

# 类和对象

【类声明】

【公有、私有、保护成员】

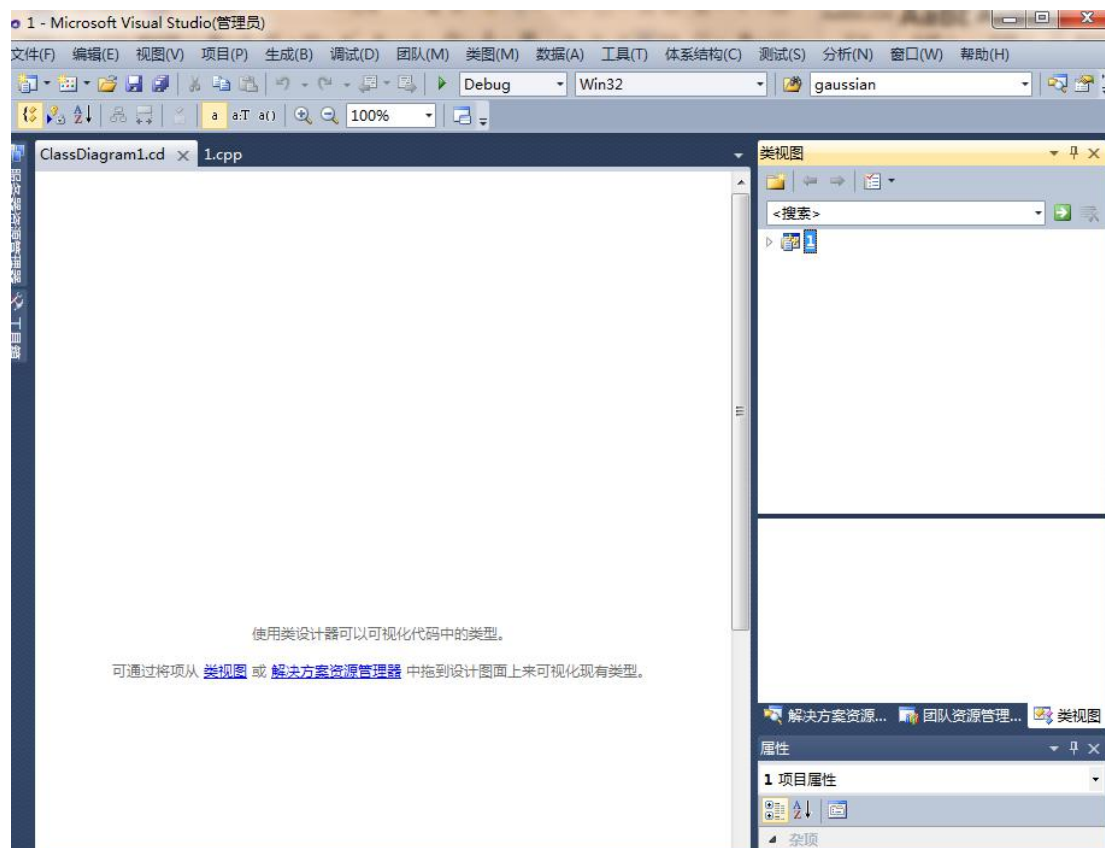
【数据抽象和封装】

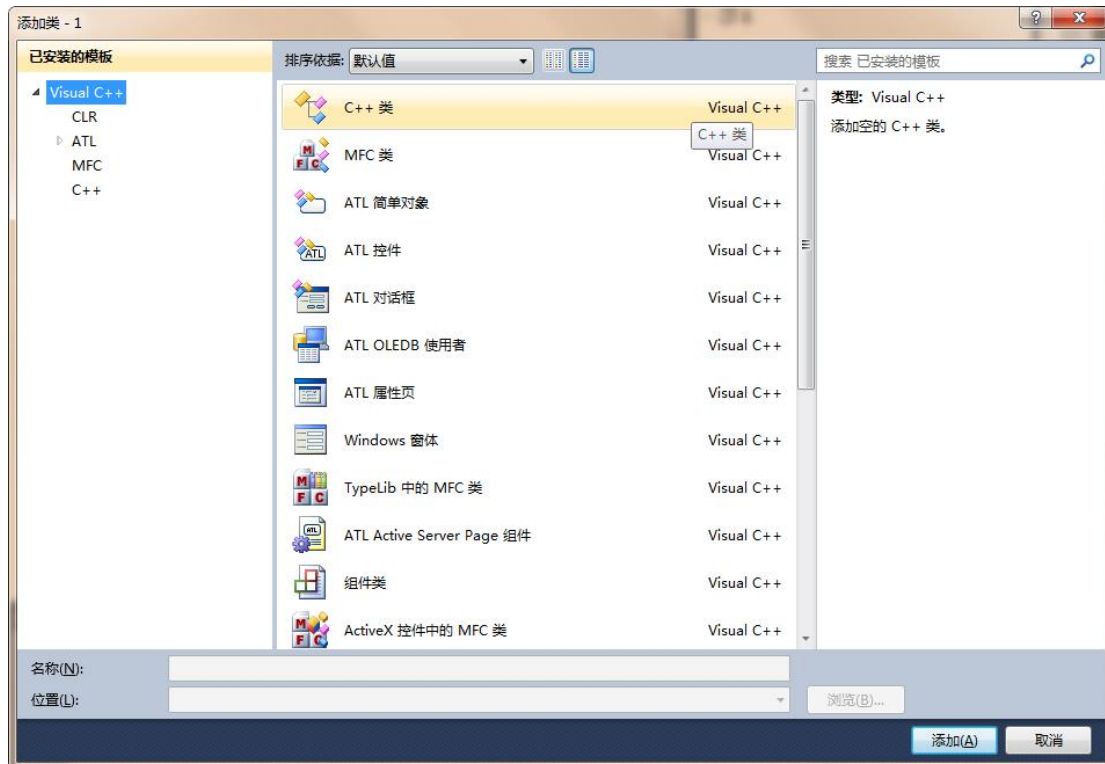
➤ 类的声明：

```
//类是一种用户自定义类型，声明形式：  
class 类名称  
{  
    public:  
        公有成员（外部接口）  
    private:  
        私有成员  
    protected:  
        保护成员  
};
```

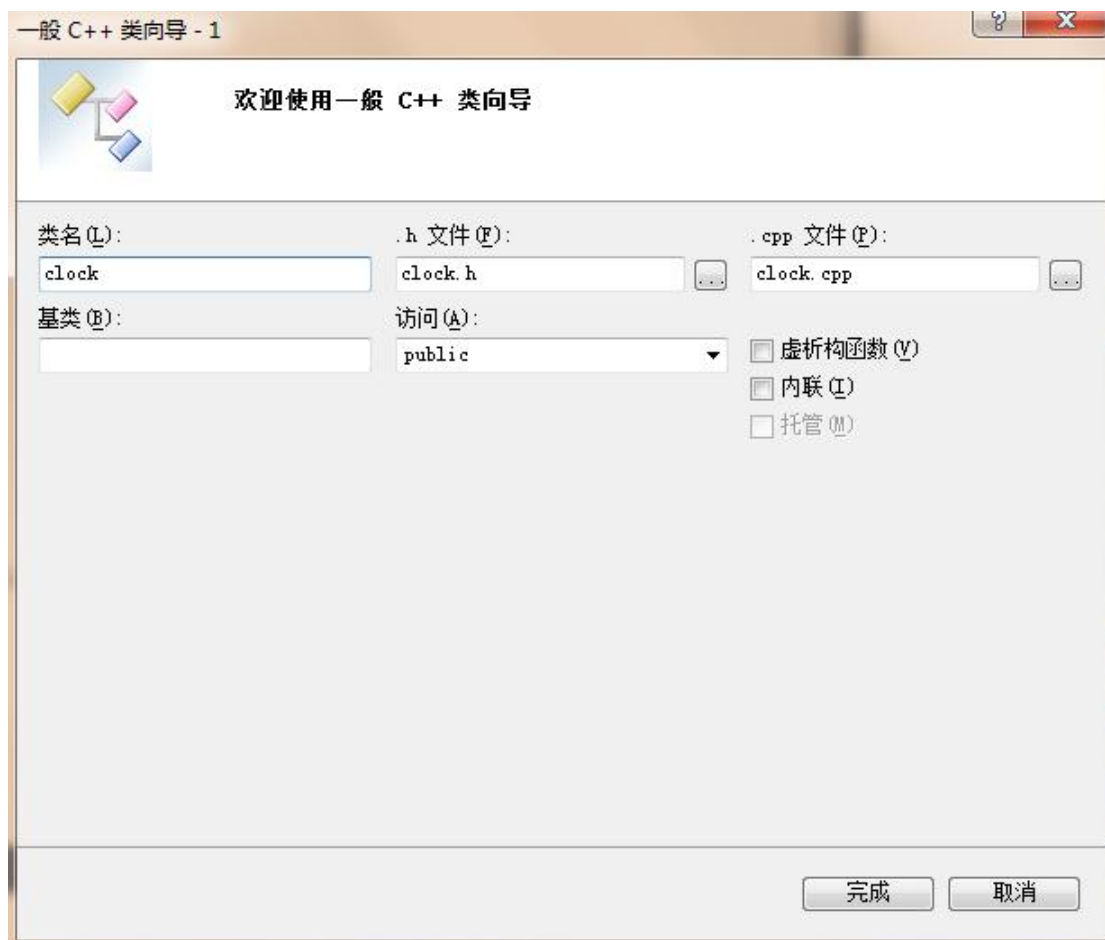
最后这个分号是不能省略的，接下来我们编写一个类：

在工程当中要建立一个类，我们可以借助 **IDE**，在**类视图**当中，添加一个类，



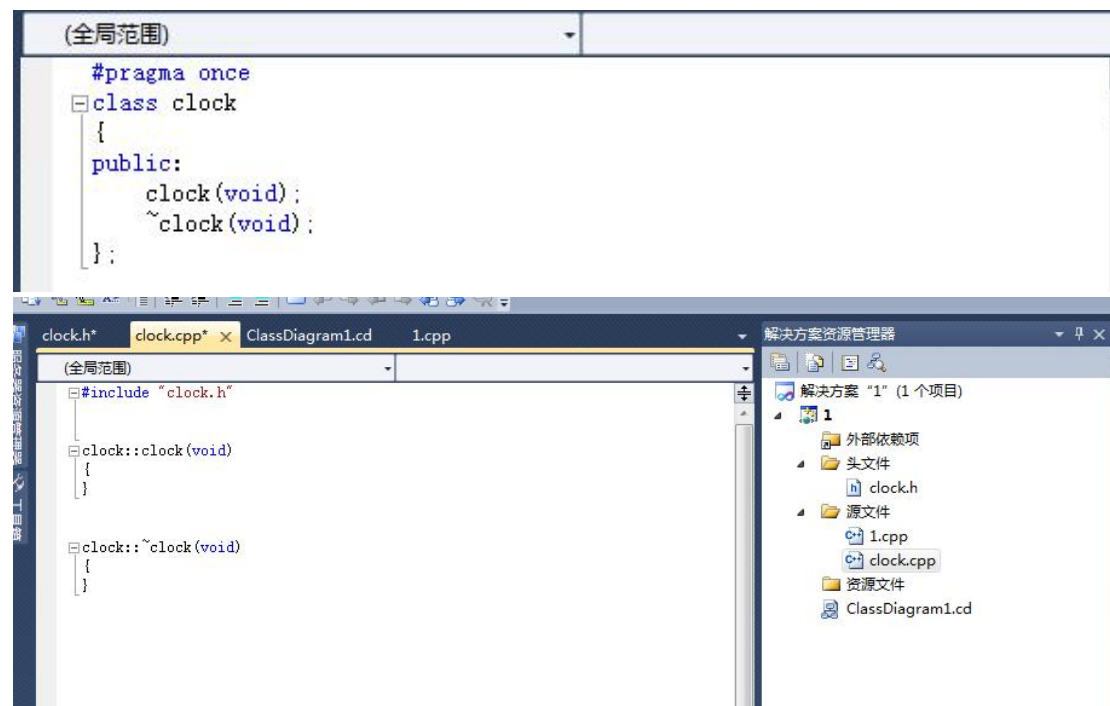


这里我们添加一个时钟类（clock）：

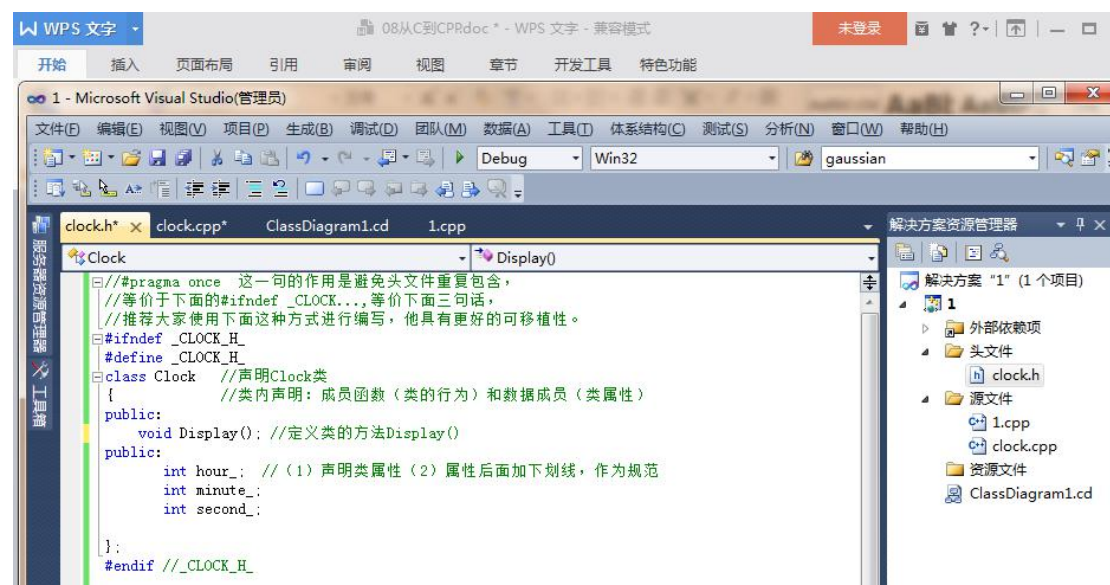


它会产生两个文件， 一个是头文件（.h）一个是实现文件(.cpp)。并且 IDE 还会产生一些额

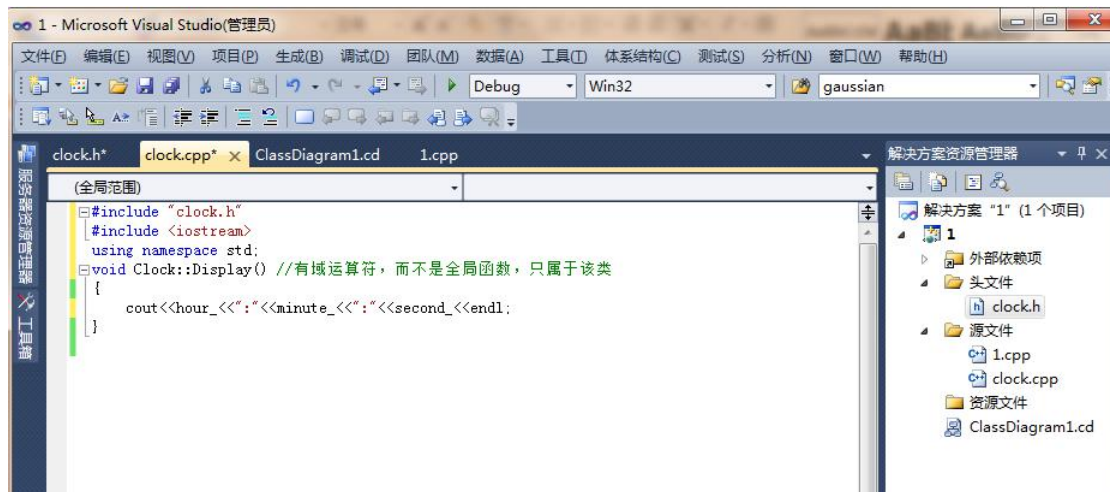
外的代码。



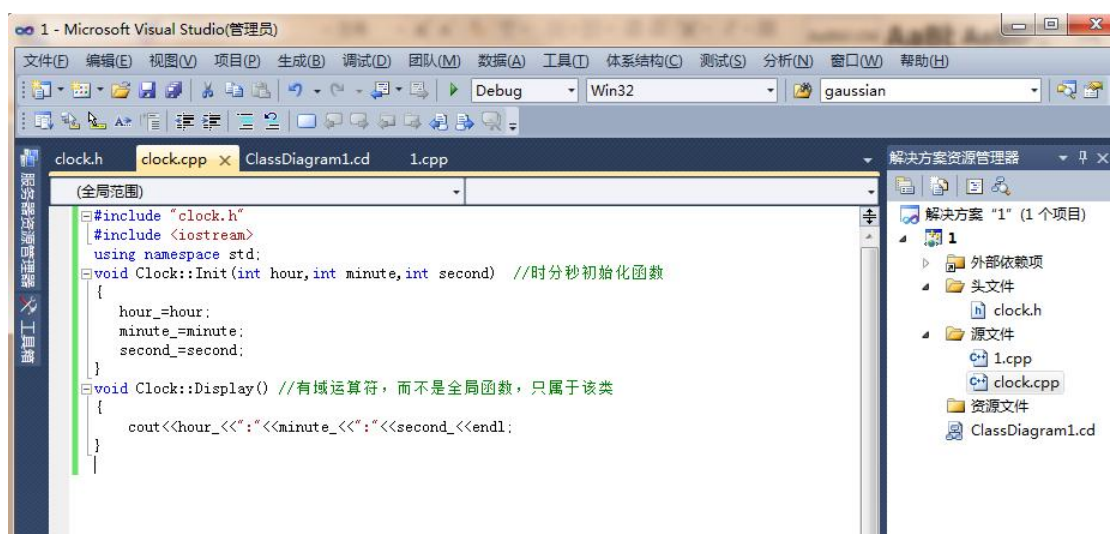
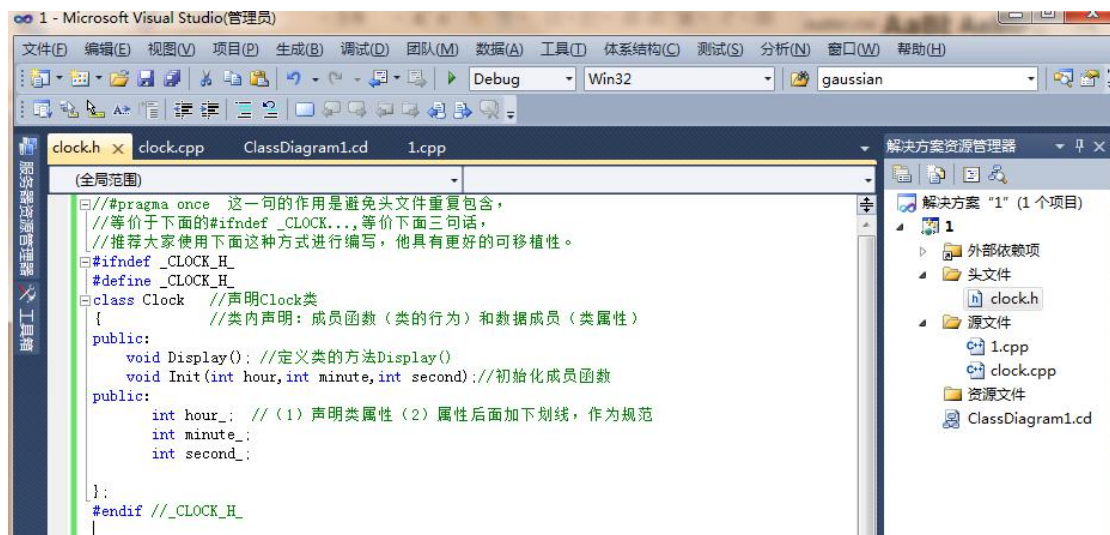
对于初学者来说，这些代码我们都手工进行别写（把自动生成的都删掉），也就是说我们使用**类向导**生成类，仅仅是为了产生这两个文件。当然大家也可以在这两个目录底下创建这两个文件，然后添加到工程当中。编写这个类：



Display 的实现：



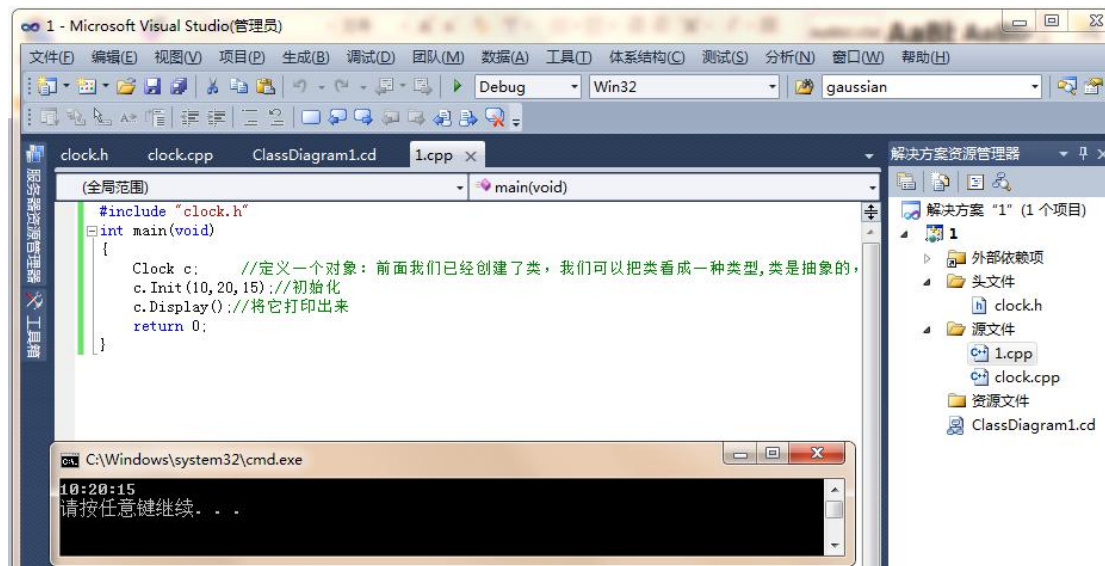
接下来我们编写程序的测试程序，并添加初始化函数 `Init()`:





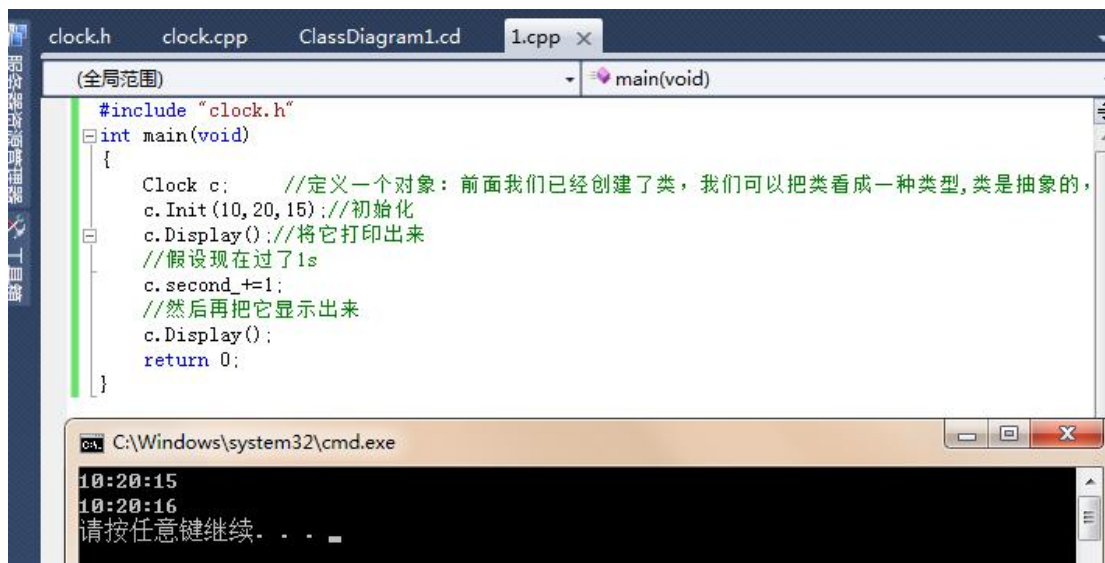


接下来我们运行这个程序：



假设我们过了 1s:





```
clock.h  clock.cpp  ClassDiagram1.cd  1.cpp x
(全局范围)  main(void)

#include "clock.h"
int main(void)
{
    Clock c;    //定义一个对象：前面我们已经创建了类，我们可以把类看成一种类型，类是抽象的，
    c.Init(10, 20, 15); //初始化
    c.Display(); //将它打印出来
    //假设现在过了1s
    c.second_ += 1;
    //然后再把它显示出来
    c.Display();
    return 0;
}

C:\Windows\system32\cmd.exe
10:20:15
10:20:16
请按任意键继续. . .
```

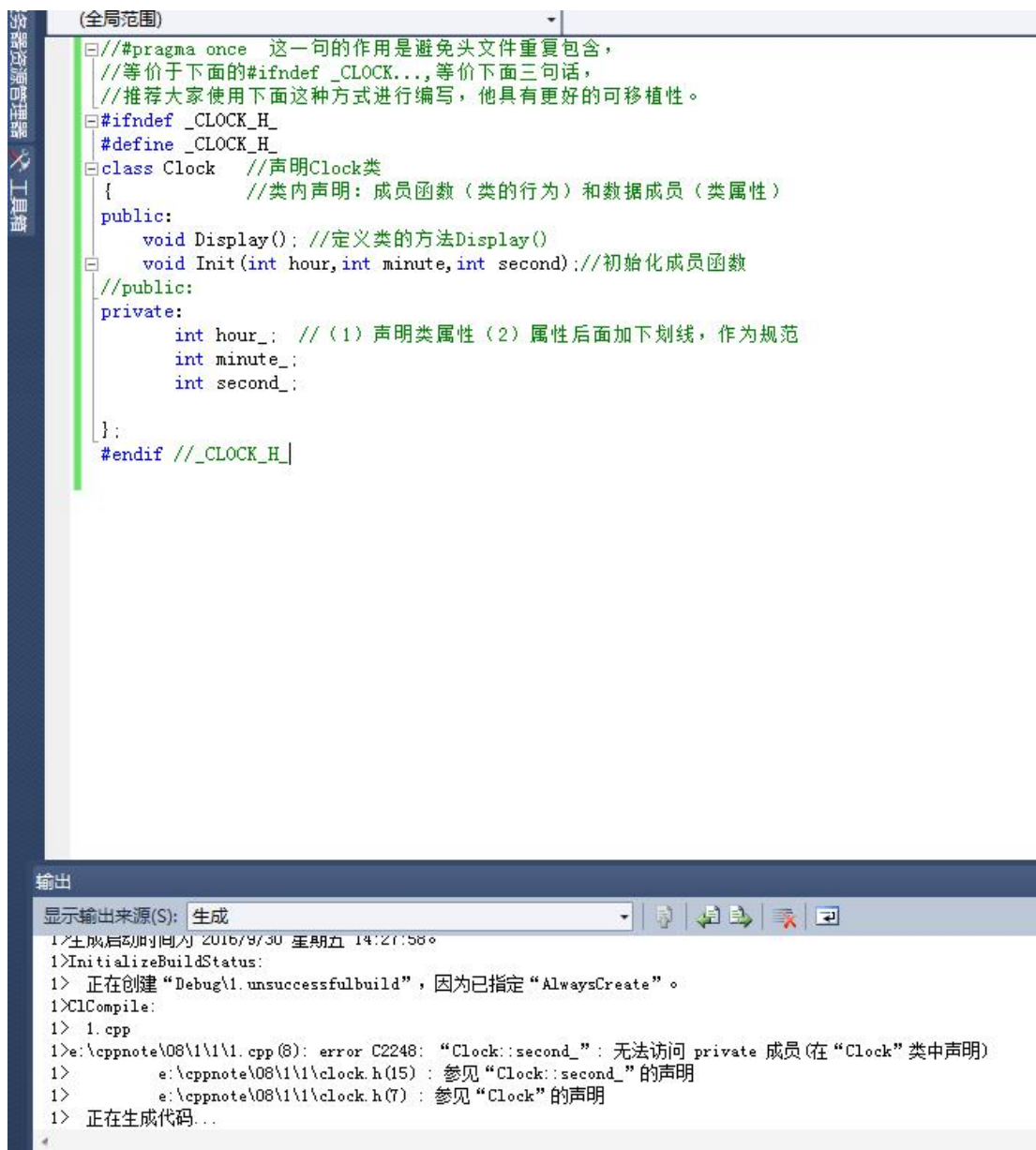
由于我们的 `second_` 是声明在 `public`, 所以说类外部是可以访问的, 接下来我们看一下类的三种访问权限。

### ➤ 公有、私有、保护成员

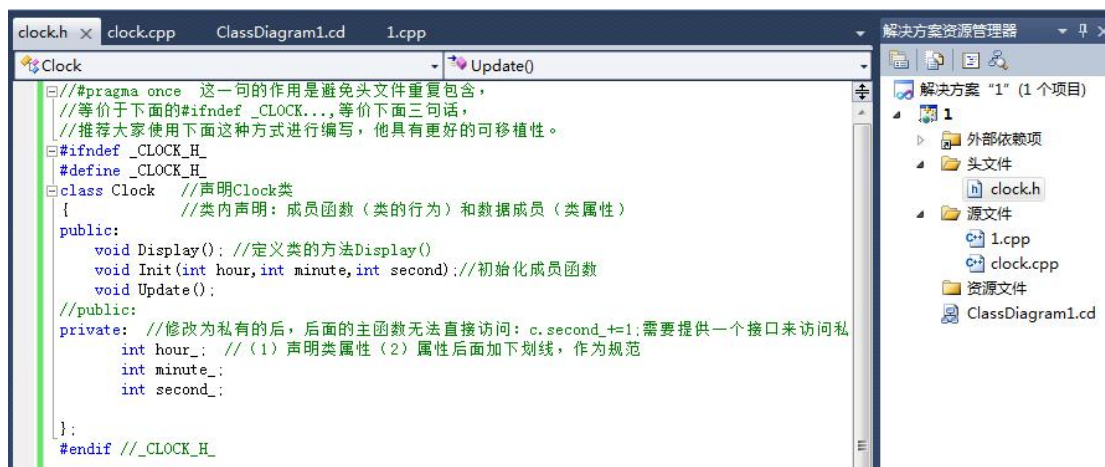
- ❖ 在关键词 **public** 后面声明, 它们是**类和外部的接口**, 任何外部函数都可以访问**公有类型数据**和**函数**。
- ❖ 在关键词 **private** 后面声明, 只允许本类中的函数访问, 而类外部的任何函数都不能访问。
- ❖ 在关键词 **protected** 后面声明, 与 **private** 类似, 其主要的差别表现在继承和派生时对派生类的影响不同。

如果将前面的程序中的 `public` 修改为 `private`, 如下:

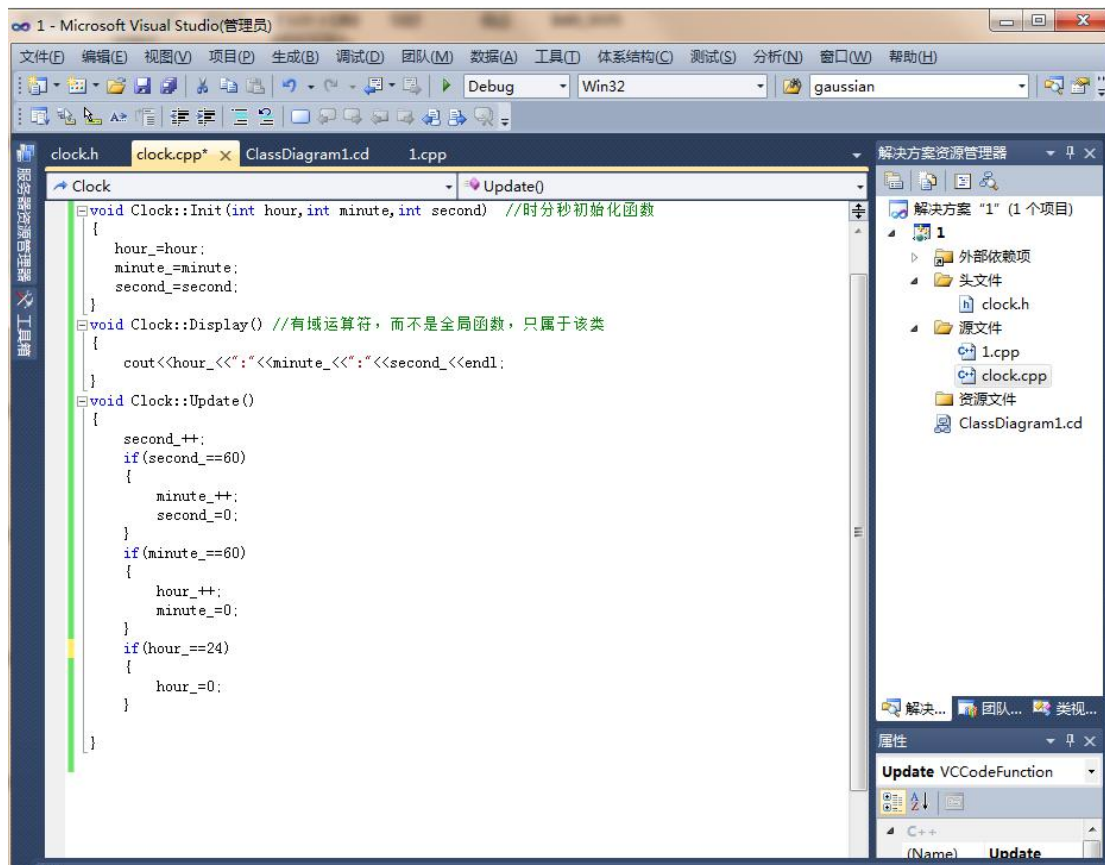
就会出错, 无法访问 `private` 成员。



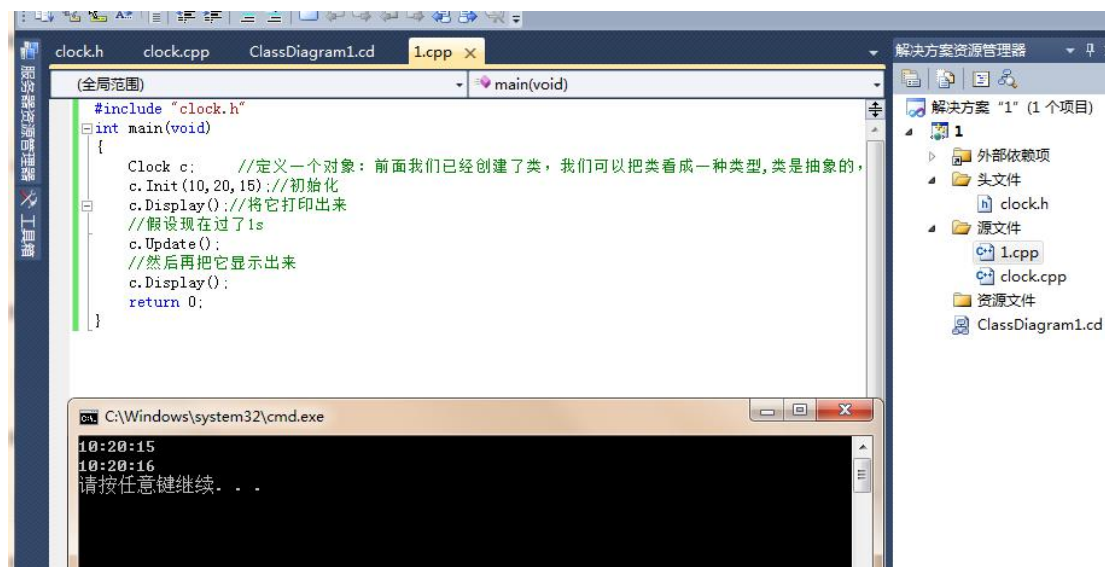
但是可以通过定义接口（函数）来修改这些成员变量：添加接口函数 Update():



接口函数的具体实现:

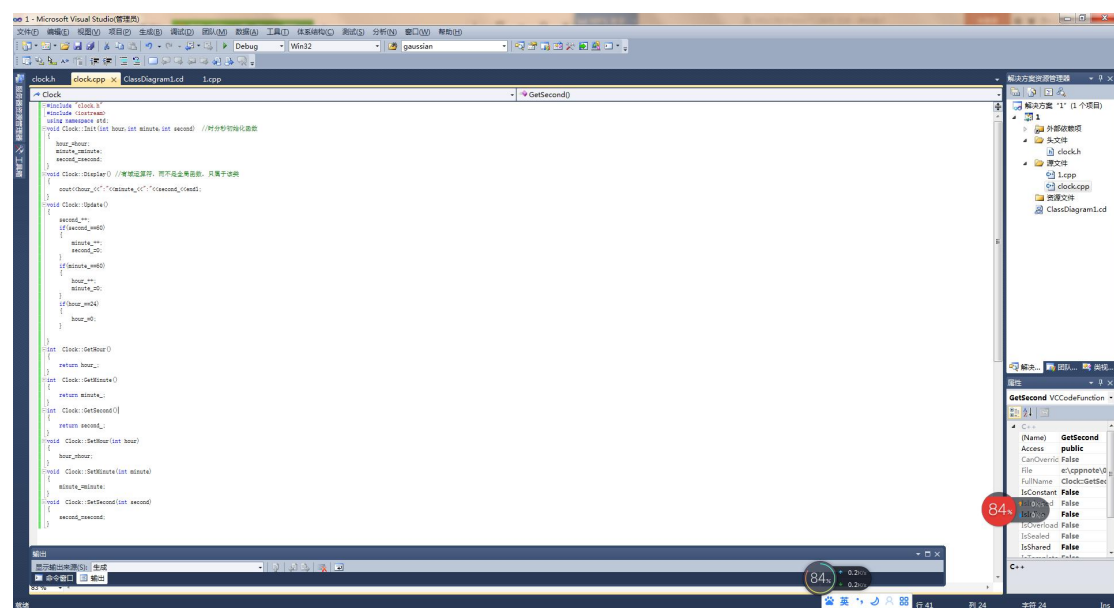


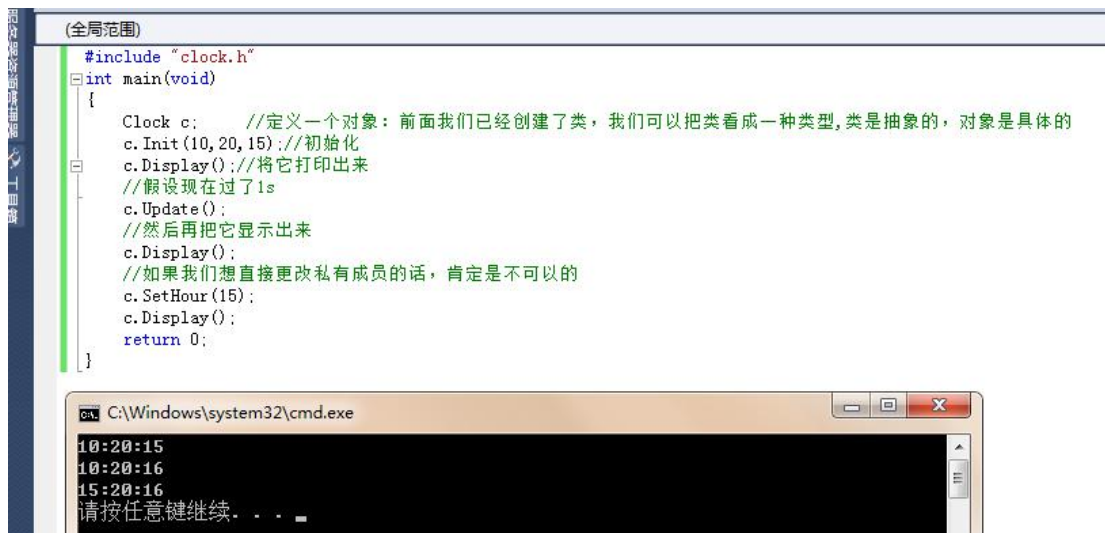
运行结果：



如果想直接访问私有成员，肯定是不可以的，这个时候就需要提供一些借口出来，直接访问。







如果我们不考虑继承的话，实际上我们只需要考虑 **public** 和 **private** 类型，公有的表示外界的函数可以访问，私有表示只有类内部的函数才能访问，也就是说只有类内部的代码才能改变类内部的状态，换句话说就是**外因必须通过内因**起作用，这就比较哲学了。对于 **protected** 类型来说，仅仅只是和 **private** 在**继承和派生**的时候不一样。它表示保护成员在派生类的成员函数是能访问的。关于这点，我们在讲到继承的时候再详细的说明。

## 数据抽象和封装

- 数据抽象是一种依赖于**接口和实现分离**的编程（和设计）技术，类设计者（也叫设计程序员）必须关心类是如何实现的，但**使用该类的程序员（服务程序员）**不必了解这些细节，使用者只要抽象的考虑该类型是**做什么的**，而不必具体的考虑**该类如何工作**。

假设我们实现一个栈类：

```
class Stack
{
public:
    void Push(int elen); //压栈操作
    Int Pop();    //出栈操作，它只需要关注这个类有什么功能，不需要
                //关注功能是如何实现的
}
```

//如何实现是交给**类设计者**去实现的，对于类的使用者来说，不需要关注类是如何实现的，也不需要关注类所采用的数据结构，只需要关注这个类能提供什么功能。只要这个类所提供的功能不变，也就是类所包含的接口不变，那么对于类使用者所编写的代码就是稳定的。

这就是数据抽象的编程思想。

- **封装**是一项将**低层次**的元素组合起来形成**新的、高层次的实体技术**，函数是封装的一种形式：函数所执行的细节行为被封装在函数这个更大的实体中，**被封装的元素隐藏了它们的实现细节**——可以调用函数，但是不能直接访问函数所执行的语句。同样的，**类是一个封装的实体**：它代表若干成员的聚集，设计良好的类隐藏了类实现的细节。

并非给客户越多的东西就是越好的服务，相反，仅仅给他需要关注的东西才是最好的服务。（哲学观点），这样可以防止粗心的程序员破坏类内部的数据结构。只能通过类所暴露出来的接口来访问，类内部的数据成员。这就是数据抽象与封装的意义。