



中國人民大學
RENMIN UNIVERSITY OF CHINA

程序设计荣誉课程

3 数据的共享与保护

授课教师：孙亚辉

本章内容

1. 友元
2. 共享数据的保护
3. 命名空间

1. 友元

- 类中成员可被`public`、`private`和`protected`修饰，不同的修饰符代表着不同的访问权限。
- 一般来说，类外部的任何函数都不能访问被`private`修饰的成员。这种限制有时过于严格——在某些情况下，外部函数或类需要访问某个类A中所有成员，但又不想将类A中的所有成员都变为`public`或者为所有非公有成员添加外部访问接口。
- 可通过指定“**友元**（**friend**）”来允许外部函数或类访问某个类的非公有成员。**友元是C++提供的一种破坏数据封装和数据隐藏的机制。**

友元函数

- 某个类想把一个函数作为它的友元，只需要添加一条以关键字**friend**开始的函数声明语句即可。
 - 友元声明只能出现在类定义的内部，但是位置不受所在区域访问控制级别的约束（**private**、**public**、**protected**）。
 - 友元不是类的成员，类本身或类成员的可访问性不影响友元的可访问性。

```

1  #include <iostream>
2  using namespace std;
3
4  class Complex2; // claim this early, since Complex2 is used in Complex1
5  class Complex1
6  {
7  private:
8      int real, imag; //real:实部 imag:虚部
9  public:
10     Complex1(int r = 0, int i = 0)
11     {
12         real = r;
13         imag = i;
14     }
15     friend Complex1 operator+(Complex1 &c1, Complex2 &c2);
16     friend Complex1 operator+(int, Complex1 &c);
17 };
18 class Complex2
19 {
20 private:
21     int real, imag; //real:实部 imag:虚部
22 public:
23     Complex2(int r = 0, int i = 0)
24     {
25         real = r;
26         imag = i;
27     }
28     friend Complex1 operator+(Complex1 &c1, Complex2 &c2);
29 };
30
31 Complex1 operator+(Complex1 &c1, Complex2 &c2)
32 {
33     Complex1 ret(c1.real + c2.real, c1.imag + c2.imag);
34     return ret;
35 }
36 Complex1 operator+(int val, Complex1 &c)
37 {
38     Complex1 ret(c.real + val, c.imag);
39     return ret;
40 }
41
42 int main()
43 {
44     Complex1 a(1, 2);
45     Complex2 b(2, 3);
46     Complex1 c = a + b; //调用operator+(Complex1, Complex2)
47     1 + a;              //调用operator+(int, Complex1)
48 }

```

友元类

- 类B是类A的友元类，则B的成员函数能够访问A的所有成员
 - 语法：将友元类名在另一个类中使用`friend`修饰说明

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Logi;
6  class Person
7  {
8  private:
9      int age;
10     bool sex;
11     char name[21];
12     friend class Logi; // 友元类
13 public:
14     Person(int x, bool y, char *z)
15     {
16         age = x;
17         sex = y;
18         strcpy(name,z);
19     }
20 };
21
22 class Logi
23 {
24 public:
25     void log(Person &pn);
26 };
27 void Logi::log(Person &pn)
28 {
29     cout << "Person: " << pn.age
30         << " " << pn.sex << " "
31         << pn.name << endl;
32 }
33 int main()
34 {
35     Person a1(1,true,(char*)"You"); // (char*) is need for C++, not needed for C
36     Logi b;
37     b.log(a1);
38 }
```

打印结果： Person: 1 1 You

友元的性质

- 友元关系不传递
 - 你朋友的朋友不是你的朋友
 - `Person`的友元类是`Logi`，`Logi`的友元类是`OutputStream`，若`OutputStream`未在`Person`中声明，`OutputStream`不是`Person`的友元
- 友元关系不继承
 - 你朋友的孩子不是你的朋友
 - `Person`的友元类是`Logi`，而`XLogi`是`Logi`的派生类（子类），若`XLogi`未在`Person`中声明，`XLogi`不是`Person`的友元
 - 你的朋友不是你孩子的朋友
 - `Person`的友元类是`Logi`，`Teacher`是`Person`的派生类（子类），但`Logi`若未在`Teacher`中声明，`Logi`不是`Teacher`的友元

友元关系是单向、非传递、非继承的

- 友元关系不传递

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Logi;
6  class Person
7  {
8  private:
9      int age;
10     friend class Logi; // 友元类
11 public:
12     Person(int x){age = x;}
13 };
14
15 class Logi
16 {
17 public:
18     void log(Person &pn){cout << "Person: " << pn.age << endl;}
19     friend class Logi2; // 友元类
20 };
21
22 class Logi2
23 {
24 public:
25     void log(Person &pn){cout << "Person: " << pn.age << endl;}
26 };
27
28 int main()
29 {
30     Person a1(1); // (char*) is need for C++, not needed for C
31 }
```

yahui@Yahui:/media/sf_VM\$ g++ test.cpp

test.cpp: In member function 'void Logi2::log(Person&)':

test.cpp:25:51: error: 'int Person::age' is private within this context

25 | void log(Person &pn){cout << "Person: " << pn.age << endl;}
 | ^~~

test.cpp:9:9: note: declared private here

9 | int age;
 | ^~~

本章内容

1. 友元
2. 共享数据的保护
3. 命名空间

2. 共享数据的保护

- 除“友元”外，还可以通过其他两种方式实现数据的共享：
 - 将数据成员声明为public
 - 将数据成员声明为private或protected，但提供接口（setter/getter）来操作该数据
- 保护共享数据成员的一种简单方法
 - 数据成员对外部可见但不可改：private数据成员，提供public的getter可访问该数据成员，但不提供setter来修改该数据成员

打印结果：11

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Person
6  {
7  private:
8      int age;
9
10 public:
11     Person(int x) : age(x) {}
12     int age_getter() const
13     {
14         return age;
15     }
16     void age_setter(int new_age)
17     {
18         age = new_age;
19     }
20 };
21
22 int main()
23 {
24     Person a(10);
25     a.age_setter(11);
26     cout << a.age_getter() << endl;
27 }
```

常类型（const）

但有时我们可能希望在某些时刻用户对public数据成员不具有修改的权限：

- 对于既需要共享、又要防止改变的数据应该声明为**常类型**（用const进行修饰）
 - 常成员：用const进行修饰的类成员——常数据成员和常函数成员
 - 常对象：必须进行初始化，不能被更新
const 类名 对象名
 - 常引用：被引用的对象不能被更新
const 类型说明符 &引用名
 - 常指针：指向常量的指针

常数据成员

- 常数据成员
 - 使用`const`修饰的数据成员
 - 任何函数中都不能对该成员赋值
 - 构造函数**只能通过初始化列表对该数据成员进行初始化**

```
1. class A {  
2.     public:  
3.         A(int i, int j) : x(i), y(j){}  
4.     private:  
5.         const int x, y;  
6. };
```



```
1. class A {  
2.     public:  
3.         A(int i, int j) { x = i; h = j; }  
4.     private:  
5.         const int x, y;  
6. };
```



常成员函数

- 常成员函数
 - 使用`const`修饰的成员函数
 - 常成员函数不更新对象的数据成员（不对数据成员进行赋值）
 - 格式：返回类型说明符 函数名(参数表) `const`;
 - `const`关键字可以被用于参与对重载函数的区分

```
5  class Person
6  {
7  private:
8      int age;
9
10 public:
11     int public_age = 18;
12     Person(int x) : age(x) {}
13     int age_getter()
14     {
15         return age;
16     }
17     int age_getter() const // const关键字可以被用于参与对重载函数的区分
18     {
19         return age;
20     }
21 };
```

```

1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Person
6  {
7  private:
8      int age;
9
10 public:
11     int public_age = 18;
12     Person(int x) : age(x) {}
13     int age_getter()
14     {
15         return age;
16     }
17     int age_getter() const // const关键字可以被用于参与对重载函数的区分
18     {
19         age++;
20         return age;
21     }
22 };
23
24 int main()
25 {
26     const Person a(10);
27     cout << a.age_getter() << endl;
28 }

```

yahui@Yahui:/media/sf_VM\$ g++ test.cpp

test.cpp: In member function 'int Person::age_getter() const':

test.cpp:19:9: error: increment of member 'Person::age' in read-only object

```

19 |         age++;
   |         ^~~

```

常对象

- 常对象示例

```
1. class A {  
2.     public:  
3.         A(int i, int j) : x(i), y(j){}  
4.     private:  
5.         int x, y;  
6. };  
  
7. int main() {  
8.     A const a(3, 4);  
9.     const A b(3, 4);  
10.    return 0;  
11.}
```

常对象的数据成员被视为常量，只能（且必须）被初始化，不能被赋值

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Person
6  {
7  private:
8
9  public:
10     int age = 18;
11     Person(int x) : age(x) {}
12     int age_getter()
13     {
14         return age;
15     }
16     int age_const_getter() const
17     {
18         return age;
19     }
20 };
21
22 int main()
23 {
24     const Person a(10);
25     a.age=10;
26     const Person b; // 错误
27     //a.age_getter(); // 错误
28 }
```

要搞清楚什么是初始化，什么是赋值？

```
const int n = 10;
n = 10;

const A a0(10,10);
const A a1;
a1 = a;
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function 'int main()':
test.cpp:25:10: error: assignment of member 'Person::age' in read-only object
   25 |     a.age=10;
      |     ~~~~~^~~
test.cpp:26:18: error: no matching function for call to 'Person::Person()'
   26 |     const Person b; // 错误
      |           ^
test.cpp:11:5: note: candidate: 'Person::Person(int)'
   11 |     Person(int x) : age(x) {}
      |     ^~~~~~
test.cpp:11:5: note: candidate expects 1 argument, 0 provided
test.cpp:5:7: note: candidate: 'constexpr Person::Person(const Person&)'
    5 | class Person
      |     ^~~~~~
test.cpp:5:7: note: candidate expects 1 argument, 0 provided
test.cpp:5:7: note: candidate: 'constexpr Person::Person(Person&&)'
test.cpp:5:7: note: candidate expects 1 argument, 0 provided
```


- 不能通过常对象调用普通成员函数（即使不改变数据成员的成员函数）
- 能通过常对象调用其常成员函数
- 能通过常对象直接访问普通的public数据成员

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Person
6  {
7  private:
8  public:
9      int age = 18;
10     Person(int x) : age(x) {}
11     int age_getter()
12     {
13         return age;
14     }
15     int age_const_getter() const
16     {
17         return age;
18     }
19 };
20
21 int main()
22 {
23     const Person a(10);
24     // a.age=10;
25     // const Person b; // 错误
26     a.age_getter(); // 错误
27     a.age_const_getter();
28     cout << a.age << endl;
29 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
```

```
test.cpp: In function 'int main()':
```

```
test.cpp:26:18: error: passing 'const Person' as 'this' argument discards qualifiers [-fpermissive]
```

```
26 |         a.age_getter(); // 错误
```

```
test.cpp:11:9: note: in call to 'int Person::age_getter()'
11 |         int age_getter()
```

常对象与常成员函数的例子

```
1. class A {
2.     public:
3.         A(int i, int j) : x(i), y(j){}
4.         void print(int z) const {
5.             cout << z-5 << " " << x * y << endl;
6.         }
7.         void print(int z) {
8.             cout << z << " " << x * y << endl;
9.         }
10.    private:
11.        int x, y;
12. };
13. int main() {
14.     const A a1(3, 4);
15.     A a2(3, 4);
16.     a1.print(5); // 输出?
17.     a2.print(6); // 输出?
18.     return 0;
19. }
```

- 上页示例程序输出：
 - 0 12 第一次print调用
 - 6 12 第二次print调用
- 两次调用在汇编代码中能看出区别（`const`关键字可以被用于参与对重载函数的区分）
 - `call _ZNK1A5printEi`
 - `call _ZN1A5printEi`
- 注意：若不存在普通成员函数，普通对象（无`const`修饰）可以调用常成员函数

```
1. class A {
2.     public:
3.         A(int i, int j) : x(i), y(j){}
4.         void print(int z) const {
5.             cout << z-5 << " " << x * y << endl;
6.         }
7. };
8. int main() {
9.     A a2(3, 4);
10.    a2.print(6); // 输出  1 12
11.    return 0;
12. }
```

指针或引用传递时，常实参不能传递给非常形参

- 值传递时，常实参可以传递给非常形参，非常实参也可以传递给常形参（因为值传递肯定不会改变实参，所以也不在乎实参是不是常类型）
- 指针或引用传递时，为防止实参改变，常实参不能传递给非常形参！

```
1  #include <iostream>
2  #include <vector>
3  #include <deque>
4  #include <list>
5  using namespace std;
6
7  void f1(const int a) {}
8  void f2(int a) {}
9
10 int main()
11 {
12     int b;
13     const int c = 10;
14
15     f1(b);
16     f2(c);
17 }
```

指针传递时，常实参不能传递给非常形参！

```
1  #include <iostream>
2  #include <vector>
3  #include <deque>
4  #include <list>
5  using namespace std;
6
7  void f1(const int* a) {}
8  void f2(int* a) {}
9
10 int main()
11 {
12     int b;
13     const int* c = &b;
14
15     f1(c);
16     f2(c);
17 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function 'int main()':
test.cpp:16:8: error: invalid conversion from 'const int*' to 'int*' [-fpermissi
ve]
   16 |         f2(c);
       |         ^
       |         |
       |         const int*
test.cpp:8:14: note: initializing argument 1 of 'void f2(int*)'
    8 | void f2(int* a) {}
       |         ~~~~~^
```

引用传递时，常实参不能传递给非常形参！

```
1  #include <iostream>
2  #include <vector>
3  #include <deque>
4  #include <list>
5  using namespace std;
6
7  void f1(const int& a) {}
8  void f2(int& a) {}
9
10 int main()
11 {
12     const int c = 10;
13
14     f1(c);
15     f2(c);
16 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function 'int main()':
test.cpp:15:8: error: binding reference of type 'int&' to 'const int' discards q
ualifiers
   15 |     f2(c);
      |         ^
test.cpp:8:14: note:   initializing argument 1 of 'void f2(int&)'
    8 | void f2(int& a) {}
      |         ~~~~~^
```

mutable关键字

- mutable是为了突破const的限制而设置的，被mutable关键字修饰的成员变量永远处于可变的狀態，即使是在被const修饰的成员函数中。
 - 在某些特殊情况下，我们需要在const函数中修改类的某些成员变量，因为要修改的成员变量与类本身并无多少关系，即使修改了也不会对类造成多少影响；且我们只想修改某个成员变量，其余成员变量仍然希望被const保护。

mutable关键字

Mutable的用处的例子：


我们想要获取getValue函数被调用次数，普遍的做法是在getValue函数里对成员变量count进行加1处理，可是getValue被关键字const修饰，无法修改count的值。这个时候mutable派上用场了！我们用mutable关键字修饰count

```
5  class Widget{
6  public:
7      Widget() : value(1), count(0) { }
8      int getValue() const;
9      int getCount() const;
10 private:
11     int value;
12     mutable int count;
13 };
14
15 int Widget::getValue() const{
16     count++;
17     return value;
18 }
19 int Widget::getCount() const{
20     return count;
21 }
22
23 int main()
24 {
25     Widget w1;
26     for(int i = 0; i < 5; i++){
27         w1.getValue();
28     }
29     std::cout << w1.getCount() << std::endl;
30     return 0;
31 }
```


mutable关键字

可对常对象（或常引用）的mutable成员进行赋值

```
5  class A
6  {
7  public:
8      A(int i = 0, int j = 0) : x(i), y(j) {}
9      int x, y;
10     mutable int z;
11 };
12 int main()
13 {
14     const A a1;
15     const A &a2 = a1;
16
17     expression must be a modifiable lvalue C/C++(137)
18     const A a1
19     View Problem Fix... (Ctrl+.)
20
21     a1.x = 10;
22     a2.y = 11;
23
24     a1.z = 20;
25     a2.z = 30;
26     cout << a1.z << endl;
27     return 0;
28 }
29
```



常引用

- 在声明引用时用**const**修饰，被声明的引用就是常引用
 - **const** 类型说明符 &引用名 = 目标对象;
- 常引用所引用的对象不能通过引用名被更新
 - 如果常引用作为形参，便不会意外地发生对实参的更改
- 注意：
 - 对常对象的引用必须使用常引用：`const A a(3, 4);`
 - `A &b = a;` // 错误
 - `const A &c = a;` // 正确
- 示例：
 - C++标准库中**basic_string**类的拷贝构造函数

```
basic_string(const basic_string& __str);  
basic_string(const basic_string& __str,  
             size_type __pos, const _Alloc& __a = _Alloc());
```

- 常引用所引用的对象不能通过引用名被更新

```
4  int main()
5  {
6      int b = 10;
7      const int &a = b;
8      b = 11; //b是可以修改的, 但是a不能修改
9
10     expression must be a modifiable lvalue C/C++(137)
11     const int &a
12     View Problem Quick Fix... (Ctrl+.)
13
14     a = 10;
15 }
```

- 常引用能够指向常量

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      const int &c = 15; //编译器会给常量15开辟一片内存, 并将引用名作为这片内存的别名
7
8      cout << c << endl;
9  }
```

常指针

- 常指针：指向常量的指针（pointer to const），不能通过常指针改变其所指向的对象的值
 - `const` 类型说明符 *指针名 = 目标地址;
 - 但是：指针可以再指向其他地址（`const`修饰的是*pointer）

```
1. class A {  
2.     public:  
3.         A(int i, int j) : x(i), y(j){}  
4.         int x, y;  
5. };  
6. int main() {  
7.     A a1(3, 4);  
8.     A a2(4, 5);  
9.     const A *ptr_a = &a1;  
10.    ptr_a->x = 10;  
  
11.    ptr_a = &a2;  
12.    return 0;  
13. }
```



常量指针

- 常量指针（`const pointer`）是将`*`放在`const`关键字之前，说明指针是一个常量，即：不变的是指针本身的值而非指向的那个值
 - 类型说明符 `* const` 指针名 = 目标地址;
 - 指针指向的对象值可以改变（`const`修饰的是`pointer`）

```
1. int main() {  
2.     A a1(3, 4);  
3.     A a2(4, 5);  
4.     A * const ptr_a = &a1;  
5.     ptr_a->x = 10;  
  
6.     ptr_a = &a2;  
7.     return 0;  
8. }
```



如果是`const A * const ptr_a = &a1`呢？

```

1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class A
6  {
7  public:
8      double xx;
9      mutable double xxx;
10     A(int i, int j) : x(i), y(j) {}
11
12 private:
13     const int x, y;
14 };
15
16
17 int main()
18 {
19     const A a(3, 4), b(3, 4);
20
21     const A *const ptr_a = &a; // 常量指针
22     ptr_a->xx = 1;
23     ptr_a = &b;
24     ptr_a->xxx = 1;
25     return 0;
26 }

```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
```

```
test.cpp: In function 'int main()':
```

```
test.cpp:22:15: error: assignment of member 'A::xx' in read-only object
```

```

22 |         ptr_a->xx = 1;
   |         ~~~~~^~~~

```

```
test.cpp:23:11: error: assignment of read-only variable 'ptr_a'
```

```

23 |         ptr_a = &b;
   |         ~~~~~^~~~

```

如果是 `const A * const ptr_a = &a` 呢？
对于 `mutable` 成员变量呢？

本章内容

1. 友元
2. 共享数据的保护
3. 命名空间

3. 命名空间

- C++中的名字可能表示变量、函数、结构、枚举、类等。当项目变得越来越大，名字之间冲突可能性也随之增大。
 - 如：程序使用来源于多个源的类库，很可能不同的库同时定义了诸如List、Tree、Node，但不同库中的定义不兼容
 - 这样的冲突被称为**命名空间问题**
- C++标准提供了**命名空间**（namespace）来为防止名字冲突提供更加可控的机制。一个命名空间的定义：
 - 关键字**namespace**，随后是命名空间的名字
 - 名字后是一系列由花括号括起来的声明和定义，包括：类、变量（及其初始化操作）、函数和**其他命名空间**
 - 位于该命名空间之外的代码必须明确指出所用的名字属于哪个命名空间（使用“命名空间名字::目标名字”的形式），如**std::cout**
- 标准库中的名字多定义在命名空间**std**中

- 下边定义了一个名为**NS**的命名空间，其中包含了一个类、一个初始化了的变量以及一个函数定义

```
1. namespace NS {  
2.     class KLAZZ { ... ... };  
3.     int val = 10;  
4.     void say_hello() {  
5.         std::cout << "Hello World\n";  
6.     }  
7. }
```

- 命名空间作用域后面（即“**}**”之后）无需分号
- 命名空间可以定义在全局作用域内，也可以定义在其他命名空间中，但不能定义在函数或类的内部

嵌套命名空间可以有两种方式定义

```
1. namespace A {  
2.     namespace B {  
3.         int x;  
4.     }  
5. }
```



```
1. namespace A::B {  
2.     int x;  
3. }
```



```
4  namespace A  
5  {  
6      int x = 1;  
7      namespace B  
8      {  
9          double x = 2;  
10     }  
11 }  
12  
13 int main()  
14 {  
15     cout << A::x << endl;  
16     cout << A::B::x << endl;  
17 }
```

打印结果:

1
2

- 在同一个作用域内，命名空间的名称不能与其他类型的实体的名称相同，否则产生冲突
- 但是同一个命名空间可以分批次定义里面的内容（不同批次命名空间的名称相同）

```
1. namespace A {
2.     int x;
3. }
4. namespace A {
5.     int y;
6. }
```



```
1. namespace A {
2.     int x;
3. }
4. class A {
5.     int y;
6. }
```



```
4  namespace A
5  {
6      int x = 1;
7      namespace B
8      {
9          double x = 2;
10     }
11 }
12 class A
13 {
14     int x2 = 1;
15 };
16
17 int main()
18 {
19     cout << A::x << endl;
20     cout << A::B::x << endl;
21 }
```

test.cpp:12:7: error: 'struct A' redeclared as different kind of entity

```
12 | class A
    |      ^
```

全局命名空间

- 除了用户定义的命名空间，还有一种**全局命名空间**，对应于文件层次的作用域。
- 文件内的全局变量可被描述为是全局命名空间的一部分，因此也可以按照引用命名空间内部名字的格式使用全局变量。
 - 由于全局命名空间没有名字，所以应用全局命名空间中名字时使用格式“**::目标名字**”，即省略命名空间名字
 - 下例中的两种引用全局变量的方式是等价的

```
1. static int GNSV = 100;
2. int main() {
3.     std::cout << GNSV << "\n";
4.     std::cout << ::GNSV << "\n";
5.     return 0;
6. }
```

- 查看C++标准库中与C语言标准头文件等价的那些文件，可以见到**通过全局命名空间的形式**引用C标准库头文件中定义的函数、变量、结构等
- 以<cstdio>为例，其中包含如下代码
 - 表明在命名空间std中使用定义在全局命名空间中的FILE、fpos_t、fgets等

```
1. namespace std
2. {
3.     using ::FILE;
4.     using ::fpos_t;
5.     using ::fgets;
6.     .....
7. } // namespace
```

匿名命名空间

- namespace关键字后不定义名字，直接紧跟“{”，这种命名空间称为**匿名命名空间**
 - 匿名命名空间中定义的名字的使用方式跟直接将相关名字定义在匿名空间的上层命名空间中一致
 - 如下例所示

```
1. namespace NS {  
2.     namespace {  
3.         int gval = 199;  
4.     }  
5. }  
6. namespace {  
7.     int gval = 99;  
8. }  
9. int main() {  
10.     std::cout << gval << " " << NS::gval << "\n";  
11.     return 0;  
12. }
```

打印结果: 99 199

如果匿名命名空间同层次具有与匿名空间中名字相同的名字，在使用名字时会引起歧义，此时需要用“::目标名字”的格式引用名字。

- 匿名空间中的名字会被匿名命名空间同层次的相同名字覆盖

```
1  #include <iostream>
2  using namespace std;
3
4  namespace NS {
5      namespace{
6          int gval = 1;
7      }
8      int gval = 2;
9  }
10 int gval = 3;
11 namespace {
12     int gval = 4;
13 }
14
15 int main() {
16     //std::cout << gval << endl; // 此行报错
17     std::cout << ::gval << " " << NS::gval << endl;
18     return 0;
19 }
20
```

打印结果：3 2

匿名命名空间的作用域

- 匿名命名空间中的名字与声明为`static`的全局名字相同，作用域被限制在当前文件中
 - 新的C++标准推荐使用匿名命名空间替代`static`声明

未命名的命名空间取代文件中的静态声明

在标准 C++ 引入命名空间的概念之前，程序需要将名字声明成 `static` 的以使得其对于整个文件有效。在文件中进行静态声明的做法是从 C 语言继承而来的。在 C 语言中，声明为 `static` 的全局实体在其所在的文件外不可见。



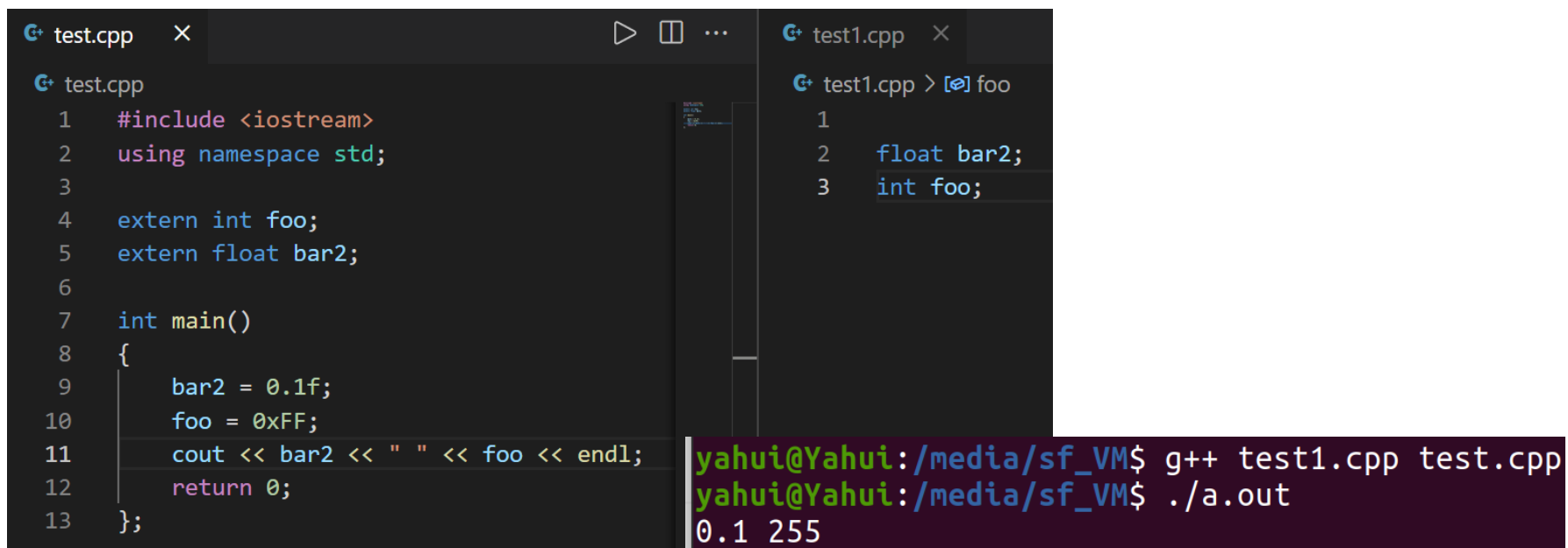
WARNING

在文件中进行静态声明的做法已经被 C++ 标准取消了，现在的做法是使用未命名的命名空间。

在链接多个C++文件时，匿名命名空间有static的作用。

C++语言支持“分别编译”

- 一个程序所有的内容，可以分成不同的部分放在不同的cpp文件里。cpp文件里的东西都是相对独立的，在编译（compile）时不需要与其他文件互通，只需要在编译成目标文件后再与其他的目标文件做一次链接（link）就行了。
- 用处：只需要编译修改过的cpp文件，再跟其他的编译文件链接，无需再次对整个项目编译。（相较而言，h文件通过“#include”包含进cpp文件中，每次都要重新编译）

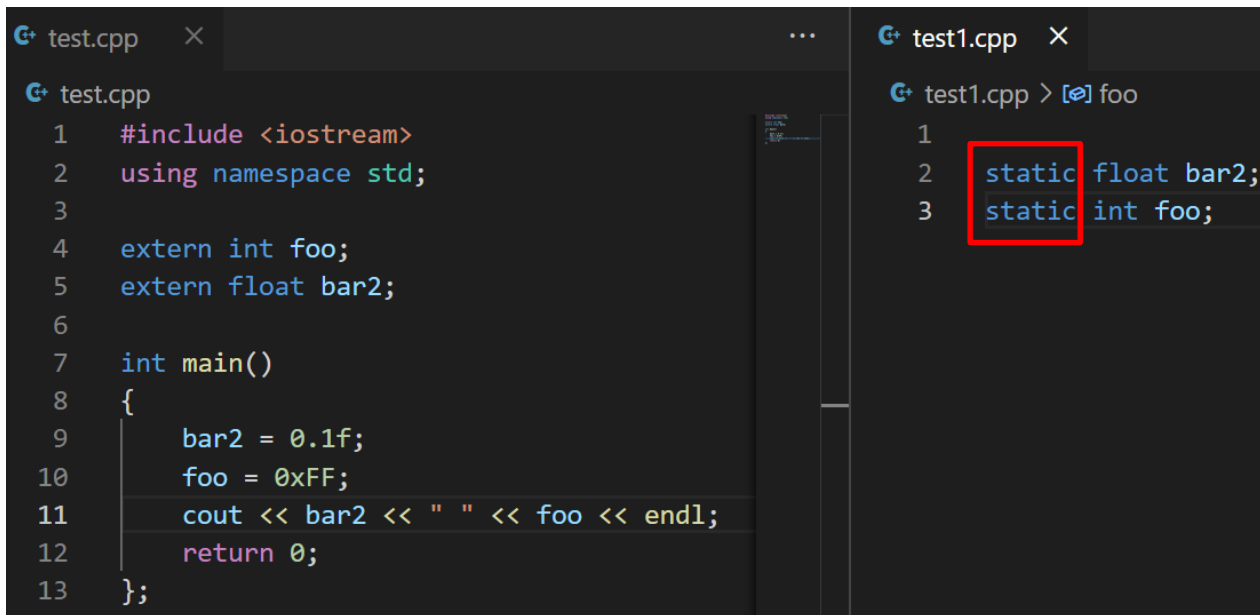


```
test.cpp
1  #include <iostream>
2  using namespace std;
3
4  extern int foo;
5  extern float bar2;
6
7  int main()
8  {
9      bar2 = 0.1f;
10     foo = 0xFF;
11     cout << bar2 << " " << foo << endl;
12     return 0;
13 };

test1.cpp
1
2  float bar2;
3  int foo;

yahui@Yahui:/media/sf_VM$ g++ test1.cpp test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
0.1 255
```

在链接多个C++文件时，匿名命名空间有static的作用。

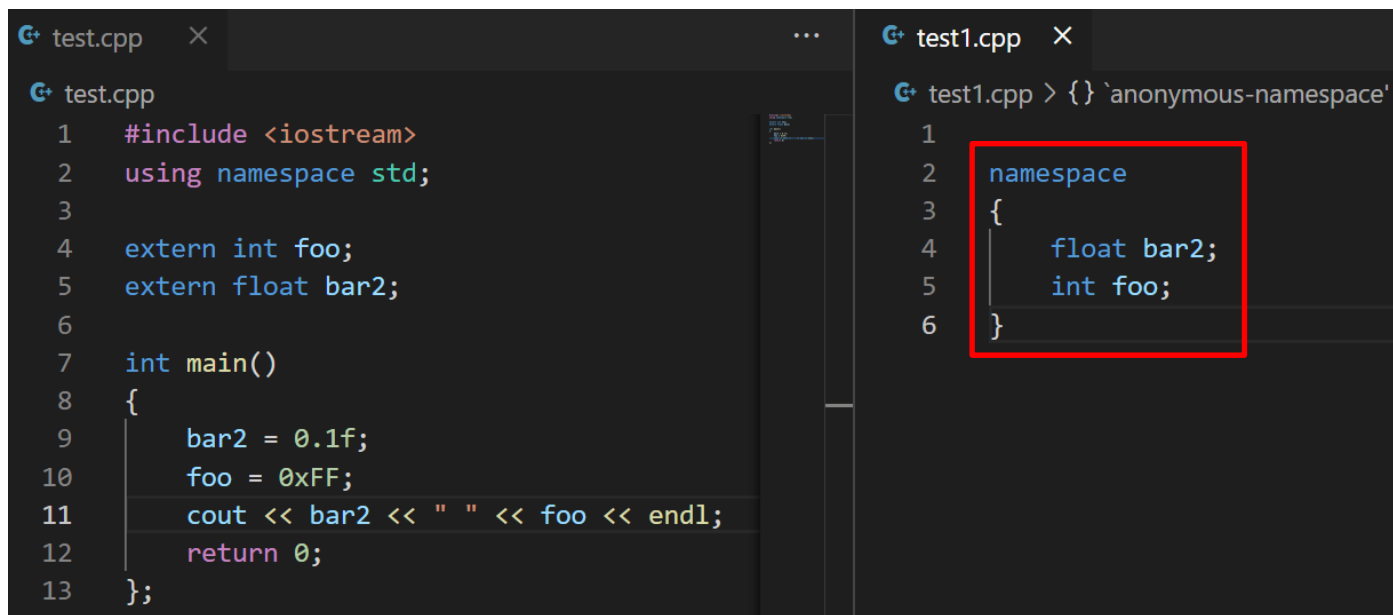


```
test.cpp
1  #include <iostream>
2  using namespace std;
3
4  extern int foo;
5  extern float bar2;
6
7  int main()
8  {
9      bar2 = 0.1f;
10     foo = 0xFF;
11     cout << bar2 << " " << foo << endl;
12     return 0;
13 };

test1.cpp > [⌕] foo
1
2  static float bar2;
3  static int foo;
```

```
yahui@Yahui:/media/sf_VM$ g++ test1.cpp test.cpp
/usr/bin/ld: /tmp/cc05NcA1.o: in function `main':
test.cpp:(.text+0x14): undefined reference to `bar2'
/usr/bin/ld: test.cpp:(.text+0x1a): undefined reference to `foo'
/usr/bin/ld: test.cpp:(.text+0x24): undefined reference to `bar2'
/usr/bin/ld: test.cpp:(.text+0x4c): undefined reference to `foo'
collect2: error: ld returned 1 exit status
```

在链接多个C++文件时，匿名命名空间有static的作用。



```
test.cpp
1  #include <iostream>
2  using namespace std;
3
4  extern int foo;
5  extern float bar2;
6
7  int main()
8  {
9      bar2 = 0.1f;
10     foo = 0xFF;
11     cout << bar2 << " " << foo << endl;
12     return 0;
13 };

test1.cpp > {} `anonymous-namespace'
1
2  namespace
3  {
4      float bar2;
5      int foo;
6  }
```

```
yahui@Yahui:/media/sf_VM$ g++ test1.cpp test.cpp
/usr/bin/ld: /tmp/cc05NcA1.o: in function `main':
test.cpp:(.text+0x14): undefined reference to `bar2'
/usr/bin/ld: test.cpp:(.text+0x1a): undefined reference to `foo'
/usr/bin/ld: test.cpp:(.text+0x24): undefined reference to `bar2'
/usr/bin/ld: test.cpp:(.text+0x4c): undefined reference to `foo'
collect2: error: ld returned 1 exit status
```

分散定义命名空间

- 命名空间无需一次性将所有应该包含的名字定义在一个地方，可以分散在多个文件内分别定义所需的名字。
 - 典型的例子是标准库中，`std`命名空间在多个头文件中定义。
 - 我们可以在头文件中的命名空间内定义多个文件可用的全局变量声明、类的声明等，而在代码实现文件的命名空间中可定义更多的仅用于当前文件的变量、函数等。
 - **每个文件只能看见当前编译单元中命名空间内定义的成员。**
 - 下边两个文件中，`ns1.cpp`中不能直接使用`NS::print()`；同理`ns2.cpp`中不能直接使用`NS::val`、`NS::say_hello`等，即使它们定义在同一个命名空间。

```
1. // ns1.cpp
2. namespace NS {
3.     class KLAZZ { ... .. };
4.     int val = 10;
5.     void say_hello() {
6.         std::cout << "Hello World\n";
7.     }
8. }
```

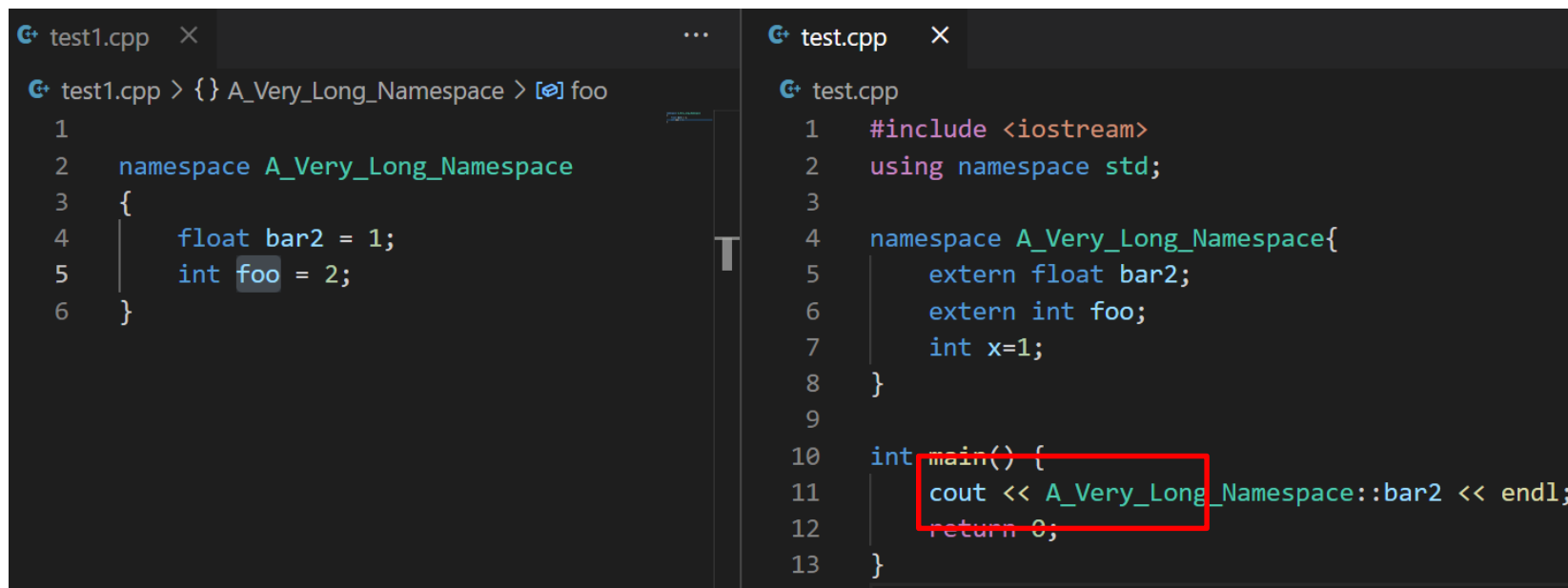
```
1. // ns2.cpp
2. namespace NS {
3.     static int sval;
4.     void print() { ... .. }
5. }
```

```
test1.cpp × ... test.cpp 1 ×
test1.cpp > {} A_Very_Long_Namespace > [x] bar2
1
2 namespace A_Very_Long_Namespace
3 {
4     float bar2 = 1;
5     int foo = 2;
6 }

test.cpp
1 #include <iostream>
2 using namespace std;
3
4 namespace A_Very_Long_Namespace{
5     int x=1;
6 }
7
8 int main() {
9     cout << A_Very_Long_Namespace::bar2 << endl;
10    return 0;
11 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test1.cpp test.cpp
test.cpp: In function 'int main()':
test.cpp:9:36: error: 'bar2' is not a member of 'A_Very_Long_Namespace'
   9 |     cout << A_Very_Long_Namespace::bar2 << endl;
     |                                     ^~~~~
```

- 利用extern修饰符使用其他cpp文件内相同命名空间里的变量。（extern用在变量或者函数的声明前，用来说明“此变量/函数是在别处定义的，要在此处引用”）



```
test1.cpp > {} A_Very_Long_Namespace > foo
1
2 namespace A_Very_Long_Namespace
3 {
4     float bar2 = 1;
5     int foo = 2;
6 }

test.cpp
1 #include <iostream>
2 using namespace std;
3
4 namespace A_Very_Long_Namespace{
5     extern float bar2;
6     extern int foo;
7     int x=1;
8 }
9
10 int main() {
11     cout << A_Very_Long_Namespace::bar2 << endl;
12     return 0;
13 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test1.cpp test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
1
```

定义命名空间成员的方式

- 头文件中命名空间中声明的变量、名字等，在实现代码文件中可以有两种方式定义。
- 一种是常规的定义命名空间的形式，一种是在命名空间之外在成员名字前加上命名空间全称的方式来定义成员。

```
1. // 头文件
2. namespace NS {
3.     void say_hello();
4. }
```

```
1. // 定义方式1
2. namespace NS {
3.     void say_hello() { ... }
4. }

5. // 定义方式2
6. void NS::say_hello() { ... }
```


使用命名空间中的成员

- 最直接使用命名空间成员的方式是在成员名字前加上其所在的命名空间全称，如`std::cout`，`NS::val`。
 - 当命名空间的名称很长时，这种方式非常繁琐。
- C++支持多种简便的使用命名空间成员的方法：
 - 命名空间别名（namespace alias）
 - `using`声明（using declaration）
 - `using`指示（using directive）

```
test.cpp
1  #include <iostream>
2  //using namespace std;
3
4  int main() {
5      cout << 1 << endl;
6      return 0;
7  }
```

```
1  #include <iostream>
2  //using namespace std;
3
4  int main() {
5      std::cout << 1 << std::endl;
6      return 0;
7  }
```

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      cout << 1 << endl;
6      return 0;
7  }
```

命名空间别名

- 命名空间别名使得我们可以为命名空间的名字设定一个短的同义词
 - 如命名空间名字为A_Very_Long_Namespace
 - 设定一个短的同义词namespace AVLNS =
A_Very_Long_Namespace

```
1  #include <iostream>
2  using namespace std;
3
4  namespace A_Very_Long_Namespace{
5      int x=1;
6  }
7  namespace NS=A_Very_Long_Namespace;
8
9  int main() {
10     cout << NS::x << endl;
11     return 0;
12 }
```

命名空间别名

- 不能在命名空间还没有定义前就声明别名
- 别名也可指向嵌套的命名空间
 - namespace Qlib = cplusplus_primer::QueryLib
 - 然后使用其中的名字: Qlib::Query q;
- 一个命名空间可以有多个别名

```
1  #include <iostream>
2  using namespace std;
3
4  namespace A_Very_Long_Namespace{
5  |   int x=1;
6  | }
7  namespace NS=A_Very_Long_Namespace;
8
9  int main() {
10 |   cout << NS::x << endl;
11 |   return 0;
12 | }
```

```
1  #include <iostream>
2  using namespace std;
3
4  namespace A_Very_Long_Namespace{
5  |   int x=1;
6  | }
7  namespace NS=::A_Very_Long_Namespace;
8
9  int main() {
10 |   cout << NS::x << endl;
11 |   return 0;
12 | }
```

- 命名空间别名的作用域仅在声明别名的cpp文件内，不能在未声明该别名的cpp文件内使用
 - 可以在头文件中声明别名，然后在多个文件内使用

```
1. // ns1.cpp
2. namespace NsOne {
3.     void say_hello() { ... }
4. }
5. namespace N1 = NsOne;
```

```
1. // ns2.cpp
2. void func() {
3.     N1::say_hello(); //编译错误: 'N1' has not been declared
4. }
```

using声明

- **using声明**，是关键词**using**紧跟一个限定词，用于引入命名空间中特定的成员
 - `using std::cout;`
 - 使用**using**声明后，可直接使用所引入的成员名字，无需加上命名空间前缀

```
1  #include <iostream>
2
3  using std::cout;
4  using std::endl;
5
6  int main() {
7      cout << 1 << endl;
8      return 0;
9  }
```

using声明

- 一个**using**声明引入命名空间中的一个成员，该成员可以是类、变量、函数，但**不能是命名空间！**

```
3  namespace NS
4  {
5      int x;
6      class C
7      {
8      };
9      namespace NSS
10     {
11     }
12 }
13
14 namespace NS2
15 {
16     using NS::C;
17     using NS::x;
18
19     using NS::NSS;           // using声明不能用于namespace
20     using namespace NS::NSS; // using指示
21 }
22
23 int main()
24 {
25 }
```

using声明

- 在哪个作用域中使用**using**声明，则所引入的成员名字可被视作当前作用域中的名字
 - 因此**要确保使用using声明的作用域中没有同名名字**
 - 在头文件中使用**using**声明，可能导致包含该头文件的代码产生始料未及的名字冲突

```
yahui@Yahoo:/media/st_VM$ g++ test.cpp
test.cpp:17:15: error: 'int NS::x' conflicts with a previous declaration
   17 |         using NS::x;
      |         ^
test.cpp:16:9: note: previous declaration 'int NS2::x'
   16 |         int x;
      |         ^
```

```
3 namespace NS
4 {
5     int x;
6     class C
7     {
8     };
9     namespace NSS
10    {
11    }
12 }
13
14 namespace NS2
15 {
16     int x;
17     using NS::x;
18 }
19
20 int main()
21 {
22 }
```

- 使用**using**声明语句引入命名空间中的成员，可以出现在全局作用域、局部作用域、命名空间作用域。
 - 在类的作用域中使用**using**声明只能指向类的基类成员（在后续章节中介绍）。

```
1  #include <iostream>
2
3  using std::cout; // 正确
4  void say_hello()
5  {
6      using std::cout; // 正确
7  }
8  namespace NsOne
9  {
10     using std::cout; // 正确
11 }
12 class K
13 {
14     using std::cout; // 错误: using-declaration for non-member at class scope
15 };
16
17 int main()
18 {
19 }
```


using指示

- **using指示**是以关键字**using**开始，紧跟关键字**namespace**及命名空间的名字。
 - `using namespace std;`
 - 一个**using**指示引入的是一个命名空间，可以是多层嵌套命名空间中的任意一层
 - 使用**using**指示后，该命名空间中所有名字都可见
- **using**指示可以出现在全局作用域、局部作用域和命名空间作用域中，但不能出现在类的作用域中。

```
1. using namespace std; // 正确
2. namespace NsOne {
3.     using namespace std; // 正确
4.     void say_hello() {
5.         using namespace std; // 正确
6.     }
7.     namespace Nest { ... ... }
8. }
```

提示：避免 using 指示

using 指示一次性注入某个命名空间的所有名字,这种用法看似简单实则充满了风险:只使用一条语句就突然将命名空间中所有成员的名字变得可见了。如果应用程序使用了多个不同的库,而这些库中的名字通过 using 指示变得可见,则全局命名空间污染的问题将重新出现。

而且,当引入库的新版本后,正在工作的程序很可能会编译失败。如果新版本引入了一个与应用程序正在使用的名字冲突的名字,就会出现这个问题。

另一个风险是由 using 指示引发的二义性错误只有在使用了冲突名字的地方才能被发现。这种延后的检测意味着可能在特定库引入很久之后才爆发冲突。直到程序开始使用该库的新部分后,之前一直未被检测到的错误才会出现。

未使用冲突名字时
不会编译错误

相比于使用 using 指示,在程序中对命名空间的每个成员分别使用 using 声明效果更好,这么做可以减少注入到命名空间中的名字数量。using 声明引起的二义性问题在声明处就能发现,无须等到使用名字的地方,这显然对检测并修改错误大有益处。



using 指示也并非一无是处,例如在命名空间本身的实现文件中就可以使用 using 指示。

作业

1. YOJ-606. 数据共享保护与操作符重载
2. YOJ-789. 数据共享保护与操作符重载2