



# C++程序设计精要教程

华中科技大学

# 第7章 可访问性

## ◆7.1 作用域

- 作用域：标识符起作用的范围。作用域运算符::既是单目运算符，又是双目运算符。其优先级和结合性与括号相同。
- 单目::用于限定全局标识符(类型名、变量名、函数名以及常量名等)。
- 双目::用于限定类的枚举元素、数据成员、函数成员以及类型成员等。双目运算符::还用于限定名字空间成员，以及恢复从基类继承的成员的访问权限。
- 在类体外定义数据和函数成员时，必须用双目::限定类的数据和函数成员，以便区分不同类之间的同名成员。

# 第7章 可访问性

【例7.1】 定义二维及三维坐标上的点的类型。

```
class POINT2D{  
    //定义二维坐标点  
    int x, y;  
public:  
    //获得点的二维x轴坐标  
    int getx();  
    POINT2D (int x, int y){  
        //int x访问优先于数据成员x  
        POINT2D::x=x;  
        //POINT2D::x为数据成员x  
        POINT2D::y=y;  
    }  
};
```

```
class POINT3D{  
    //定义三维坐标点  
    int x, y, z;  
public:  
    //获得点的三维x轴坐标  
    int getx();  
    POINT3D (int x, int y, int z){  
        POINT3D::x=x;  
        //POINT3D::x为数据成员x  
        POINT3D::y=y;  
        POINT3D::z=z;  
    }  
};
```

# 第7章 可访问性

//以下代码在类的体外定义getx(), 用::限定getx所属的类

```
int POINT2D::getx() {return x;}
```

```
int POINT3D::getx() {return x;}
```

```
static int x;
```

```
void main(int argc, char *argv[ ]) {
```

```
    POINT2D p(3,5);
```

```
    int x;
```

```
    x=p.POINT2D::getx();
```

```
    x=::x+p.getx();          //等价于x=::x+p.POINT2D::getx();
```

```
    x=POINT2D(4,7).getx();
```

```
    //常量POINT2D(4,7)的作用域局限于表达式
```

```
}
```

# 第7章 可访问性

## ◆7.1 作用域

- 作用域分为面向对象的作用域、面向过程的作用域(C传统的作用域, 含被引用的名字空间及成员)。
- 面向过程的: 词法单位的作用范围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(参数和局部自动变量、局部类型), ③作用于程序文件内(static变量、函数), ④作用于整个程序(全局变量、函数、类型)。
- 面向对象的: 词法单位的作用范围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(参数和局部自动变量、局部类型), ③作用于类或派生类内, ④作用于基类内, ⑤作用于虚基类内。
- 标识符作用域越小, 被访问优先级就越高。当函数成员的参数和数据成员同名时, 优先访问的是函数成员的参数。作用域层次: 面向对象->面向过程。

# 第7章 可访问性

【例7.2】用链表定义容量无限的栈。

```
#include <iostream.h>
```

```
class STACK{
```

```
    struct NODE{
```

```
        int val; NODE *next;
```

```
        NODE(int v);
```

```
    }*head; //head为数据成员
```

```
public:
```

```
    STACK(){head=0;} //0为空指针
```

```
    ~STACK();
```

```
    int push(int v); int pop(int &v);
```

```
};
```

```
STACK::NODE::NODE(int v){
```

```
//::自左向右结合, 函数的所属类
```

```
    val=v; next=0;
```

```
}
```

```
STACK::~~STACK(){
```

```
    NODE *p;
```

```
    while(head)
```

```
        {p=head->next; delete head; head=p;}
```

```
}
```

```
int STACK::push(int v){...}
```

```
int STACK::pop(int &v){...}
```

```
void main(void){
```

```
    STACK stk; int v;
```

```
    if(stk.push(5)==0)
```

```
        {cout<< "Stack overflow" ; return; }
```

```
    if(stk.pop(v)==0)
```

```
        {cout<< "Stack underflow" ; return; }
```

```
}
```

# 第7章 可访问性

## ◆7.1 作用域

- 单目运算符::可以限定存储类型为static和extern的全局变量、函数、类型以及枚举元素等。

```
extern int fork();//fork 外部函数
```

```
class Process{
```

```
    int processes;
```

```
public:
```

```
    int fork();//自定义fork函数
```

```
};
```

```
static int processes=1; //总进程数
```

```
int Process::fork()
```

```
{
```

```
    processes++; //访问数据成员processes
```

```
    ::processes++; //访问static变量processes
```

```
    return ::fork();//调用外部fork, 去掉::会自递归
```

```
}
```

- 当同一作用域的标识符和类名同名时, 可以用class、struct和union限定标识符为类名。

```
class CLERK{... }; int CLERK; class CLERK v("V", 0);
```

# 第7章 可访问性

## ◆7.2 名字空间

- 名字空间是C++引入的一种新作用域，类似Java(只包含类的)包。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。

```
namespace A{int x, f(){return 1;}; class B{/*...*/};};  
class B{ namespace C{ int y; }; int z; }; //错  
namespace B::C{ int z; }; //错  
void f(){ namespace E{ int x; }; }; //错
```

- 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



# 第7章 可访问性

## ◆7.2 名字空间

- 名字空间(包括匿名名字空间)可以分多次定义(在不同文件里):
  - 可以先在初始定义中定义一部分成员,然后在扩展定义中再定义另一部分成员;
  - 或者先在初始定义中声明的函数原型,然后在扩展定义中再定义函数体;
  - 初始定义和扩展定义的语法格式相同。
- 保留字using用于指示程序要引用的名字空间,或者用于声明程序要引用的名字空间成员。
- 在引用名字空间的某个成员之前,该成员必须已经在名字空间中声明了原型或进行了定义。

# 第7章 可访问性

【例7.5】访问名字空间ALPHA中定义的变量x及函数g。

```
#include <iostream.h>

namespace ALPHA { //初始定义ALPHA
    extern int x; //声明整型变量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";
    }
}

using ALPHA::x; //声明引用变量x
using ALPHA::g; //声明引用void g(int)和g(long)
```

# 第7章 可访问性

```
namespace ALPHA {           //扩展定义ALPHA
    int x=5;                 //定义整型变量x
    void g(int a)            //定义函数void g(int)
    { cout << "Processing a int argument.\n"; }
    void g(void)             //定义新的函数void g(void)
    { cout << "Processing a void argument.\n"; }
}

void main(void) {
    g(4);                    //调用函数void g(int)
    g(4L);                   //调用函数void g(long)
    cout<<"X="<<x;          //访问整型变量x
    g(void);                 //using之前无该原型，失败，必须ALPHA::g(void);
}
```

# 第7章 可访问性

## ◆7.2 名字空间

- 名字空间成员三种访问方式：①直接访问成员，②引用名字空间成员，③引用名字空间。
- 直接访问成员的形式为：<名字空间名称>::<成员名称>。直接访问总能唯一的访问名字空间成员。
- 引用成员的形式为：using <名字空间名称>::<成员名称>。如果引用时只声明或定义了一部分重载函数原型，则只引用这些函数，并且引用时只能给出函数名，不能带函数参数。
- 引用名字空间的形式为：using namespace <名字空间名称>，其中所有的成员可用。多个名字空间成员同名时用作用域运算符限定。



# 名字空间-using声明

using声明 ( **using declaration** ) 以关键字using开头, 后面是名字空间成员名 (限定修饰名- qualified name), 如果是声明嵌套多层的名字空间里的成员, 要多级限定。

using声明的效果是将包含在名字空间的一个成员引入到新的作用域 (using声明所在的域), 因此在using声明所在的域不能再定义同名的函数或变量

```
namespace A { int i = 0; }
```

```
using A::i; //在当前域引入名字i, i的作用域从声明开始到当前域结束
```

```
int i = 0; //错误, 该域已经有名字i, 不能再定义同名变量
```

# 名字空间-using指示符

using指示符 ( **using directive** ) 使名字空间包含的成员都可见;

using指示符以关键字using开头, 后面是名字空间名;

using指示符的效果是程序可以直接使用名字空间里的成员而不用加限定符, **但它没有把名字空间的成员引入到当前域** (即using指示符出现的域)。因此在当前域可以定义与名字空间里成员同名的变量和函数。但二义性只有当名字被使用时才被检测到;

```
namespace A { int i = 0; int j = 0; }
```

```
using A; //并没有将成员引入到当前域, 只是使程序可以直接用成员名
```

```
int x = i; //可以直接用名字i
```

```
int j = 1; //可以定义同名变量
```

```
//如果程序到此结束, 编译器不会报错。因为没有使用变量j
```

```
int y = j + 1; //使用了j, 编译器会报错, 因为不知道是哪个j
```

```
int z = A::j + ::j; //OK
```

# 第7章 可访问性

## ◆7.2 名字空间

- 嵌套名字空间：名字空间内可定义名字空间，形成多个层次的作用域，引用时多个作用域运算符自左向右结合。
- 引用名字空间后，整个名字空间所有成员都能被访问。同名冲突时，用作域运算符限定名字空间成员。



# 第7章 可访问性

```
namespace A {    // A的初始定义
    int x = 5;
    int f() { return 6; }
    namespace B { int y = 8, z = 9; }
    using namespace B;
}
using A::x;        //特定名字空间成员using声明, 不能再定义变量x
using A::f;        //特定名字空间成员using声明, 不能再定义函数f
using namespace A::B;    //非特定成员using, 可访问A::B::y, A::B::z, 还可重新定义
int y = 10;        //定义全局变量y
void main(void) {
    f();            //调用A::f()
    A::f();         //调用A::f()
    ::A::f();       //调用A::f()
    cout<<x+ ::y + z + A::B::y; //同一作用域有两个y, 必须区分
}
```

# 第7章 可访问性

## ◆7.2 名字空间

- 可以为名字空间定义别名，以代替过长和多层的名字空间名称。对于**嵌套定义的名字空间**，使用别名可以大大提高程序的可读性。
  - namespace **AB**=A::B;
- 匿名名字空间的作用域为当前程序文件**，名字空间被自动引用，其成员定义不加入当前作用域（面向过程或面向名字空间），即可以在当前作用域定义同名成员。一旦同名冲突，自动引用的匿名名字空间的成员将是不可访问的。
- 在同一个文件里，匿名名字空间也可分多次定义。

# 名字空间-匿名名字空间

由于没有名字，匿名名字空间声明后自动引用

```
namespace {int m = 4; int n = 5;}
```

等价于

```
namespace _empty_name {int m = 4; int n = 5;}
```

```
using namespace _empty_name ;
```

因此我们可以直接使用m， n

```
namespace {int m = 4; int n = 5;}
```

```
int z = m + n; //OK
```

```
int m = 0; //可以定义同名变量，但匿名空间里的m永
```

远访问不到了

# 名字空间-匿名名字空间

```
//匿名名字空间作用域是当前文件，在一个文件里可以分多次定义，如下例
namespace {
    void f(); //第一部分匿名名字空间里只申明函数
}

namespace {
    //第二部分匿名名字空间里，给出函数定义
    void f() { std::cout << "Hello World!\n"; }
}

int main()
{
    f();
}
```

# 第7章 可访问性

【例7.10】名字空间别名和匿名名字空间。

程序文件A.CPP如下:

```
#include <iostream.h>
namespace {
    //匿名, 独立, 局限于A.CPP,
    //不和B.CPP的合并
    void f() {cout<< "A.CPP\n" ;}
    //必须在名字空间内定义函数体
}
namespace A{int g(){return 0;}}
//名字空间A将和B.CPP合并
int m(){
    f(); return A::g();
}
```

程序文件B.CPP如下:

```
#include <iostream.h>
namespace A{
    int g();
    namespace B{
        namespace C{int k=4;}
    }
}
namespace ABCD=A::B::C;
//定义别名ABCD
using ABCD::k;
//引用成员A::B::C::k
```

# 第7章 可访问性

程序文件B.CPP如下(续):

```
namespace {    //独立的，局限于B.CPP，不和A.CPP的合并
    int x=3;    //相当于在本文件定义static int x=3
    void f() { cout<< "B.CPP\n" ;}
    class ANT{ char c; };
}
int x=5;        //定义全局变量
int z=::x+k;    //冲突，必须使用::，匿名名字空间x永远不能访问
int main(void){
    ANT a;
    m(); f();
    return A::g();
}
```

# 第7章 可访问性

## ◆7.3 成员友元

- **成员友元**是一种将一个类的函数成员声明为其它类友元的函数。派生类函数要访问基类私有成员，必须定义为基类的友元。
- 如果类A的**实例**函数成员被声明为类B的成员友元，则这种友元称为**实例成员友元**。如果类A的**静态**函数成员被声明为类B的成员友元，则这种友元称为**静态成员友元**。
- 如果某类A的所有函数成员都是类B的友元，则可以简单的在B的定义体内用 `friend A;` 声明，不必列出A的所有函数成员。此时称类A为类B的**友元类**。
  - 友元关系不能传递，即若类A是类B的友元类，类B是类C的友元类，此时类A并不是类C的友元类；
  - 友元关系也不能互换，即类A是类B的友元类，类B并不一定是类A的友元类

# 第7章 可访问性

```
#include <iostream.h>
class B; //A、B类互为依赖
class A{
    int i;
public:
    int set(B &);
    int get( ) { return i; };
    A(int x) { i=x; };
};
class B{
    int i;
public:
    B(int x) { i=x; };
    friend A;
    //声明A为B的友元类
};
```

```
int A::set(B&b){
    return i=b.i;
    //在A的成员函数体内访问
    //B的私有数据成员
}
void main(void){
    A a(1);
    B b(2);
    a.set(b);
    cout<<"a.i="<<a.get( );
} //输出: a.i=2
```



# 第7章 可访问性

## ◆7.4 普通友元及其注意事项

- 包括主函数main在内，任何普通函数(全局函数)都可以定义为一个类的普通友元。普通友元不是类的函数成员，故普通友元可在类的任何访问权限下定义。一个普通函数可以定义为多个类的普通友元。
- 友元函数的参数也可以缺省和省略。
- 普通友元可以访问类的任何数据成员和函数成员。【例7.14】
- 未声明为当前类友元的函数只能访问当前类的公有成员，声明为当前类友元的函数可以访问类的所有成员。【例7.15】

# 第7章 可访问性

## ◆7.4 普通友元及其注意事项

- 任何函数的原型声明及其函数定义都可分开，但函数的函数体只能定义一次。在声明普通友元时，也可同时定义函数体(自动内联)。 【例7.16】
- 内联的友元函数的存储类默认为static，作用域局限于当前代码文件。全局main的作用域为整个程序，故不能在类中内联并定义函数体，否则便会成为局部(即static)的main函数。

```
struct A{  
    friend void main(void); //全局函数main()定义为A的普通友元  
    A() {}                 //自动成为inline()函数  
};  
void main(void){A a(5);} //全局函数main()为A的普通友元
```