第 5 章
数据的共享与保护

C++ 语言适合编写大型、复杂的软件系统，当程序的规模变大时，数据的共享与保护显得尤其重要。本章的主要内容包括标识符的作用域、可见性、生存期等重要的变量属性概念；C++ 中类成员的共享与保护的实现机制（主要包括静态成员、友元以及常成员等）；多源程序文件结构与编译预处理等。

【本章学习要求】

理解：对象作用域与可见性的含义；作用域与生存期的区别和联系；存储类型和生存期的关系。

理解：程序模块间数据共享和保护的作用，C++ 工程的多文件结构和编译预处理。

理解：静态成员函数和常成员函数的调用规则。

掌握：静态成员的定义及使用，友元函数和友元类的定义和使用。

掌握：共享数据的保护机制，常对象，常类成员和常引用。

5.1 标识符的作用域与可见性

本节与 5.2 节的内容主要讨论标识符的几个重要属性，即作用域、可见性、生存期等。本节首先介绍作用域与可见性的概念。

作用域表示一个标识符（变量）的有效范围，即起作用的范围；可见性表示一个标识符是否可以被引用，即在编写程序的位置是否可以看到该标识符。两者之间既相互联系，又有很大的区别。

5.1.1 作用域

C++ 语言中的标识符作用域主要有局部作用域（包括函数原型作用域、块作用域），类作用域和全局作用域（命名空间）。标识符的作用域属性描述了在程序文件中一个标识符的有效区域。

1. 函数原型作用域

函数原型作用域是 C++ 程序中范围最小的作用域，仅存在于函数原型声明的括号中。例如：

```
double getArea(double radius);
```

上例声明了一个计算圆面积的函数原型，其形参 radius 是 double 类型，表示需要计算

面积的圆的半径，其作用范围起止于形参列表的左右括号之间。程序的其他任何地方不可以引用该标识符，其目的只是增加程序的可读性，可以省略不写。

上例的声明也可这样进行：

```
double getArea(double);
```

2. 块作用域

程序块一般指的是具有独立功能的小程序片段，如一个分支语句、一个循环结构的结构体或者一个函数体等，一般用“{”与“}”括起来，有时候也可以人为地使用一对花括号将一段代码括起来构成程序块。

在程序块中声明的标识符只能在该块内起作用，例如：

```
void fun(int x){
      int y = x;
      cin>>y;
      if(y>0){
                int z;
                …
      }
            …
}
```

上例是一个函数的定义，共声明了 3 个变量 x、y、z。其中，x 是函数的形参，可以在函数体中使用（形参 x 的作用域非函数原型作用域，注意函数的定义与声明的区别）；y 是在函数体内声明的局部变量，其作用域是从声明的位置开始，直到函数块结束为止（函数结束的右花括号为标志）；变量 z 也是一个局部变量，是在 if 语句后面的块中声明的。变量 z 的作用域从声明的位置开始，到 if 语句块结束为止，即使 if 语句后面还有函数体的其他语句，但是变量 z 已经不起作用了。

3. 类作用域

从作用域的角度可以把类理解为一组数据成员与函数成员的集合。类中的成员都具有类作用域，表示这些成员是属于该类的，因此在访问类的成员时需要通过类名限定。例如，类 X 有一个成员名称为 M，则程序中访问 M 的方式有以下几种情况。

（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，那么在该成员函数内可以直接访问成员 M。

（2）通过表达式 x.M 或者 X::M 访问。其中 x 是类 X 的对象，通过这种方式可访问类的静态成员。

（3）通过表达式 ptr->M 访问，其中 ptr 是指向类 X 的某个对象的指针。

有关类的静态成员以及指向类对象的指针及对象成员的指针等概念，后续章节会详细介绍。

4. 命名空间作用域

与局部作用域相对应的是文件作用域，也称为全局作用域，即在程序文件中全局位置声明的标识符在整个文件中都可以引用。定义在所有函数之外的标识符具有文件作用域，作用域从定义处开始到整个源文件结束。文件中定义的全局变量和函数都具有文件作用域。如果某个文件中说明了具有文件作用域的标识符，该文件又被另一个文件包含，则该标识符

的作用域将会延伸到新的文件中。

但是，如果一个大型的程序由多个程序文件(模块)构成，且这些程序文件中有同名的标识符，此时引用标识符会产生冲突，为此先介绍命名空间的概念。

C++ 标准中引入命名空间的概念，是为了解决不同模块或者函数库中相同标识符冲突的问题。有了命名空间的概念，标识符就被限制在特定的范围(函数)内，不会引起命名冲突。最典型的例子就是标准命名空间，即 std 命名空间。C++ 标准库中所有标识符都包含在该命名空间中。本书之前的例程中都使用语句 using namespace std，表示采用标准的命名空间。

如果确信在程序中引用某个标识符或者某些程序库不会引起命名冲突(即库中的标识符不会在程序中代表其他函数名称)，那么可以通过 using 操作符来简化对程序库中标识符(通常是函数)的使用，例如：

```
#include<iostream>
using namespace std;
void main()
{
    cout<< "hello!"<<endl;
}
```

如果不用"using namespace std;"，那么用如下语句实现输出：

```
std::cout << "hello!"<<std::endl;
```

注意：<iostream>和<iostream.h>之间的区别。前者没有后缀，实际上，在编译器 include 文件夹里，二者是两个文件，里面的代码是不一样的。对于后缀为.h 的头文件，C++ 标准已经明确提出不支持了，早些时候的实现将标准库功能定义在全局空间里，声明在带.h 后缀的头文件里。C++ 标准为了和 C 区别，也为了正确使用命名空间，规定头文件不使用后缀.h。因此，当使用<iostream.h>时，相当于在 C 中调用库函数，使用的是全局命名空间，也就是早期的 C++ 实现；当使用<iostream>的时候，该头文件没有定义全局命名空间，必须使用 namespace std；这样才能正确使用 cin、cout 等输入输出流的功能。实际上，C++ 标准程序库的所有标识符都被声明在标准命名空间，即 std 命名空间内，如 cin、cout、endl 等标识符。

【例 5.1】 标识符作用域示例。

```
#include<iostream>
using namespace std;
int x;                              //在全局命名空间中声明的全局变量
namespace ns1{
  int y;                            //在 ns1 命名空间中声明的全局变量
}
int main(void){
   x = 5;                           //为全局变量 x 赋值
   ns1::y = 6;                      //为全局变量 y 赋值
{
using namespace ns1;                //表示在当前块中可以引用 ns1 空间的标识符
   int x;                           //声明局部变量 x
   x = 7;
   cout<<"x = "<<x<<endl;           //输出为 7
```

```
    cout<<"y = "<<y<<endl;        //输出为 6
    return 1;
}
cout<<"x = "<<x<<endl;            //输出为 5
}
```

运行结果：

```
x = 7
y = 6
x = 5
Press any key to continue
```

5.1.2　可见性

标识符的可见性是从引用该标识符的角度考察一个变量在当前位置能否被使用。如果标识符在某个位置可见，则表示此时可访问该标识符的值，该标识符一定在其作用域内，但有时候标识符尽管在其作用域内，也不可见，即不能够引用。

【例 5.2】 显示同名变量可见性的例子。

```
#include<iostream>
using namespace std;
int n = 100;
int main()
{
  int i = 200,j = 300;
  cout<<n<<'\t'<<i<<'\t'<<j<<endl;      //输出全局变量 n 和外层局部变量 i、j
  {
    int i = 500,j = 600,n;              //内层块
    n = i+j;
    cout<<n<<'\t'<<i<<'\t'<<j<<endl;    //输出内层局部变量 n 和 i、j
    cout<<::n<<endl;                    //输出全局变量 n
  }
  n = i+j;                              //修改全局变量
  cout<<n<<'\t'<<i<<'\t'<<j<<endl;      //输出全局变量 n 和外层局部变量 i、j
  return 0;
}
```

运行结果：

```
100     200     300
1100    500     600
100
500     200     300
Press any key to continue
```

上例在局部块中声明了与全局变量 i、j 同名的变量，因此在局部块中尽管全局变量 i、j 在其作用域内，但访问的变量 i、j 却是局部块中声明的，即全局变量 i、j 在局部块中不可见，被同名的局部变量屏蔽了。

作用域与可见性的特征不仅适用于变量，也适用于常量、用户定义类型名、函数名以及枚举类型等。

5.2 对象的存储类型与生存期

1. 对象的存储类型

存储类型决定了变量的生命期。变量生命期是指从获得内存空间到释放空间经历的时间。存储类型的说明符有 4 个：auto、register、static 和 extern。前两者称为“自动”类型，后两者分别为“静态”和“外部”类型。

(1) 自动存储类型。

自动存储类型包括自动变量和寄存器变量两种。

自动变量：用 auto 说明的变量，通常省略 auto。前面提到的局部变量都是自动类型，其生命期开始于块的执行，结束于块的结束。自动变量的空间分配在栈中，在程序运行过程中，块开始执行时系统自动分配空间(未初始化时值为随机数)，块执行结束时系统自动释放空间。因此，自动变量的生命期和作用域是一致的。

寄存器变量：说明时用 register 修饰，如“register int i;”。系统将这样说明的变量尽可能保存在寄存器中，以提高程序的运行速度。但不同的编译器对哪些变量可以说明为寄存器变量有不同的规定，而且一般的编译器都会对寄存器的使用进行优化，因此，不提倡使用寄存器变量。

(2) 静态存储类型。

使用 static 关键字声明的变量称为“静态变量”。静态变量均存储在全局数据区，如果程序未显式给出初始化值，系统自动初始化为全 0，且初始化只进行一次。

静态变量占有的空间要到整个程序执行结束才释放，故静态变量具有全局生命期。根据定义的位置不同，还分为“局部静态变量”和“全局静态变量”，也称为“内部静态变量”和“外部静态变量”。其中，局部静态变量是定义在块中的静态变量，当块第一次被执行时，编译器在全局数据区为其开辟空间并保存数据，该空间一直到整个程序结束才释放。该变量具有局部作用域，但却具有全局生命期。

【例 5.3】 自动变量与局部静态变量的区别。

```
#include<iostream>
using namespace std;
st();
at();
int main() {
   int i;
   for(i = 0;i<5;i++)  cout<<at()<<'\t';
   cout<<endl;
   for(i = 0;i<5;i++)  cout<<st()<<'\t';
   cout<<endl;
   return 0;
}
st() {
   static int t = 100;          //局部静态变量
   t++;
   return t;
}
at() {
```

```
    int t = 100;                    //自动变量
    t++;
    return t;
}
```

请读者运行上例程序，分析程序的输出结果。

(3) 外部存储类型。

外部存储类型是用 extern 关键字声明的变量。在程序文件中定义的全局变量和函数默认为外部的，其作用域可以延伸到程序的其他文件中。

一个 C++ 程序可以由多个源程序文件组成。多文件程序系统可以通过外部存储类型的变量和函数来共享某些数据和操作。其方法是：其他文件如果要使用某个文件中定义的全局变量和函数，应该在使用前用 extern 做外部声明，表示该全局变量或函数不是在本文件中定义的。

外部声明通常放在文件的开头(函数总是省略 extern)。

在同一个文件中，如果函数使用到定义在该函数之后的全局变量，就必须对使用到的全局变量进行外部变量声明，以满足先定义后使用的原则。因此，全局变量最好集中定义在文件的起始部分。

外部变量声明不同于全局变量定义。变量定义时，编译器为其分配内存空间，而变量声明则表示该全局变量已在其他地方定义过，编译器不再分配内存空间，直接使用变量定义时所分配的空间。因此，所声明变量的变量名和类型必须与定义的完全相同。

2. 标识符的生存期

生命期(life time)也叫作生存期。生命期与存储区域相关，C++ 编译的程序占用的内存分为以下几个存储区域：栈区，由编译器自动分配释放，存放函数的参数值、局部变量的值等，其操作方式类似于数据结构中的栈；堆区，一般由程序员分配释放，若程序员不释放，程序结束时可能由操作系统回收；全局区(静态区)，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域，程序结束后由系统释放；文字常量区，常量字符串就是放在这里的，程序结束后由系统释放该区域内存；程序代码区，存放函数体的二进制代码。相应地，标识符的生命期分为静态生命期、局部生命期和动态生命期。

(1) 静态生命期。

静态生命期指的是标识符从程序开始运行时存在，即具有存储空间，到程序运行结束时消亡，即释放存储空间。具有静态生命期的标识符存放在静态数据区，属于静态存储类型，如全局变量、静态全局变量、静态局部变量。

具有静态生命期的标识符在未被用户初始化的情况下，系统会自动将其初始化为全 0。函数驻留在代码区，也具有静态生命期。所有具有文件作用域的标识符都具有静态生命期。

(2) 局部生命期。

在函数内部或块中定义的标识符具有局部生命期，其生命期开始于执行到该函数或块的标识符声明处，结束于该函数或块的结束处。

具有局部生命期的标识符存放在栈区。具有局部生命期的标识符如果未被初始化，其内容是随机的、不可用的。具有局部生命期的标识符必定具有局部作用域；但静态局部变量具有局部作用域，却具有静态生命期。

(3) 动态生命期。

具有动态生命期的标识符由特定的函数调用或运算来创建和释放,如调用 malloc()或用 new 运算符为变量分配存储空间时,变量的生命期开始;而调用 free()或用 delete 运算符释放空间或程序结束时,变量生命期结束。

具有动态生命期的变量存放在堆区。关于 new 运算和 delete 运算将在第 6 章中介绍。

【例 5.4】 不同内存存储区域示例。

```
#include<iostream>
using namespace std;
int a = 0;                          //全局初始化区
char *p1;                           //全局未初始化区
void main(void) {
   int b;                           //栈区
   char s[] = "abc";                //栈区
   char *p2;                        //栈区
   char *p3 = "123456";             //123456在常量区,p3在栈上
   static int c =0;                 //全局(静态)初始化区
   p1 = (char *)malloc(10);
   p2 = (char *)malloc(20);         //分配得来的10和20字节的区域就在堆区
   strcpy(p1, "123456");            //123456放在常量区,编译器可能会将它与p3所指向的
                                    //"123456"优化成一个地方
}
```

5.3 类的静态成员

本节主要讨论通过类的静态成员来实现数据的共享。实现数据共享有许多方法,设置全局性的变量或对象是一种方法。但是,全局变量或对象是有局限性的。类的静态成员的提出是为了解决同类对象之间的数据共享问题。

考虑如下学生类中,如果要统计当前学生总数 totalNumber,这个数据应该存放在什么地方合适? 如果将该数据放在类外部的全局变量,则无法实现数据的隐藏;若在类中增加一个数据成员存放学生总数信息,则每个学生对象的存储空间都会存在这样一个数据副本,不仅冗余,还会造成数据的不一致性,给数据维护带来不便。

```
class Student{
   private:
     int StuNo;
     char *name;
     …
     int totalNumber;
     //其他成员略
}
```

实际上学生总数 totalNumber 应该是学生类 Student 的所有对象所共享的,比较理想的方案是类的所有对象共享这一数据,程序运行时内存中只允许存在一个副本。

5.3.1 静态数据成员

首先需要明确一个原则:类的什么样的数据成员可以定义为静态数据成员? 通过前面

章节的学习，我们了解到类的数据成员描述的是该类所有对象共有的特征，如学生类中的数据成员学号、姓名等。每个学生对象都具有这样的属性，这些数据成员在每个学生对象的内存空间都有一个副本，用于区分每个不同学生对象的状态。但对象学生总数 totalNumber 这样的数据成员，其描述的特征不属于某个具体的学生对象，而是描述整个学生类别的特征。对于某个学生对象来说，学生总数是无意义的，totalNumber 这一成员只用来描述整个学生类的所有对象的个数，即学生总数。这样的属性是专门用于描述类别而非用于描述具体类对象的。

类中用于描述类的属性的数据成员，定义时通过使用 static 关键字区分，表示这是一个类的静态成员。使用静态数据成员可以节省内存，因为它是所有对象所公有的，因此，对多个对象来说，静态数据成员只存储在一处，并供所有对象共用。静态数据成员的值对每个对象都是一样，但它的值是可以更新的。只要对静态数据成员的值更新一次，保证所有对象存取更新后的相同的值，这样就可以提高时间效率。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了引用性声明，还必须在类外部进行初始化。

静态数据成员的使用方法和注意事项如下。

(1) 静态数据成员在定义或说明时前面加关键字 static。

(2) 静态成员初始化与一般数据成员初始化不同。静态数据成员初始化需要在全局作用域范围使用如下格式进行：

```
<数据类型><类名>::<静态数据成员名> = <值>
```

(3) 静态数据成员是静态存储的，它是静态生存期，必须对它进行初始化。

(4) 引用静态数据成员时，采用如下格式：

```
<类名>::<静态成员名>
```

【例 5.5】 具有静态成员的 Point 类。

```
#include<iostream>
using namespace std;
class Point {
   public:
       Point(int xx = 0, int yy = 0) {
          X = xx;
          Y = yy;
          countP++;
     }
   ~Point() {
     countP--;
}
Point(Point &p);
int GetX() {return X;}
int GetY() {return Y;}
void GetCount() {cout<<" countP = "<<countP<<endl;}
private:
   int X,Y;
   static int countP;              //此处不可赋值
};
Point::Point(Point &p)
```

```
{
   X = p.X;
   Y = p.Y;
   countP++;
}
int Point::countP = 0;          //在文件作用域内初始化
void main()
{
   Point A(4,5);
   cout<<"Point A,"<<A.GetX()<<","<<A.GetY();
   A.GetCount();
   Point B(A);
   cout<<"Point B,"<<B.GetX()<<","<<B.GetY();
   B.GetCount();
}
```

运行结果：

```
Point A,4,5 countP = 1
Point B,4,5 countP = 2
Press any key to continue
```

例 5.5 中 Point 类有一个静态数据成员名为 countP，表示当前系统中当前点对象的个数。因此在 Point 类的构造函数与拷贝构造函数中都将该静态成员值加 1。因为构造函数与拷贝构造函数被调用时系统中点对象个数会增加，相应地，在 Point 类的析构函数中将该静态成员值减 1，表示系统中点对象的个数减 1。注意在主程序中，不管构造多少点对象，系统中只有一个 countP 的副本，Point 类的所有对象共享这一静态数据。

5.3.2 静态函数成员

例 5.5 存在一个小小的问题。当需要知道程序运行时系统有多少个点对象时，通过调用成员函数 GetCount 可以获取。但该方法是一个普通的函数成员，必须通过对象名才能够调用。如果当前系统中点对象的个数是零，即还没有构造出任何点对象的时候，则无法通过对象名来访问该方法以获取点对象的个数。那么应该如何解决这个问题呢？

实际上，静态成员 countP 是属于类的属性，与具体的某个对象无关，因此即使系统没有初始化任何对象的时候，也是可以访问静态属性 countP 的。问题在于访问 countP 的程序代码应该放在什么地方合适呢？由于普通的成员函数必须通过对象名才能够访问，那么是否存在一种类的成员函数可以不通过对象名就可以直接访问呢？答案是肯定的，与静态数据成员对应的一种类成员是静态函数成员。静态函数成员在定义的时候加上 static 关键字即可。与静态数据成员一样，静态函数成员也是属于整个类的，可通过类名直接访问静态函数成员。因此可以将访问静态数据成员的代码写在类的静态成员函数中，再通过类名直接访问就可以了。

【例 5.6】 具有静态数据成员与静态函数成员的 Point 类。

```
#include<iostream>
using namespace std;
class Point {
    public:
      Point(int xx = 0, int yy = 0){
```

```
        X = xx;
        Y = yy;
        countP++;
    }
    ~Point() {
      countP--;
    }
    Point(Point &p);
    int GetX() {return X;}
    int GetY() {return Y;}
    static void GetCount() {cout<<" countP = "<<countP<<endl;}
private:
    int X,Y;
    static int countP;          //此处不可赋值
};
Point::Point(Point &p)
{
    X = p.X;
    Y = p.Y;
    countP++;
}
int Point::countP = 0;          //在文件作用域内初始化
void main()
{
    Point::GetCount();
    Point A(4,5);
    cout<<"Point A,"<<A.GetX()<<","<<A.GetY();
    Point::GetCount();
    Point B(A);
    cout<<"Point B,"<<B.GetX()<<","<<B.GetY();
    Point::GetCount();
}
```

运行结果：

```
Object id = 0
Point A,4,5 countP = 1
Point B,4,5 countP = 2
Press any key to continue
```

与例 5.5 相比，本例只是在定义成员函数 GetCount 的时候添加了关键字 static，使其成为类的静态函数成员，这样就可通过类名直接访问静态成员函数，并可以不用依赖任何对象直接访问类的静态数据成员。

下面列出几点有关类的静态数据成员与静态成员函数比较容易出错的地方，请大家通过实验验证。

(1) 不能通过类名来调用类的非静态成员函数，例如：

```
void main()
{
    Point::GetCount();
    Point ::GetX();
}
```

编译出错：