;
        int minute;
        int sec;
    public:
        Time();
        Time(int h, int m, int s);
        void display();
};

int Time::hour = 10;

Time::Time()
{
    hour=0; minute=0; sec=0;
}

Time::Time(int h, int m, int s)
{
    hour=h; minute=m; sec=s;
}

void Time::display()
{
    cout << hour << ":" << minute << ":" << sec << endl;
}
```




§ 14. C++知识补充

8. 类的静态成员

8.3. 静态成员函数

定义: class 类名 {

private/public:

static 返回类型 函数名(形参表);

...

};

调用:

类名::成员函数名(实参表);

任意对象名.成员函数名(实参表);

使用:

- ★ 允许体内实现或体外实现
- ★ 静态成员函数中可以直接访问静态数据成员
- ★ 既可以“对象.静态成员函数名()”形式访问静态成员函数
也可以用“类名::静态成员函数名()”形式访问静态成员函数
- ★ 没有this指针, 不属于某个对象, 不能直接访问普通数据成员
普通成员函数:

Time t1;

t1.display() ⇔ t1.display(&t1);

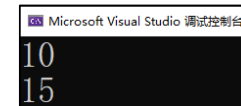
静态成员函数:

无

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

class test {
public:
    static void fun(int x)
    {
        cout << x << endl;
    }
};

int main()
{
    test t1;
    t1.fun(10); //10
    test::fun(15); //15
}
```





§ 14. C++知识补充

8. 类的静态成员

8.3. 静态成员函数

使用:

- ★ 允许体内实现或体外实现
- ★ 静态成员函数中可以直接访问静态数据成员
- ★ 既可以“对象.静态成员函数名()”形式访问静态成员函数,也可以用“类名::静态成员函数名()”形式访问静态成员函数

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

体内实现

```
class test {
private:
    static int a;
public:
    static void fun()
    { cout << a << endl;
    };
};
int test::a = 10;
int main()
{ test t1;
  t1.fun(); //10
  test::fun(); //10
}
```

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

```
class test {
private:
    static int a;
public:
    static void fun()
    { cout << this->a << endl;
    };
};
int test::a = 10;
int main()
{ test t1;
  t1.fun();
  test::fun();
}
```

静态函数没有this指针,
显式加上后会报错!!!

error C2355: “this”: 只能在非静态成员函数或非静态数据成员初始值设定项的内部引用

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

体外实现

```
class test {
private:
    static int a;
public:
    static void fun();
};
int test::a = 10;
void test::fun() //此处不能static
{ cout << a << endl;
}
int main()
{ test t1;
  t1.fun(); //10
  test::fun(); //10
}
```



§ 14. C++知识补充

8. 类的静态成员

8.3. 静态成员函数

使用:

★ 没有this指针, 不属于某个对象, 不能直接访问普通数据成员

推论 => 在静态成员函数中不能对非静态数据成员进行直接访问, 而要通过对象参数的方式

(不提倡, 建议静态成员函数只访问静态数据成员)

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

```
class test {
private:
    static int a;
    int b;
public:
    static void fun()
    {
        a=10; //正确
        b=11; //错误, 因为没有this指针, 不知道
              //应该访问那个对象的b成员
    }
};
```

```
int test::a = 5;
```

```
int main()
{
    test t1;
    t1.fun();
    test::fun();
}
```

demo.cpp(12,9): error C2597: 对非静态成员“test::b”的非法引用
demo.cpp(7): message : 参见“test::b”的声明

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

```
class test {
private:
    static int a;
    int b;
public:
    static void fun(test &t)
    {
        a=10; //正确
        t.b=11; //正确
    }
};
```

```
int test::a = 5;
```

```
int main()
{
    test t1, t2;
    t1.fun(t1);
    test::fun(t2);
}
```

不提倡, 建议静态成员函数
只访问静态数据成员



§ 14. C++知识补充

9. 类模板

9.1. 函数模板

函数重载的不足：对于参数个数**相同**，类型**不同**，而实现过程**完全相同**的函数，仍要分别给出各个函数的实现

```
int max(int x, int y)
{
    return x>y?x:y;
}
double max(double x, double y)
{
    return x>y?x:y;
}
```

问题：两段一样的代码
能否合并为一段？

函数模板：建立一个通用函数，其返回类型及参数类型**不具体指定**，用一个**虚拟类型**来代替，该通用函数称为**函数模板**，调用时再根据不同的实参类型来取代模板中的虚拟类型，从而实现不同的功能

```
#include <iostream>
using namespace std;
template <typename T> //虚类型T
T my_max(T x, T y)
{
    cout << sizeof(x) << ' ';
    return x > y ? x : y;
}
int main()
{
    int a=10, b=15;
    double f1=12.34, f2=23.45;
    cout << my_max(a, b) << endl;
    cout << my_max(f1, f2) << endl;
    return 0;
}
```

一段代码, 两个功能
1、两个int型求max
2、两个double型求max

问题1：如果传入两个unsigned int型数据，
T的类型被实例化为什么？如何证明？
问题2：如果x, y的类型不同，自行摸索转换规律

```
Microsoft Visual Studio 调试控制台
4 15
8 23.45
```



§ 14. C++知识补充

9. 类模板

9.1. 函数模板

使用:

★ 仅适用于参数个数相同、类型不同，实现过程完全相同的情况

★ typename可用class替代

★ 类型定义允许多个，调用时可匹配不同的实参类型

```
template <typename T1, typename T2>
```

```
template <class T1, class T2>
```

```
#include <iostream>
using namespace std;
template <class T1, class T2>
char my_max(T1 x, T2 y)
{
    cout << sizeof(x) << ' ';
    cout << sizeof(y) << ' ';
    return x>y ? 'A' : 'a';
}
int main()
{
    int    a = 10, b = 15;
    double f1 = 12.34, f2 = 23.45;
    cout << my_max(a, f1) << endl;
    cout << my_max(f2, b) << endl;
    return 0;
}
```

```
Microsoft Visual Studio 调试控制台
4 8 a
8 4 A
```

问: max(a, f1)时, T1/T2类型分别是?
max(f2, b)时, T1/T2类型分别是?



§ 14. C++知识补充

9. 类模板

9.2. 类模板

引入：多个类，功能及实现完全相同，仅数据类型不同

```
class compare_int {  
private:  
    int x, y;  
public:  
    compare_int(int a, int b)  
        { x=a; y=b; }  
    int max()  
        { return x>y?x:y; }  
    int min()  
        { return x<y?x:y; }  
};
```

```
class compare_float {  
private:  
    float x, y;  
public:  
    compare_float(float a, float b)  
        { x=a; y=b; }  
    float max()  
        { return x>y?x:y; }  
    float min()  
        { return x<y?x:y; }  
};
```

合并为一个类型
为虚类型T的类

```
#include <iostream>  
using namespace std;  
  
template <class T>  
class compare {  
private:  
    T x, y;  
public:  
    compare(T a, T b)  
        { x=a; y=b; }  
    T max()  
        { return x > y ? x : y; }  
    T min()  
        { return x < y ? x : y; }  
};  
  
int main()  
{  
    compare <int>    c1(10,15);  
    compare <float> c2(10.1f, 15.2f);  
    cout << c1.max() << endl;  
    cout << c2.min() << endl;  
}
```

Microsoft Visual Studio 调试控制台
15
10.1



§ 14. C++知识补充

9. 类模板

9.2. 类模板

使用:

★ 类模板使用的多个类需要满足下面的条件

- 数据成员个数**相同**, 类型**不同**
- 成员函数个数**相同**, 类型**不同**, 实现过程**完全相同**

=> 推论: 如果两个类大部分相同, 可以通过定义不需要的数据成员及成员函数来达到使用类模板的目的(按实际情况决定)

```
class name_1 {  
    private:  
        int x, y, z;  
    public:  
        int f1(); //实际不需要  
        void f2(int);  
        void f3(int);  
        void f4(int);  
};
```

```
class name_1 {  
    private:  
        float x, y, z; //z实际不需要  
    public:  
        float f1();  
        void f2(float);  
        void f3(float);  
        void f4(float);  
};
```



§ 14. C++知识补充

9. 类模板

9.2. 类模板

使用:

- ★ 类模板使用的多个类需要满足下面的条件
- ★ 类模板可以看作是类的抽象, 称为参数化的类
- ★ 类模板成员函数体外实现时形式有所不同

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

template <class T>
class compare {
private:
    T x, y;
public:
    compare(T a, T b)
        { x=a; y=b; }
    T max()
        { return x>y?x:y; }
    T min()
        { return x<y?x:y; }
};

int main()
{
    compare <int> c1(10,15);
    compare <float> c2(10.1f, 15.2f);
    cout << c1.max() << endl;
    cout << c2.min() << endl;
}
```

体内实现

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

template <class T>
class compare {
private:
    T x, y;
public:
    compare(T a, T b);
    T max();
    T min();
};

template <class T> //每个体外实现的函数前都要
compare<T>::compare(T a, T b)
{
    x=a;
    y=b;
}

template <class T> //每个体外实现的函数前都要
T compare<T>::max()
{
    return x>y?x:y;
}

template <class T> //每个体外实现的函数前都要
T compare<T>::min()
{
    return x<y?x:y;
}

int main()
{
    compare <int> c1(10,15);
    compare <float> c2(10.1f, 15.2f);
    cout << c1.max() << endl;
    cout << c2.min() << endl;
}
```

体外实现

普通类构造函数的体外实现:

```
test::test(int x, int y)
{
    ...
}
```

普通类成员函数的体外实现:

```
int test::fun(int x, int y)
{
    ...
}
```




§ 14. C++知识补充

9. 类模板

9.2. 类模板

使用:

★ 类模板使用的多个类需要满足下面的条件

- 数据成员个数**相同**, 类型**不同**
- 成员函数个数**相同**, 类型**不同**, 实现过程**完全相同**

=> 推论: 如果两个类大部分相同, 可以通过定义不需要的数据成员及成员函数来达到使用类模板的目的(按实际情况决定)

★ 类模板可以看作是类的抽象, 称为参数化的类

★ 类模板成员函数体外实现时形式有所不同

★ 类型定义允许多个

template <class T1, class T2>



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

template <class T1, class T2>
class test {
    private:
        T1 x;
        T2 y;
    public:
        test(T1 a, T2 b) {
            x=a;
            y=b;
        }
};

int main()
{
    test <int, float> c1(10, 15.1);
    test <int, int> c2(10, 15);
}
```



§ 14. C++知识补充

10. 枚举类

10.1. C方式传统枚举存在的问题

- ★ 不同enum的枚举常量值不能相同，但实际应用中**有此需求**
- ★ 不同enum的变量/常量**不应该**允许比较，但编译运行正确(无法解释语义)
- ★ 不同enum的变量/常量均可以直接整型输出，容易造成误解
- ★ C方式，enum可以直接用整型赋值，不检查范围
- ★ C++方式，enum不能直接整型赋值，需加强制类型转换，不检查范围



§ 14. C++知识补充

10. 枚举类

10.1. C方式传统枚举存在的问题

★ 不同enum的枚举常量值不能相同，但实际应用中**有此需求**

```
#include <iostream>
using namespace std;
enum week { sun, mon, tue, wed, thu, fri, sat };
enum candidates { zhao, qian, sun, li };
int main()
{
    cout << wed << endl;
    return 0;
}
```

1>D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(4,31): error C2365: “sun”: 重定义; 以前的定义是“枚举数”
1> D:\Workspace\VS2022-demo\demo-cpp\demo.cpp(3,13):
1> 参见“sun”的声明

★ 不同enum的变量/常量均可以直接整型输出，容易造成误解

```
#include <iostream>
using namespace std;
enum week { sun, mon, tue };
enum candidates { zhao, qian, li };
int main()
{
    cout << sun << ' ' << li << endl;
    printf("%d %d\n", sun, li);
    return 0;
}
```

Microsoft Visual Studio 调试控制台
0 2
0 2

★ 不同enum的变量/常量**不应该**允许比较，但编译运行正确(无法解释语义)

```
#include <iostream>
using namespace std;
enum week { sun, mon, tue };
enum candidates { zhao, qian, li };
int main()
{
    enum week w1=mon;
    enum candidates c1=qian;
    cout << (sun < li) << ' ' << (w1==c1)<< endl;
}
```

Microsoft Visual Studio 调试控制台
1 1



§ 14. C++知识补充

10. 枚举类

10.1. C方式传统枚举存在的问题

★ C方式, enum可以直接用整型赋值, 不检查范围

★ C++方式, enum不能直接整型赋值, 需加强制类型转换, 不检查范围

```
#include <stdio.h>
```

注意: 源程序必须.c形式

```
enum week { sun, mon, tue };
int main()
{
    enum week w1 = mon, w2 = 0, w3 = 4; //超定义范围
    printf("%d %d %d\n", w1, w2, w3);
    return 0;
}
```

Microsoft Visual Studio 调试控制台
1 0 4

```
#include <iostream>
```

cpp: 直接整型, 报错

```
using namespace std;
enum week { sun, mon, tue };
int main()
{
    enum week w1 = mon, w2 = 0, w3 = 4; //超定义范围
    cout << w1 << ' ' << w2 << ' ' << w3 << endl;
    return 0;
}
```

demo.cpp(6,31): error C2440: “初始化”: 无法从“int”转换为“week”
demo.cpp(6,28): message : 转换为枚举类型需要显式强制转换(static_cast<C 样式强制转换或带圆括号函数样式强制转换>)
demo.cpp(6,39): error C2440: “初始化”: 无法从“int”转换为“week”
demo.cpp(6,36): message : 转换为枚举类型需要显式强制转换(static_cast<C 样式强制转换或带圆括号函数样式强制转换>)

```
#include <iostream>
```

cpp: 强制类型转换, 不检查范围

```
using namespace std;
enum week { sun, mon, tue };
int main()
{
    enum week w1 = mon, w2 = week(0), w3 = week(4);
    cout << w1 << ' ' << w2 << ' ' << w3 << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台
1 0 4



§ 14. C++知识补充

10. 枚举类

10.2. 枚举类的定义与使用

定义:

```
enum class 枚举类型 { 枚举常量1, ..., 枚举常量n }
```

使用:

类名::常量值

- ★ 枚举类常量为整型值，从0开始（同enum）
- ★ 枚举类常量不能直接输出(enum可直接输出为整型)
- ★ 其余使用方法基本同enum

注意:

- ★ 不同enum class可定义相同的枚举常量标识符(常量值可不同)
- ★ 不同enum class之间的常量值不允许直接比较
- ★ enum class的枚举变量/常量不允许直接输出，需要加强制类型转换
- ★ C++方式，enum不能直接整型赋值，需加强制类型转换，不检查范围(同enum)



§ 14. C++知识补充

10. 枚举类

10.2. 枚举类的定义与使用

★ 不同enum class可定义相同的枚举常量标识符(常量值可不同)

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

enum class week { sun, mon, tue, wed, thu, fri, sat };
enum class candidates { zhao, qian, sun, li };

int main()
{
    return 0;
}
```

编译正确

★ 不同enum class之间的常量值不允许直接比较

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

enum class week { sun, mon, tue };
enum class candidates { zhao, qian, li };

int main()
{
    cout << (week::sun < candidates::li) << endl;
    return 0;
}
```

编译报错

error C2676: 二进制“<”: “week” 未定义该运算符或到预定义运算符可接收的类型的转换



§ 14. C++知识补充

10. 枚举类

10.2. 枚举类的定义与使用

★ enum class的枚举变量/常量不允许直接输出，需要加强制类型转换

```
#include <iostream>
using namespace std;
enum class week { sun, mon, tue, wed, thu, fri, sat };
int main()
{
    cout << wed << endl;
    printf("%d\n", mon);
    return 0;
}
```

enum class: 直接写常量值,
报未声明错

demo.cpp(6,13): error C2065: “wed”: 未声明的标识符
demo.cpp(7,20): error C2065: “mon”: 未声明的标识符

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

enum class: 写类名::常量值 方式
cout直接输出报错

```
enum class week { sun, mon, tue, wed, thu, fri, sat };
int main()
{
    cout << week::wed << endl;
    return 0;
}
```

demo.cpp(6,23): error C2679: 二元“<<”: 没有找到接受“week”类型的右操作数的运算符(或没有可接受的转换)

```
#include <iostream>
using namespace std;
enum class week { sun, mon, tue, wed, thu, fri, sat };
int main()
{
    cout << int(week::wed) << endl;
    printf("%d\n", week::mon);
    return 0;
}
```

enum class: 写类名::常量值 方式
cout需强转后输出整型
printf直接输出整型值

Microsoft Visual Studio 调试控制台

```
3
1
```



§ 14. C++知识补充

10. 枚举类

10.2. 枚举类的定义与使用

★ C++方式，enum不能直接整型赋值，需加强制类型转换，不检查范围(同enum)

```
#include <iostream>
using namespace std;
enum class week { sun, mon, tue };
int main()
{
    week w1 = week::mon, w2 = week(0), w3 = week(4);
    printf("%d %d %d\n", w1, w2, w3);
    return 0;
}
```

Microsoft Visual Studio 调试控制台

1 0 4



§ 14. C++知识补充

10. 枚举类

10.3. enum与enum class的比较

- ★ 不同enum的枚举常量值不能相同，但实际应用中**有此需求**
=> **enum class允许相同**
- ★ 不同enum的变量/常量**不应该**允许比较，但编译运行正确(无法解释语义)
=> **enum class不允许比较**
- ★ 不同enum的变量/常量均可以直接整型输出，容易造成误解
=> **enum class的cout不允许直接输出(加强制类型转换)，printf允许**
- ★ C方式，enum可以直接用整型赋值，不检查范围
=> **enum class不支持C方式**
- ★ C++方式，enum不能直接整型赋值，需加强制类型转换，不检查范围
=> **enum class同**



§ 14. C和C++知识点补充

11. 空指针nullptr

11.1. NULL的不足与隐患

★ NULL是#define NULL 0，本质是一个整数，在C语言中用来表示0地址

★ int *p = NULL/ if (p==NULL)之类的语句，实际上是将NULL做了隐式转换，因此当整型与指针同时出现时，可能会出现非预期结果

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

int main()
{
    int k = 0, *p = NULL;

    cout << typeid(NULL).name() << endl;
    if (k == NULL)
        cout << "int" << endl;
    if (p == NULL)
        cout << "int *" << endl;

    return 0;
}
```

```
Microsoft Visual Studio 调试控制台
int
int
int *
```

```
#include <iostream>
using namespace std;
void fun(int x)
{
    cout << "int : " << x << endl;
}
void fun(int *p)
{
    cout << "int* : " << p << endl;
}
int main()
{
    int k = 10, *p = &k;
    fun(k);
    fun(p);
    fun(0);
    fun(NULL); //匹配int而不是int*
    return 0;
}
```

```
Microsoft Visual Studio 调试控制台
int : 10
int* : 00AFFB50
int : 0
int : 0
```



§ 14. C和C++知识点补充

11. 空指针nullptr

11.1. NULL的不足与隐患

11.2. nullptr的引入

★ nullptr（不加using namespace std下是std::nullptr）是C++引入的专门表示空指针的常量

★ 使用时，nullptr可以显示转换，但不允许隐式转换，因此安全性和可读性更高

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

int main()
{
    int k = 0, *p = nullptr;

    cout << typeid(nullptr).name() << endl;
    if (k == nullptr)
        cout << "int" << endl;
    if (p == nullptr)
        cout << "int *" << endl;

    return 0;
}
```


demo.cpp(9,11): error C2446: “=”: 没有从“nullptr”到“int”的转换

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

int main()
{
    int k = 0, *p = nullptr;

    cout << typeid(nullptr).name() << endl;
    // if (k == nullptr)
    // cout << "int" << endl;
    if (p == nullptr)
        cout << "int *" << endl;

    return 0;
}
```




```
Microsoft Visual Studio 调试控制台
std::nullptr_t
int *
```

```
#include <iostream>
using namespace std;
void fun(int x)
{
    cout << "int : " << x << endl;
}

void fun(int *p)
{
    cout << "int* : " << p << endl;
}

int main()
{
    int k = 10, *p = &k;
    fun(k);
    fun(p);
    fun(0);
    fun(nullptr);
    return 0;
}
```



```
Microsoft Visual Studio 调试控制台
int : 10
int* : 0093FE94
int : 0
int* : 00000000
```



§ 14. C和C++知识点补充

12. 程序优化

算法优化

概念：编程者通过不同的代码实现方法来提升程序的效率（略）

编译器优化

概念：编译器通过**数学上等价的方法**将编程者的代码转换为另一种高效的实现，从而提升程序的效率

★ 编译器的编程者对计算机系统的了解**远远高于**普通编程者

CPU优化

概念：CPU硬件通过**预测、预读**的各种手段将后续指令的数据提前准备好，从而提升程序的执行效率

★ 硬件是程序运行最底层的支撑

§ 14. C和C++知识点补充

12. 程序优化

★ 编译器优化实例：观察下列两种实现 2^5 的代码实现

- 正常编译每次会带时间戳、符号表、调试信息，同一程序多次编译的可执行文件不一致
- Linux用`strip`命令可以去除这些信息，Windows下的自行研究

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

int main()
{
    cout << (2<<5) << endl;

    return 0;
}
```

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

int main()
{
    cout << (2*2*2*2*2) << endl;

    return 0;
}
```

```
[u1234567@oop ~]$ cat t1.cpp
#include <iostream>
using namespace std;
```

```
int main()
{
    cout << (2<<5) << endl;
    return 0;
}
```

```
[u1234567@oop ~]$ g++ -Wall -o t1 t1.cpp
[u1234567@oop ~]$ ./t1
64
```

```
[u1234567@oop ~]$ cat t2.cpp
#include <iostream>
using namespace std;
```

```
int main()
{
    cout << (2*2*2*2*2) << endl;
    return 0;
}
```

```
[u1234567@oop ~]$ g++ -Wall -o t2 t2.cpp
[u1234567@oop ~]$ ./t2
64
```

```
[u1234567@oop ~]$ diff -s t1 t2
二进制文件 t1 和 t2 不同
```

```
[u1234567@oop ~]$ strip t1 t2
[u1234567@oop ~]$ diff -s t1 t2
檔案 t1 和 t2 相同
```

§ 14. C和C++知识点补充

12. 程序优化

★ CPU优化实例：观察不同数据序列的数据进行冒泡排序的运行时间

- 正序数据
- 逆序数据
- 乱序数据

问题：乱序数据的比较及交换次数均少于逆序数据，为什么执行速度反而慢于逆序数据？CPU做了什么优化？

x86服务器 (Xeon E5)

```
[root@MyVM-x64 t]# time ./bubble < a.txt
比较次数 : 99999 交换次数 : 0

real    0m0.025s      100000个正序数据
user    0m0.023s
sys     0m0.002s

[root@MyVM-x64 t]# time ./bubble < b.txt
比较次数 : 4999950000 交换次数 : 4999950000

real    0m18.986s      100000个逆序数据
user    0m18.933s
sys     0m0.001s

[root@MyVM-x64 t]# time ./bubble < c.txt
比较次数 : 4999934600 交换次数 : 2498226345

real    0m28.617s      100000个乱序数据
user    0m28.534s
sys     0m0.001s
[root@MyVM-x64 t]#
```

arm服务器

```
[u1234567@oop ~]$ time ./bubble < a.txt
比较次数 : 99999 交换次数 : 0

real    0m0.039s      100000个正序数据
user    0m0.029s
sys     0m0.010s

[u1234567@oop ~]$ time ./bubble < b.txt
比较次数 : 4999950000 交换次数 : 4999950000

real    0m35.851s      100000个逆序数据
user    0m35.850s
sys     0m0.000s

[u1234567@oop ~]$ time ./bubble < c.txt
比较次数 : 4999934600 交换次数 : 2498226345

real    0m43.821s      100000个乱序数据
user    0m43.820s
sys     0m0.000s
[u1234567@oop ~]$
```

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

#define N 100000

int main(int argc, char** argv)
{
    int a[N], i, j, t;
    long long count1 = 0, count2 = 0; //计数器

    /* 读入数据，不考虑输入错误 */
    for (i = 0; i < N; i++)
        cin >> a[i];

    /* 冒泡法 */
    bool exchange;
    for (i = 0; i < N - 1; i++) {
        exchange = false; //每次（外循环）前交换标记置false

        for (j = 0; j < N - (i + 1); j++) {
            count1++; //比较计数
            if (a[j] > a[j + 1]) {
                count2++; //交换计数
                exchange = true;
                t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
        }

        if (!exchange) //如果本次无交换则说明已经有序
            break;
    }

    #if 0 /* 排序完成后输出，按需打开条件编译 */
    for (i = 0; i < N; i++) {
        cout << a[i] << ' ';
        if (i % 10 == 9)
            cout << endl;
    }
    if (i % 10 != 9)
        cout << endl;
    #endif

    cerr << "比较次数 : " << count1
        << " 交换次数 : " << count2 << endl;
    return 0;
}
```