第7章 可访问性

许向阳

xuxy@hust.edu.cn

内容

7.1 作用域

7.2 名字空间

7.3 成员友元

7.4 普通友元及其注意事项

7.5 覆盖与隐藏

7.1 作用域

标识符：变量名、函数名、参数名、类型名、常量名……

可以在什么范围内被访问?

- 全局变量
- 局部变量（包括参数变量/形参）
- 语句内的变量

```
for (int i=0; i<10; i++) {...}
```

- 静态变量（static 变量）：全局、局部、类的成员

7.1 作用域

面向过程(C传统)的作用域

从小到大可以分为四级：

① 作用于表达式内 (常量)

② 作用于函数内 (函数参数、局部变量、局部类型)

③ 作用于程序文件内 (static变量和函数)

④ 作用于整个程序 (全局变量、函数、类型)

整个程序 》一个文件内 》函数内 》表达式内

7.1 作用域

面向对象的作用域

从小到大可以分为五级：

① 作用于表达式内 (常量)

② 作用于函数成员内 (函数参数、局部变量、局部类型)

③ 作用于类或派生类内 (数据/函数/类型 成员)

④ 作用于基类内 (数据/函数/类型 成员)

⑤ 作用于虚基类内 (数据/函数/类型 成员)

虚基类 》基类 》类 /派生类 》成员函数 》表达式内

7.1 作用域

有同名符号时，该符号优先解释成什么？

全局变量 **int x;**

在一个函数中又有局部变量 **int x;**

该函数中 有 **x=5;** **x**是指的全局变量还是局部变量？

- 标识符作用域越小，被访问优先级就越高。

问：当函数成员的参数和数据成员同名时，优先访问谁？

7.1 作用域

有同名符号时，指定是用哪儿定义的符号 (::)

单目:: 指定为全局标识符

全局类型名、全局变量名、全局函数名等

```
int  x;
void  f ()
{
    int  x;
    :: x = 10;   // 全局变量
    x =20;       // 局部变量
}
```

7.1 作用域

有同名符号时，指定是用哪儿定义的符号(::)

双目:: 指定类或者名字空间中的枚举元素、数据成员、函数成员、类型成员等。

用法：类名::成员名

::的优先级为最高级，结合性自左向右。

```
class STACK{
        struct NODE { NODE(int v); };
};
STACK::NODE::NODE(int v) { } //自左向右结合
```

7.1 作用域

```
class A {
public:  int x;
   int getx() {cout<< “class A” <<endl;  return x;}
};
class B : public A {
public: int x;
        int getx() { cout<< “class B” <<endl;  return x;  }
        int setx(int x) {
            this->x=x;  // (*this).x=x;   B::x=x;
            A::x=2*x;
};
B  b;
Z=b.getx();
Z=b.A::getx();
```

```
A::x = 2 * x;
B::A::x = 2 * x;
this->A::x = 2*x;
(*this).A::x = 2*x;
((A*)this)->x = 3 * x;
```

7.1 作用域


```
class  POINT2D{
    int  x, y;
public:
    int  getx( ) {return x; };
    POINT2D (int x, int y){
        POINT2D::x=x;
        this->y=y;
    }
} p(3,5);
static  int  x=7;
void main(void) {
    int  x=p.POINT2D::getx( ); //等价于x=p. getx( )
    ::x=POINT2D(4,7).getx( );
                //常量对象POINT2D(4,7)作用域局限于表达式
}
```

7.2 名字空间

- 名字空间 namespace 是C++引入的一种新作用域;
- C++名字空间既面向对象又面向过程：除可包含类外，还可包含函数、变量定义;
- 名字空间必须在全局作用域内用namespace定义，不能在类、函数及函数成员内定义，最外层名字空间名称必须在全局作用域唯一;
- 同一名字空间内的标识符名必须唯一，不同名字空间内的标识符名可以相同;
- 当程序引用多个名字空间的同名成员时，可以用名字空间加作用域运算符::限定。

7.2 名字空间

- 名字空间 namespace 的定义

```
namespace ALPHA {  //初始定义ALPHA
    int x;                    // 定义变量x
   void g(int);               // 声明函数原型void g(int)
   void g(long) {             //定义函数void g(long)
            cout << "Processing a long argument.\n";
   }
}
```

- 名字空间的使用

```
using namespace ALPHA ;
Using ALPHA::x;
```

7.2 名字空间

- 名字空间可分多次和嵌套地用namespace定义

```
namespace A {
    int x;
    namespace B {
        namespace C {
            int k=4;
        }
    }
}
namespace AB=A::B;
 using namespace A::B::C;
   using namespace AB;//A::B无成员可用
```

7.2 名字空间

- 直接访问成员

  std::cout <<“hello”<<std::endl;

- 引用名字空间的某一个成员

  using std::cout;

  cout<<“hello”<<std::endl;

- 引用名字空间

  using namespace std;

  cout<<“hello”<<endl;

- 先定义、后引用

7.2 名字空间


```
namespace ALPHA {          //初始定义ALPHA
   int x;
   void g(int t);          //声明void g(int)
    g(long t){ ……  };      //定义void g(long)
}
namespace ALPHA {          //扩展定义ALPHA
   int y=5;                //定义整型变量y
   void g(void);           //新函数void g(void)
}
 using ALPHA::g;           //声明引用名字空间void g(int)和g(long)
void main(void) {
   g(ALPHA::x);            //调用函数void g(int)
}
```

7.2 名字空间


```
namespace A { int x=1; };
namespace B { int y=2; };
namespace C { int z=3; }
namespace   { int m=4; }
using namespace A; //此用法允许在全局作用域定义新x
using B::y;              //此用法不允许在全局作用域定义y
int z=x+3;  //访问A::x
int x=y+2;  //访问B::y，，此时定义了一个全局变量 x
int v=::x+A::x;  //用::区分全局变量x和名字空间成员x
//int y=4;    //错误，当前作用域有变量y
int main(void){   return  z;  }    //优先访问全局变量::z
```

7.3 成员友元

友元：

不是本类的函数成员；

但可以像类的函数成员一样，访问该类的所有成员

7.3 成员友元


```
class Student {
   private:
      int  number;
      char name[15];
      float score;
   public:
       Student(int number1, char* name1, float score1);
       Student() {  };
       Student(const Student &a);
       void Print();   // 显示信息
       friend void display(Student &a); // 显示信息
};

void display(Student &a) { ……}
```

7.4 普通友元及其注意事项

```
void Student::Print()      // 函数成员
{
    cout << " name : " << name << " score: " << score << "\n";
}
void display(Student &s)   // 非 Student的函数成员，是友元
{
   cout << " name : " << s.name << " score: " << s.score << "\n";
}                                                  // 访问私有成员
int  main()
{   Student  stu1(……);
    Stu1.Print();
    display(stu1);
}
```

7.4 普通友元及其注意事项


```
void Student::Print()
{ ……  }

void display(Student &s)
{ ……  }
```

普通友元：C语言普通函数

display 声明为类的友元

缺点：　破坏了类中信息隐藏的特性，可访问私有成员

为什么定义友元函数，而不直接将其定义为成员函数？

优点：提高程序的运行效率

减少类型检查和安全性检查的时间开销

7.4 普通友元及其注意事项

友元函数

- 友元函数不是声明该友元的当前类的成员
- 不受访问权限的限制，可以在任何访问权限下用**friend**
- 可以访问当前类的任何成员
- 一个函数可成为多个类的友员

- 若在类体定义友元函数体，友元函数自动成为内联函数
- 同其他内联函数一样，内联有可能失败。

7.4 普通友元及其注意事项

友元类：一个类是另外一个类的友元

```
class A {
        friend class B;   // B为类A的友元类
};
class B { …b1(…); …b2(…); };
```

- 类B 的所有函数成员都是类A的友元
- 关系不传递
- 友元关系不对称
- static、virtual、friend 只能单个独立使用。

7.4 普通友元及其注意事项

华中科技大学

友元类：一个类是另外一个类的友元

```
#include <iostream>
using namespace std;
class A {
private:
        int x;
public:
        A(int x) { this->x = x; }
        friend class B;   // 类B 为类A的友元
};                        // 类B的所有函数成员都是类A的友元，
                          // 均可访问A的私有成员
```

7.4 普通友元及其注意事项


```
class B {
  public:
     void display(A &a)
       {  cout << "display A :" << a.x << endl;  }
};
// 若删除A类中的申明  friend class B;
// 则B类中的display函数不能访问A的私有成员
int main()
{        A  a(10);
         B  b;
         b.display(a);
         return 0;
}
```

7.4 普通友元及其注意事项

一个类的成员函数能否成为另一个类的友元？

```
#include <iostream>
using namespace std;
class B {
    public: void display();
};
class  A {
private:
    int x;
public:
     A(int x) { this->x = x; };
     friend void  B::display();
};
```

7.4 普通友元及其注意事项


```
void B::display() {
        A  a(10);
        cout << "display A :" << a.x << endl;
}

int main()
{
        B  b;
        b.display();
        return 0;
}
```

7.4 普通友元及其注意事项

类成员函数作为另一个类的友元

```
// A.h
#include "B.h"
class  A {
private:      int x;
public:
    A(int x);
    friend void  B::display();
};
```

```
// A.cpp
#include "A.h"
A::A(int x) { this->x = x; }
```

```
// B.h
class B {
public:
    void display( ) ;
};
```

```
// B.cpp
#include "B.h"
#include "A.h"
#include <iostream>
using namespace std;
void B::display() {
 A  a(10);
 cout << "display A :" << a.x;
}
```

7.5 覆盖与隐藏

当有基类成员和派生类成员同名时

- **隐藏**：通过派生类对象只能访问到派生类成员，而无法访问到其基类的同名成员。

- **覆盖**：通过派生类对象还能访问到基类的同名成员。

派生类对象 . 基类名称 :: 成员名称

Q：怎样实现隐藏？

7.5 覆盖与隐藏

- 在派生类函数中，基类的保护和公开成员会被派生类同名函数覆盖。
- 在一个函数中派生类成员隐藏了基类同名成员，但在另一个函数中可能只是覆盖。
- 注意：访问权限

7.5 覆盖与隐藏


```
class BAG {        //例7.19   包
   int  *const e;   // 包中元素指针
   const  int  s;    //  包中能够存放的 最大元素数目
   int      p;          // 当前元素 数目
public:
   BAG(int m): e(new int[m]), s(e ? m : 0) { p = 0; }
   virtual ~BAG( ) { delete e;         };
   virtual int pute(int f) {
      return p < s ? (e[p++] = f, 1) : 0;
   } //BAG允许重复的元素
   virtual int getp( ) { return p; }
   virtual int have(int f) {
      for (int i = 0; i < p; i++)
         if (e[i] == f) return 1; return 0;
   }
};
```

7.5 覆盖与隐藏

```
class SET : public BAG{
public:
   int pute(int f)
   { return have(f) ? 1 : BAG::pute(f); } //不能去掉BAG::，否则自递归
   SET(int m): BAG(m) { }
};
```

- SET中不允许重复元素，BAG中允许重复元素
- SET中，要重新定义 pute(…)，
  BAG 中的 pute(…) 允许加入重复元素
- SET没有数据成员，可直接使用编译程序自动生成的~SET( )，
  它将自动调用 ~BAG( )
- 若自定义了构造函数、析构函数、赋值运算符重载函数，编译程序不再生成原型相同的函数，相当于自动屏蔽或隐藏了编译的原型相同的函数。

7.5 覆盖与隐藏


```
void main( ) {
    SET s(10);
    s.pute(2);
    s.BAG::pute(2);   //BAG::pute()只是被覆盖，可以被调用
    s.BAG::getp( );
    int x = s.getp( );   //BAG::getp()被重用，SET中没有自定义getp函数
    x = s.have(2);       //BAG::have()被重用，SET没有自定义have()函数
}
```

Q： 派生类SET的对象 s，能够访问基类的同名函数，存在什么问题？

Q： 如何解决存在的问题， 让 s.BAG::pute(…) 不可访问？

隐藏

7.5 覆盖与隐藏


```
class SET : protected BAG {
    using BAG::pute;        //使基类函数成员成为私有成员，
                            // 派生类还可定义同名函数
public:
    using BAG::have;        // 重用基类的实例函数成员
    BAG::getp;              // 等价于using BAG::getp;
    int pute(int f) { return have(f) ? 1 : BAG::pute(f); }
    SET(int m) : BAG(m) { }
    SET(const SET& s) : BAG(s) { }
    SET(SET&& s) : BAG((BAG&&)s) { }
} s(10);

// s.BAG::pute(2);    // 被隐藏，不能调用
                      SET中int BAG::pute(int)为private
```

7.5 覆盖与隐藏

- 在派生类中， using特定基类数据成员后，不允许再在派生类中定义同名数据成员，并且可以通过前述using改变或指定新的访问权限。

- 在派生类中， using特定基类函数成员后，还可以再在派生类中定义同名函数成员，并且可以通过前述using改变或指定基类成员继承后的访问权限。

总结

華中科技大學

作用域

名字空间

普通友元

成员友元