# 10.6 类作用域

第9章介绍了全局（文件）作用域和局部（代码块）作用域。可以在全局变量所属文件的任何地方使用它，而局部变量只能在其所属的代码块中使用。函数名称的作用域也可以是全局的，但不能是局部的。C++类引入了一种新的作用域：类作用域。

在类中定义的名称（如类数据成员名和类成员函数名）的作用域都为整个类，作用域为整个类的名称只在该类中是已知的，在类外是不可知的。因此，可以在不同类中使用相同的类成员名而不会引起冲突。例如，Stock类的shares成员不同于JobRide类的shares成员。另外，类作用域意味着不能从外部直接访问类的成员，公有成员函数也是如此。也就是说，要调用公有成员函数，必须通过对象：

Stock sleeper("Exclusive Ore”, 100, 0.25); // 创建对象

sleeper.show{); // 使用对象调用成员函数

show(); // 不可行——无法直接调用方法

同样，在定义成员函数时，必须使用作用域解析运算符：

void Stock::update (double price)

{

…

}

总之，在类声明或成员函数定义中，可以使用未修饰的成员名称（未限定的名称），就像sell()调用set\_tot()成员函数时那样。构造函数名称在被调用时，才能被识别，因为它的名称与类名相同。在其他情况下，使用类成员名时，必须根据上下文使用直接成员运算符（.）、间接成员运算符（->）或作用域解析运算符（::）。下面的代码片段演示了如何访问具有类作用域的标识符：

class Ik

{

private:

int fuse; // fuse具有类作用域

public :

Ik（int f = 9) { fuss = f; } fuse在域中

void ViewIk() const;

};

void Ik::ViewIk() const //Ik::将ViewIk放置在Ik域中

{

cout << fuse << endl; // 在域中的fuse位于类方法之内

}

…

int main()

{

Ik \* pik = new Ik;

Ik ee = Ik(8); // 构造函数在域中，因为它的名称与类名相同

ee.ViewIk(); // 通过类对象使ViewIk在域中

pik->Viewlk(); // 通过指针使ViewIk在域中

…

## 10.6.1 作用域为类的常量

有时候，使符号常量的作用域为类很有用。例如，类声明可能使用字面值30来指定数组的长度，由于该常量对于所有对象来说都是相同的，因此创建一个由所有对象共享的常量是个不错的主意。您可能以为这样做可行：

class Bakery

{

private:

const int Months = 12; // 声明一个常量？错误！

double cost[Months];

…

但这是行不通的，因为声明类只是描述了对象的形式，并没有创建对象。因此，在创建对象前，将没有用于存储值的空间（实际上，C++提供了成员初始化，但不适用于前述数组声明，第12章将介绍该主题）。然而，有两种方式可以实现这个目标，并且效果相同。

第一种方式是在类中声明一个枚举。在类声明中声明的枚举的作用域为整个类，因此可以用枚举为整型常量提供作用域为整个类的符号名称。也就是说，可以这样开始Bakery声明：

class Bakery

{

privace:

enum{Months = 12};

double costs[Months];

…

注意，用这种方式声明枚举并不会创建类数据成员。也就是说，所有对象中都不包含枚举。另外，Months只是一个符号名称，在作用域为整个类的代码中遇到它时，编译器将用12来替换它。

由于这里使用枚举只是为了创建符号常量，并不打算创建枚举类型的变量，因此不需要提供枚举名。顺便说一句，在很多实现中，ios\_base类在其公有部分中完成了类似的工作，诸如ios\_base::fixed等标识符就来自这里。其中，fixed是ios\_base类中定义的典型的枚举量。

C++提供了另一种在类中定义常量的方式——使用关键字static：

class Bakery

{

private:

static const int Months = 12;

double costs[Months];

…

这将创建一个名为Months的常量，该常量将与其他静态变量存储在一起，而不是存储在对象中。因此，只有一个Months常量，被所有Bakery对象共享。第12章将深入介绍静态类成员。在C++98中，只能使用这种技术声明值为整数或枚举的静态常量，而不能存储double常量。C++11消除了这种限制。

## 10.6.2 作用域内枚举（C++11）

传统的枚举存在一些问题，其中之一是两个枚举定义中的枚举量可能发生冲突。假设有一个处理鸡蛋和T恤的项目，其中可能包含类似下面这样的代码：

enum egg {Small, Medium, Large, Jumbo};

enum t\_shirt {Small, Medium, Large, Xlarge};

这将无法通过编译，因为egg Small和t\_shirt Small位于相同的作用域内，它们将发生冲突。为避免这种问题，C++11提供一种新枚举，其枚举量的作用域为类。这种枚举的声明类似于下面这样：

enum claas egg {Small, Medium, Large, Jumbo};

enum class t\_shirt {Small, Medium, Large, Xlarge};

也可使用关键字struct代替class。无论使用哪种方式，都需要使用枚举名来限定枚举量：

egg choice = egg::Large; // egg枚举的Large枚举量

t\_shirt Floyd = t\_shirt::Large; // t\_shirt枚举的Large枚举量

枚举量的作用域为类后，不同枚举定义中的枚举量就不会发生名称冲突了，而您可继续编写处理鸡蛋和T恤的项目。

C++11还提高了作用域内枚举的类型安全。在有些情况下，常规枚举将自动转换为整型，如将其赋给int变量或用于比较表达式时，但作用域内枚举不能隐式地转换为整型：

enum egg\_old(Small, Medium, Large, Jumbo}; // 常规枚举

enum class t\_shirt {Small, Medium, Large, Xlarge}; // 类作用域枚举

egg\_old one = Medium; // 常规枚举

t\_shirt rolf = t\_shirt::Large; //类作用域枚举

int king = one, // 常规枚举会隐式类型转换

int ring = rolf; // 不允许，没有隐式类型转换

if( king < Jumbo ) // 允许

std::cout << "Jumbo converted to int before comparison.\n";

if ( king < t\_shirt::Medium) // 不允许

std::cout << "Not allowed: < not defined for scoped enum.\n";

但在必要时，可进行显式类型转换：

int Frodo = int(t\_shirt：：Small); // Frodo被设置为0

枚举用某种底层整型类型表示，往C++98中，如何选择取决于实现，因此包含枚举的结构的长度可能随系统而异。对于作用域内枚举，C++11消除了这种依赖性。默认情况下，C++11作用域内枚举的底层类型为int。另外，还提供了一种语法，可用于做出不同的选择：

// pizza的底层类型为short

enum class:short pizza {Small, Medium, Large, XLarge};

:short将底层类型指定为short。底层类型必须为整型。在C++11中，也可使用这种语法来指定常规枚举的底层类型，但如果没有指定，编译器选择的底层类型将随实现而异。