|  |  |
| --- | --- |
| 1. 类、命名空间与作用域 2. 实参相关的查找与类类型形参 3. 查找与std::move与std::forward 4. 友元声明与实参相关的查找 | Int A::C1::f3(){return h;}  Operator>>(std::cin,s);  Std::move,std::forward  A:;C cobj;  F(cobj);  F2(); |
| 1. 类、命名空间与作用域 2. 命名空间内部的命名的查找遵循常规的查找规则：即由内向外依次查找每个外层作用域。 3. 只有位于开放块中且在使用点之前声明的名字才被考虑 4. 例子，略，就是在命名空间的类中，在此类之前的命名空间成员才会被考虑 5. 查找类作用域和命名空间作用域的相反次序 6. 实参相关的查找与类类型形参 7. operator>>在string中，string又定义在命名空间std中 8. 形参是类类型，还会查找cin和s的类所属的命名空间，就查找命名空间std，在std中，再找string就找到了输出运算符函数 9. 对于类的引用或指针同样有效 10. 更简洁 11. 查找与std::move与std::forward 12. 为了避免应用程序自定义的move与forward函数覆盖了标准库版本，所以需要书写限定语代表标准库的版本 13. 友元声明与实参相关的查找 14. 回顾：类声明一个友元函数，该友元声明并没有实际声明一个函数 15. 但若在命名空间中，若在命名空间中的类，声明一个友元函数或类（第一次出现），则我们认为它是最近命名空间的成员，与实参相关的查找规则结合一起 16. 就是这些友元函数隐式地成为命名空间A的成员，但是若没有给予类类型形参就不会查找到外层命名空间也就找不到隐式定义的成员。就报错 17. 并不是成为命名空间A的成员，只是提醒在命名空间A中有这个成员，这个成员是命名空间A的。   Friend的函数还是要在命名空间中定义，并且若普通函数有参数是本类类型，也会隐式的认为此命名空间内有这个函数的定义，则定义这个函数不能类名::函数名定义，只能普通定义代表是命名空间的成员，调用的时候需要函数名(类对象)的调用方式，不能类对象.函数名(类对象)的方式  namespace AAA {  class C {  public:  void f4(const C&);  };  void f4(const C&) {  cout << "f4(const C&)" << endl;  }  }  using AAA::C;  C obj;  f4(obj); | |