華中科技大學

嵌 套 类

许向阳

xuxy@hust.edu.cn

大纲

- 什么是嵌套类？
- 嵌套类与外围类是什么关系？
- 嵌套类与外围类 如何互访？
- 为什么引入嵌套类？
- 如何使用嵌套类？

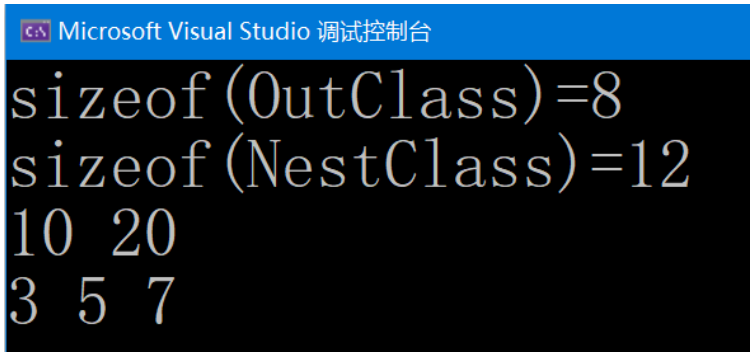
什么是嵌套类？

在一个类的内部，定义的另一个类，称为嵌套类。
亦称为 嵌套类型 （nested class）

```
class OutClass {
private:  int  x;    public: int y;       // 大小 8 个字节
public:
    OutClass(int x, int y) :x(x), y(y)
    { cout << “sizeof(OutClass)=” << sizeof(OutClass) << endl; }  // 8
    void display() {      cout << x << “ ” << y << endl;      }
public:
    class NestClass {    // 嵌套类
        private: int u;     public: int  v, w;   // 大小 12
        public:
            NestClass(int u, int v, int w) :u(u), v(v), w(w)
            {    cout << “sizeof(NestClass)=” << sizeof(NestClass) << endl; }
           void display() {   cout << u << “ ” << v << “ ”<<w<< endl;     }
    };
};
```

嵌套类

```
int main()
{
    OutClass a(10, 20);
    OutClass::NestClass b(3, 5, 7);
    a.display();
    b.display();
}
```



- 嵌套类是一个独立的类，基本上与外围类无关。
- 定义嵌套类对象时，要给出外围类的限定符。
- 嵌套类的成员不属于外围类；外围类的成员也不属于嵌套类。

嵌套类

嵌套类的成员不属于外围类；
外围类的成员也不属于嵌套类。

外围类中：

```
void display() {
      cout << x << " " << y << endl;
      cout << u << endl;        //   u 未申明的标识符
}                               //   u 不是外围类的成员
```

嵌套类中：

```
void display() {
      cout << u << " " << v <<" "<<w<< endl;
      cout << x << endl;        //   x 未申明的标识符
}                               //   x 不是嵌套类的成员
```

嵌套类

嵌套类的访问 受到 定义嵌套类时的权限控制。

```
class  OutClass  { …..
   private:
    class NestClass {
         ……   // 类内申明和定义不变
    };
};
```

OutClass::NestClass b(3, 5, 7); 无法访问 OutClass中的私有类

嵌套类可以定义为public、protected或private，这决定了嵌套类在外部类之外的可见性。如果嵌套类是private的，那么它只能在外部类内部使用，外部类之外无法访问。

嵌套类

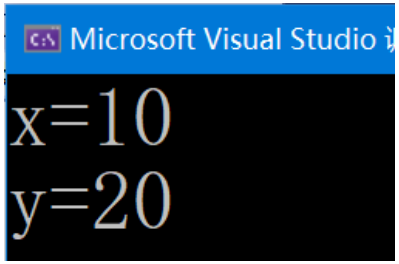
不同的外围类中，可以定义名字一样的嵌套类

```
class A {
public:
    class NestClass {
    public: int x = 10;
            void display() { cout << "x=" << x << endl; }
    };
};
class B {
public:
    class NestClass {
    public: int y = 20;
            void display() { cout << "y=" << y << endl; }
    };
};
```

嵌套类

不同的外围类中，可以定义名字一样的嵌套类

```
int main()
{
    A::NestClass a;
    a.display();
    B::NestClass b;
    b.display();
}
```



- 定义嵌套类的对象时，类型名是 外围类::嵌套类
- 通过对象访问嵌套类的成员，与访问一个独立类的成员没差别

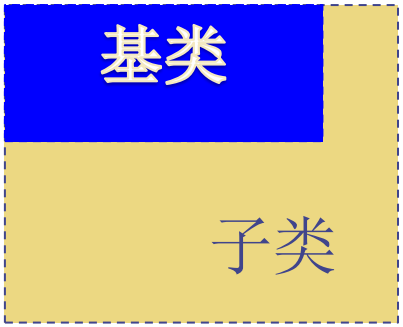
嵌套类

嵌套类成员函数在体外定义：　类型的限定符

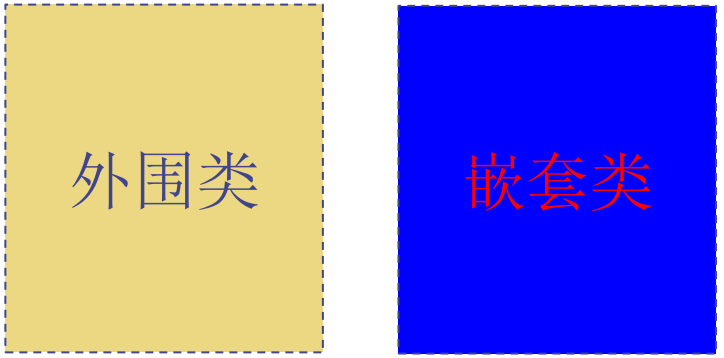
```
class B {
public:
    class NestClass {
    public: int y = 20;
            void display() { cout << "y=" << y << endl; }
            void set(int y);
    };
};
void B::NestClass::set(int y)
{
    this->y = y;
    NestClass::y = y+1;
    B::NestClass::y = y+2;
}
```

嵌套类与外围类的互访

- 嵌套类的成员不属于外围类，外围类的成员也不属于嵌套类。
- 与子类可以直接访问父类成员不同，嵌套类成员函数不能直接访问外围类的成员；外围类成员函数也不能直接访问嵌套类的成员。



指向子类的 this，可转换为基类的this；
子类可直接访问基类成员



两个对象的 this无法转换；
无法直接访问对方成员

嵌套类与外围类的互访

華中科技大學

- 在嵌套类中，可以定义指向外围类的指针；（数据成员）
- 在外围类中可以定义指向嵌套类的指针；（数据成员）
- 在两个类的成员函数中，都可以用指针作为参数，通过参数指针来访问对方的成员。

嵌套类与外围类的互访

- 外围类对嵌套类成员的访问没有任何特权
- 嵌套类可以访问其所在外部类的所有成员，包括私有成员，嵌套类默认是外部类的友元类。

```
void display(const NestClass &n) {    // 外围类的成员函数
        cout << x << " " << y << endl;
        cout << n.u << endl;                  // 不可访问 n 的私有成员 u
}

void display(const OutClass& o) {     // 嵌套类的成员函数
                                      // 参数中有外部类对象
        cout << u << " " << v << " " << w << endl;
        cout << o.x << endl;          //  可访问 o 的私有成员 x
}
```

嵌套类的优点

DeepSeek：C++，定义嵌套类有什么优点？

1、封装性：嵌套类可以隐藏在外围类内部，不暴露给外部，从而实现了更好的封装。如果嵌套类只被外围类使用，那么将其设置为私有（private）可以防止外部代码直接访问，这符合面向对象设计的封装原则。

2、逻辑分组：如果一个类只被另一个类使用，那么将其作为嵌套类放在一起，可以清晰地表示这两个类之间的逻辑关系，使代码结构更清晰。

3、访问权限：嵌套类可以访问外围类的所有成员（包括私有成员），这方便了嵌套类与外围类之间的数据交互，而无需通过额外的公共接口。

嵌套类的优点

DeepSeek：C++，定义嵌套类有什么优点？

4、避免命名污染：将类嵌套在另一个类内部可以避免全局作用域中的类名冲突。嵌套类名在外围类的作用域内，因此可以在不同的外围类中使用相同的嵌套类名而不会冲突。

5、实现细节隐藏：嵌套类通常用于实现外围类的某些功能，但又不希望这些实现细节被外部知道。例如，在实现迭代器模式时，将迭代器作为容器的嵌套类可以隐藏迭代器的实现。

嵌套类的优点

DeepSeek：C++，定义嵌套类有什么优点？

6、紧密关联：当两个类紧密关联，其中一个类是另一个类的重要组成部分时，使用嵌套类可以直观地表示这种关系。

7、辅助类：嵌套类常用于作为外围类的辅助类，例如，在构建器模式（Builder Pattern）中，使用嵌套的构建器类来逐步构建外围类的对象。

8、模板元编程：在模板元编程中，嵌套类可以用于编译时计算和类型操作，例如，通过模板特化在嵌套类中定义不同的类型。

应用示例

在 C++ 实验 （智能下车辅助驾驶）中，Execute 的优化

```
class ExecutorImpl final : public Executor
{  public:
        Pose Query(void) const noexcept override;
        void Execute(const std::string& commands) noexcept;
      ……
      private:
         void  Move(void) noexcep;
         void  TurnLeft(void) noexcept;
         void  TurnRight(void) noexcept;
};
```

按照指令处理逻辑的不同，将M/L/R 3个指令处理的逻辑抽取出Move、TurnLeft、TurnRight三个成员函数。

应用示例

能否提供统一的接口，来使用 Move、TurnLeft、TurnRight?

```
void Move(void) noexcep;
void TurnLeft(void) noexcept;
void TurnRight(void) noexcept;
```

统一接口，函数名、参数要相同！

```
void DoOperate ( )
{  Move( ); }

void DoOperate ( )
{  TurnLeft( ); }

void DoOperate ( )
{  TurnRight( ); }
```

- 三个函数要分属三个不同的类；

  MoveCommand 、**TurnLeftCommand**、**TurnRightCommand**

- 指向这三个类的this，不是指向 Executor (ExecutorImpl),

  因此增加DoOperate 函数的参数，由参数指向 Executor.

应用示例


```
class MoveCommand final {
public:
    void DoOperate(ExecutorImpl& executor) const noexcept
    {  executor . Move( );  }   // 调用 DoOperator的指针，
                                应该是指向 MoveCommand的指针，
};                              但调用Move, 用的指针是executor
```

应用示例

- 定义一个接口类（抽象类），派生出这三个不同的类；
- 定义一个接口类的指针，通过该指针来调用 DoOperate

```
class ICommand
{  public:
     virtual  ~ICommand() = default;
     virtual  void DoOperate(ExecutorImpl& executor) const noexcept = 0;
};

class MoveCommand final : public ICommand
{  public:
     void DoOperate(ExecutorImpl& executor) const noexcept override;
};
```

应用示例


```
Icommand *cmder;
cmder -> DoOperator(executor);
            //  ExecutorImpl& executor
```

cmder 实际指向 MoveCommand、TurnLeftCommand 等

Icommand、 MoveCommand 等都作为 **ExecutorImpl的嵌套类**

应用示例


```
void ExecutorImpl::Execute(const std::string& commands) noexcept
{
    for (const auto cmd : commands) {
        std::unique_ptr<ICommand> cmder;

        if (cmd == 'M') {
            cmder = std::make_unique<MoveCommand>();
        } else if (cmd == 'L') {
            cmder = std::make_unique<TurnLeftCommand>();
        } else if (cmd == 'R') {
            cmder = std::make_unique<TurnRightCommand>();
        }

        if (cmder)          cmder->DoOperate(*this);
    }
}
```

GAME OVER