第 5 章

数据的共享与保护

C++ 语言是适合于编写大型复杂程序的语言，数据的共享与保护机制是 C++ 语言的重要特性之一。本章介绍标识符的作用域、可见性和生存期的概念，以及类成员的共享与保护问题。最后介绍程序的多文件结构和编译预处理命令，即如何用多个源代码文件来组织大型程序。

5.1 标识符的作用域与可见性

作用域讨论的是标识符的有效范围，可见性是讨论标识符是否可以被引用。我们知道，在某个函数中声明的变量就只能在这个函数中起作用，这就是受变量的作用域与可见性的限制。作用域与可见性既相互联系又存在着很大差异。

5.1.1 作用域

作用域是一个标识符在程序正文中有效的区域。C++ 语言中标识符的作用域有函数原型作用域、局部作用域(块作用域)、类作用域、文件作用域、命名空间作用域和限定作用域的 enum 枚举类。

1. 函数原型作用域

函数原型作用域是 C++ 程序中最小的作用域。第 3 章中介绍过，在函数原型中一定要包含型参的类型说明。**在函数原型声明时形式参数的作用范围就是函数原型作用域**。例如，有如下函数声明：

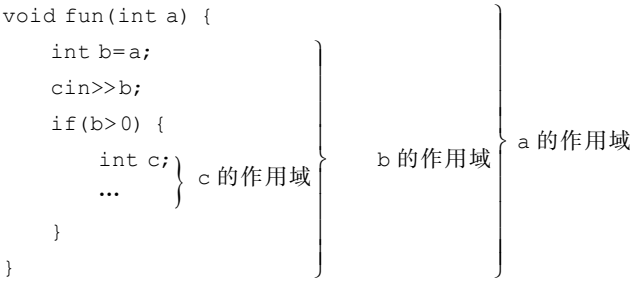
```
double area(double radius);
```

标识符 radius 的作用(或称有效)范围就在函数 area 形参列表的左右括号之间，在程序的其他地方不能引用这个标识符。因此标识符 radius 的作用域称作函数原型作用域。

注意 由于在函数原型的形参列表中起作用的只是形参类型，标识符并不起作用，因此是允许省去的。但考虑到程序的可读性，通常还是要在函数原型声明时给出形参标识符。

2. 局部作用域

为了理解局部作用域，先来看一个例子。



这里，在函数 fun 的形参列表中声明了形参 a，在函数体内声明了变量 b，并用 a 的值初始化 b。接下来，在 if 语句内，又声明了变量 c。a、b 和 c 都具有局部作用域，只是它们分别属于不同的局部作用域。

函数形参列表中形参的作用域，从形参列表中的声明处开始，到整个函数体结束之处为止。因此，形参 a 的作用域从 a 的声明处开始，直到 fun 函数的结束处为止。**函数体内声明的变量，其作用域从声明处开始，一直到声明所在的块结束的花括号为止**。所谓块，就是一对花括号括起来的一段程序。在这个例子中，函数体是一个块，if 语句之后的分支体又是一个较小的块，二者是包含关系。因此，变量 b 的作用域从声明处开始，到它所在的块（即整个函数体）结束处为止；而变量 c 的作用域从声明处开始，到它所在的块，即分支体结束为止。**具有局部作用域的变量也称为局部变量**。

3. 类作用域

类可以被看成是一组有名成员的集合，类 X 的成员 m 具有类作用域，对 m 的访问方式有如下 3 种。

（1）**如果在 X 的成员函数中没有声明同名的局部作用域标识符，那么在该函数内可以直接访问成员 m**。也就是说 m 在这样的函数中都起作用。

（2）**通过表达式 x.m 或者 X::m。这正是程序中访问对象成员的最基本方法**。X::m 的方式用于访问类的静态成员，相关内容将在 5.3 节介绍。

（3）**通过 ptr->m 这样的表达式，其中 ptr 为指向 X 类的一个对象的指针**。关于指针将在第 6 章详细介绍。

C++ 中，类及其对象还有其他特殊的访问和作用域规则，在后续章节中还会深入讨论。

4. 文件作用域

不在前述各个作用域中出现的声明，就具有文件作用域，这样声明的标识符其作用域开始于声明点，结束于文件尾。例 5-1 中所声明的全局变量就具有文件作用域，它们在整个文件中都有效。

5. 命名空间作用域

生活中存在重名现象，在 C++ 应用程序中，也存在同名变量、函数和类等情况，为避免重名冲突，使编译器能够区分来自不同库的同名实体，C++ 引入了命名空间的概念，它本质上定义了实体所属的空间。命名空间定义使用 namespace 关键字，声明方式如下：

```
namespace namespace_name{
    //代码声明
}
```

使用某个命名空间中的函数、变量等实体,需要命名空间::实体名称或通过 using namespace namespace_name 的方式。例 5-1 中 using namespace std 使得标准命名空间中实体调用无须加空间前缀,而 my_space 中的 func 通过::方式调用。

6. 限定作用域的 enum 枚举类

我们在第 4 章介绍了 enum 枚举类,枚举类分为限定作用域和不限定作用域两种,由于之前未涉及作用域的概念,因此第 4 章只给了不限定作用域的例子,在这里对限定作用域的 enum 枚举类型做更深入的讨论。

定义限定作用域的枚举类型的方式是 enum class {...},即多了 class 或 struct 限定符,此时枚举元素的名字遵循常规的作用域准则,即类作用域,在枚举类型的作用域外是不可访问的。相反,在不限定作用域的枚举类型中,枚举元素的作用域与枚举类型本身的作用域相同:

```
enum color {red, yellow, green};          //不限定作用域的枚举类型
enum color1 {red, yellow, green};         //错误,枚举元素重复定义
enum class color2 {red, yellow, green}; //正确,限定作用域的枚举元素被隐藏了
color c=red;                              //正确,color 的枚举元素在有效的作用域中
color2 c1=red;                            //错误,color2 的枚举元素不在有效的作用域中
color c2=color::red;                      //正确,允许显式地访问枚举元素
color2 c3=color2::red;                    //正确,使用了 color2 的枚举元素
```

例 5-1 作用域实例。

```
//5_1.cpp
#include <iostream>
using namespace std;

int i;                                    //全局变量,文件作用域

int main() {
    i=5;                                  //为全局变量 i 赋值
    {                                     //子块 1
        int i;                            //局部变量,局部作用域
        i=7;
        cout<<"i="<<i<<endl;              //输出 7
    }
    cout<<"i="<<i<<endl;                  //输出 5
    return 0;
}
```

运行结果:

```
i=7
i=5
```

在这个例子中,在主函数之外声明的变量 i 具有文件作用域,它的有效作用范围到文件尾才结束。在主函数开始处给这个具有文件作用域的变量赋初值 5,接下来在块 1 中又声

明了同名变量并赋初值 7。第一次输出的结果是 7,这是因为具有局部作用域的变量把具有文件作用域的变量隐藏了,也就是具有文件作用域的变量变得不可见(这是下面要讨论的可见性问题)。当程序运行到块 1 结束后,进行第二次输出时,输出的就是具有文件作用域的变量的值 5。

具有文件作用域的变量也称为全局变量。

5.1.2　可见性

现在,让我们从标识符引用的角度,来看标识符的有效范围,即标识符的可见性。**程序运行到某一点,能够引用到的标识符,就是该处可见的标识符**。为了理解可见性,先来看一看不同作用域之间的关系。文件作用域最大,接下来依次是类作用域和局部作用域。图 5-1 描述了作用域的一般关系。可见性表示从内层作用域向外层作用域"看"时能看到什么。因此,可见性和作用域之间有着密切的关系。

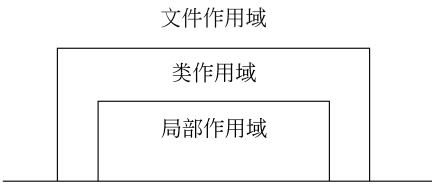


图 5-1　作用域关系图

作用域可见性的一般规则是:

- 标识符要声明在前,引用在后。
- 在同一作用域中,不能声明同名的标识符。
- 在没有互相包含关系的不同的作用域中声明的同名标识符,互不影响。
- 如果在两个或多个具有包含关系的作用域中声明了同名标识符,则外层标识符在内层不可见。

再看一下例 5-1,这是文件作用域与块作用域相互包含的实例,在主函数内块 1 之外,可以引用具有文件作用域的变量,也就是说它是可见的。当程序运行进入块 1 后,就只能引用具有局部作用域的同名变量,具有文件作用域的同名变量被隐藏了。

提示　作用域和可见性的原则不只适用于变量名,也适用于其他各种标识符,包括常量名、用户定义的类型名、函数名、枚举类型的取值等。

5.2　对象的生存期

对象(包括简单变量)都有诞生和消失的时刻。对象从诞生到结束的这段时间就是它的生存期。在生存期内,对象将保持它状态(即数据成员的值),变量也将保持它的值不变,直到它们被更新为止。本节,使用对象来统一表示类的对象和一般的变量。对象的生存期可以分为静态生存期和动态生存期两种。

5.2.1　静态生存期

如果对象的生存期与程序的运行期相同,我们称它具有静态生存期。在文件作用域中声明的对象都是具有静态生存期的。如果要在函数内部的局部作用域中声明具有静态生存期的对象,则要使用关键字 **static**,例如下列语句定义的变量 i 便是具有静态生存期的变量,也称为静态变量:

```
static int i;
```

局部作用域中静态变量的特点是,它并不会随着每次函数调用而产生一个副本,也不会随着函数返回而失效,也就是说,当一个函数返回后,下一次再调用时,该变量还会保持上一回的值,即使发生了递归调用,也不会为该变量建立新的副本,该变量会在各次调用间共享。

在定义静态变量的同时也可以为它赋初值,例如:

```
static int i=5;
```

这表示 i 会被以 5 初始化,而非每次执行函数时都将 i 赋值为 5。

类的数据成员也可以用 static 修饰,本章 5.3 节将专门讨论类的静态成员。

细节 定义时未指定初值的基本类型静态生存期变量,会被以 0 值初始化,而对于动态生存期变量,不指定初值意味着初值不确定。

5.2.2 动态生存期

除了上述两种情况,其余的对象都具有动态生存期。在局部作用域中声明的具有动态生存期的对象,习惯上也被称为局部生存期对象。**局部生存期对象诞生于声明点,结束于声明所在的块执行完毕之时**。

提示 类的成员对象也有各自的生存期。不用 static 修饰的成员对象,其生存期都与它们所属对象的生存期保持一致。

例 5-2 变量的生存期与可见性。

```
//5_2.cpp
#include <iostream>
using namespace std;
int i=1;          //i 为全局变量,具有静态生存期

void other() {
    //a, b 为静态局部变量,具有全局寿命,局部可见,只第一次进入函数时被初始化
    static int a=2;
    static int b;
    //c 为局部变量,具有动态生存期,每次进入函数时都初始化
    int c=10;
    a+=2;
    i+=32;
    c+=5;
    cout<<"---OTHER---"<<endl;
    cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c: "<<c<<endl;
    b=a;
}

int main() {
    //a 为静态局部变量,具有全局寿命,局部可见
    static int a;
    //b, c 为局部变量,具有动态生存期
    int b=-10;
```

```
    int c=0;

    cout<<"---MAIN---"<<endl;
    cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c: "<<c<<endl;
    c+=8;
    other();
    cout<<"---MAIN---"<<endl;
    cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c: "<<c<<endl;
    i+=10;
    other();
    return 0;
}
```

运行结果：

```
---MAIN---
i: 1 a: 0 b: -10 c: 0
---OTHER---
i: 33 a: 4 b: 0 c: 15
---MAIN---
i: 33 a: 0 b: -10 c: 8
---OTHER---
i: 75 a: 6 b: 4 c: 15
```

例 5-3　具有静态、动态生存期对象的时钟程序。

这里仍以时钟类的为例，在这个实例中，声明了具有函数原型作用域、局部作用域、类作用域和文件作用域的多个对象，我们来具体分析它们各自的可见性和生存期。

```
//5_3.cpp
#include<iostream>
using namespace std;

class Clock {          //时钟类定义
public:                //外部接口
    Clock();
    void setTime(int newH, int newM, int newS);    //3 个形参均具有函数原型作用域
    void showTime();
private:                                           //私有数据成员
    int hour, minute, second;
};

//时钟类成员函数实现
Clock::Clock() : hour(0), minute(0), second(0) {}      //构造函数

void Clock::setTime(int newH, int newM, int newS) {    //3 个形参均具有局部作用域
    hour=newH;
    minute=newM;
```

```
    second=newS;
}

void Clock::showTime() {
    cout<<hour<<":"<<minute<<":"<<second<<endl;
}

Clock globClock;                    //声明对象 globClock,具有静态生存期,文件作用域
//由默认构造函数初始化为 0:0:0
int main() {                        //主函数
    cout<<"First time output:"<<endl;
    //引用具有文件作用域的对象 globClock:
    globClock.showTime();           //对象的成员函数具有类作用域
    //显示 0:0:0
    globClock.setTime(8,30,30);     //将时间设置为 8:30:30

    Clock myClock(globClock);       //声明具有块作用域的对象 myClock
    //调用复制构造函数,以 globClock 为初始值
    cout<<"Second time output:"<<endl;
    myClock.showTime();             //引用具有块作用域的对象 myClock
    //输出 8:30:30

    return 0;
}
```

运行结果:

```
First time output:
0:0:0
First time output:
8:30:30
```

在这个程序中,包含了具有各种作用域类型的变量和对象,其中时钟类定义中函数成员 setTime 的 3 个形参具有函数原型作用域;setTime 函数定义中的 3 个参数、对象 myClock 具有局部作用域;时钟类的数据、函数成员具有类作用域;对象 globClock 具有文件作用域。在主函数中,这些变量、对象及公有其成员都是可见的。就生存期而言,除了具有文件作用域的对象 globClock 具有静态生存期,与程序的运行期相同外,其余都具有动态生存期。

5.3 类的静态成员

在结构化程序设计中程序模块的基本单位是函数,因此模块间对内存中数据的共享是通过函数与函数之间的数据共享来实现的,其中包括两个途径——参数传递和全局变量。

面向对象的程序设计方法兼顾数据的共享与保护,将数据与操作数据的函数封装在一起,构成集成度更高的模块。类中的数据成员可以被同一类中的任何一个函数访问。这样一方面在类内部的函数之间实现了数据的共享,另一方面这种共享是受限制的,可以设置适

当的访问控制属性，把共享只限制在类的范围之内，对类外来说，类的数据成员仍是隐藏的，达到了共享与隐藏两全。

然而这些还不是数据共享的全部。对象与对象之间也需要共享数据。

静态成员是解决同一个类的不同对象之间数据和函数共享问题的。例如，我们可以抽象出某公司全体雇员的共性，设计如下雇员类：

```
class Employee {
private:
    int empNo;
    int id;
    string name;          //字符串对象,第 6 章详细介绍
    …
    //其他数据成员与函数成员略
}
```

如果需要统计雇员总数，这个数据存放在什么地方呢？若以类外的变量来存储总数，不能实现数据的隐藏。若在类中增加一个数据成员用以存放总数，必然在每个对象中都存储一副本，不仅冗余，而且每个对象分别维护一个“总数”，容易造成数据的不一致性。由于这个数据应该是为 Employee 类的所有对象所共享的，比较理想的方案是类的所有对象共同拥有一个用于存放总数的数据成员，这就是下面要介绍的静态数据成员。

5.3.1　静态数据成员

我们说“一个类的所有对象具有相同的属性”，是指属性的个数、名称、数据类型相同，各个对象的属性值则可以各不相同，这样的属性在面向对象方法中称为“实例属性”，在 C++ 程序中以类的非静态数据成员表示。例如上述 Employee 类中的 empNo、id、name 都是以非静态数据成员表示的实例属性，它们在类的每个对象中都有，这样的实例属性正是每个对象区别于其他对象的特征。

面向对象方法中还有“类属性”的概念。如果某个属性为整个类所共有，不属于任何一个具体对象，则采用 **static** 关键字来声明为静态成员。静态成员在每个类只有一份，由该类的所有对象共同维护和使用，从而实现了同一类的不同对象之间的数据共享。**类属性是描述类的所有对象共同特征的一个数据项，对于任何对象实例，它的属性值是相同的**。简单地说，如果将“类”比作一个工厂，对象是工厂生产出的产品，那么静态成员是存放在工厂中、属于工厂的，而不是属于每个产品的。

静态数据成员具有静态生存期。由于静态数据成员不属于任何一个对象，因此可以通过类名对它进行访问，一般的用法是“**类名::标识符**”。在类的定义中仅仅对静态数据成员进行引用性声明，必须在文件作用域的某个地方使用类名限定进行定义性声明，这时也可以进行初始化。C++ 11 标准支持常量表达式类型修饰（constexpr 或 const）的静态常量在类内初始化，此时仍可在类外定义该静态成员，但不能做再次初始化操作。在 UML 中，静态数据成员是通过在数据成员下方添加下画线来表示。从下面的例子中可以看到静态数据成员的作用。

提示　之所以类的静态数据成员需要在类定义之外再加以定义，是因为需要以这种方

式专门为它们分配空间。非静态数据成员无须以此方式定义,是因为它们的空间是与它们所属对象的空间同时分配的。

例 5-4 具有静态数据成员的 Point 类。

这个程序是由第 4 章的 Point 类修改而来,引入静态数据成员 count 用于统计 Point 类的对象个数。包含静态数据成员 count 的 Point 类的 UML 图形表示如图 5-2 所示。

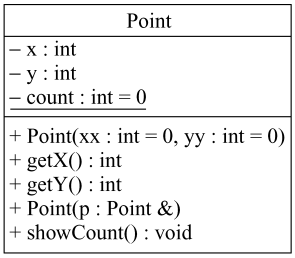


图 5-2 包含静态数据成员的 Point 类的 UML 图

```
//5_4.cpp
#include <iostream>
using namespace std;

class Point {          //Point 类定义
public:                //外部接口
    Point(int x=0, int y=0) : x(x), y(y) {//构造函数
        //在构造函数中对 count 累加,所有对象共同维护同一个 count
        count++;
    }
    Point(Point &p) {                    //复制构造函数
        x=p.x;
        y=p.y;
        count++;
    }
    ~Point() {  count--;}
    int getX() {return x;}
    int getY() {return y;}

    void showCount() {                   //输出静态数据成员
        cout<<"  Object count="<<count<<endl;
    }
private:                                 //私有数据成员
    int x, y;
    static int count;                    //静态数据成员声明,用于记录点的个数
    constexpr static int origin=0;       //常量静态成员类内初始化
};

int Point::count=0;                      //静态数据成员定义和初始化,使用类名限定
constexpr int Point::origin;             //类外定义常量静态成员,但不可二次初始化
int main() {                             //主函数
    Point a(4, 5);                       //定义对象 a,其构造函数会使 count 增 1
    cout<<"Point A: "<<a.getX()<<", "<<a.getY();
    a.showCount();                       //输出对象个数

    Point b(a);                          //定义对象 b,其构造函数会使 count 增 1
    cout<<"Point B: "<<b.getX()<<", "<<b.getY();
```

```
    b.showCount();                          //输出对象个数

    return 0;
}
```

运行结果：

```
Point A: 4, 5  Object count=1
Point B: 4, 5  Object count=2
```

上面的例子中，类 Point 的数据成员 count 被声明为静态，用来给 Point 类的对象计数，每定义一个新对象，count 的值就相应加 1。静态数据成员 count 的定义和初始化在类外进行，初始化时引用的方式也很值得注意，首先应该注意的是要利用类名来引用，其次，虽然这个静态数据成员是私有类型，在这里却可以直接初始化。除了这种特殊场合，在其他地方，例如主函数中就不允许直接访问了。count 的值是在类的构造函数中计算的，a 对象生成时，调用有缺省参数的构造函数，b 对象生成时，调用复制构造函数，两次调用构造函数访问的均是同一个静态成员 count。通过对象 a 和对象 b 分别调用 showCount 函数输出的也是同一个 count 在不同时刻的数值。这样，就实现了 a、b 两个直接的数据共享。

提示　在对类的静态私有数据成员初始化的同时，还可以引用类的其他私有成员。例如，如果一个类 T 存在类型为 T 的静态私有对象，那么可以引用该类的私有构造函数将其初始化。

5.3.2　静态函数成员

在例 5-4 中，函数 showCount 是专门用来输出静态成员 count 的。要输出 count 只能通过 Point 类的某个对象来调用函数 showCount。在所有对象声明之前 count 的值是初始值 0。如何输出这个初始值呢？显然由于尚未声明任何对象，无法通过对象来调用 showCount。由于 count 是为整个类所共有的，不属于任何对象，因此我们自然会希望对 count 的访问也不要通过对象。现在尝试将例 5-4 中的主函数改写如下：

```
int main() {
    Point::showCount();          //直接通过类名调用函数输出对象个数的初始值
    Point a(4,5);
    cout<<"Point A, "<<a.getX()<<", "<<a.getY();
    a.showCount();
    Point b(a);
    cout<<"Point B, "<<b.getX()<<", "<<b.getY();
    b.showCount();
}
```

但是不幸得很，编译时出错了，对普通函数成员的调用必须通过对象名。

尽管如此 C++ 中还是可以有办法实现我们上述期望的，这就是使用静态成员函数。所谓静态成员函数就是使用 static 关键字声明的函数成员，同静态数据成员一样静态成员函数也属于整个类，由同一个类的所有对象共同拥有，为这些对象所共享。

静态成员函数可以通过类名或对象名来调用，而非静态成员函数只能通过对象名来