III

第三篇　面向对象程序设计

本篇讨论类与对象，组合与继承，多态与虚函数，运算符重载，模板类和泛型程序设计等主题，以循序渐进的方式介绍封装 (encapsulation)，继承 (inheritance) 和多态 (polymorphism) 三种面向对象语言的主要技术。

在这个部分中，我们通过许多有意义的范例程序解释了向上类型转换 (upcast)、抽象化、派生类所定义的对象之构造和析构次序、混合组合和继承以建立新的类、重载虚函数以及虚析构函数等等被大部分介绍 C++的书籍所忽略的重要主题。在第 21 章中，我们通过复数运算 (特别是交流电路阻抗的计算) 的实例展示了“运算符重载”在简化程序上的强大功能。

第18章

类与对象

面向对象程序设计 (object-oriented programming, 简称 OOP) 是 C++ 最被重视的功能。在产生任何“对象”(object) 之前，我们需要定义一种包括了数据和函数的特殊数据类型—— 类（class）。本章介绍定义类以及如何使用类来定义对象的基本语法。这种使用类和对象来设计程序的方式称为“以对象为基础的程序设计”(object-based programming)

18.1 程序设计方法的演变

程序设计的目的是为了求得问题的解答或是对于数据进行特别的处理，并把结果以我们期待的方式呈现．我们把这些程序能够胜任的工作通称为“信息处理”。为了减轻程序设计的困难，一直不断有各种改进程序设计的方法论被提出来，形成程序语言演变的动力。

“函数” (function，或称为子程序，subroutine) 的诞生无疑是程序设计技术演变上的一个重要里程碑，它让程序变得更有组织，也提高了程序的再利用率。使用函数的中心概念是“算法”(algorithms)，每个函数都将待处理的输入数据和处理完的数据区分开来，函数的内部则由处理数据的步骤，亦即算法的具体内容来构成。如图 18.1.1 所示：

图 18.1.1 使用函数处理数据的基本概念

一个经由严格测试，能够正确处理数据的函数经常被保存下来，作为处理同类问题的基本建筑单元 (building blocks，积木的意思)．程序设计因此可以通过引用既有函数或自行建立新的函数，并加以重新组合来完成，如图 18.1.2 所示 (图中的方块代表函数，细箭头代表程序的调用和返回，粗箭头代表程序内依序进行的语句)。

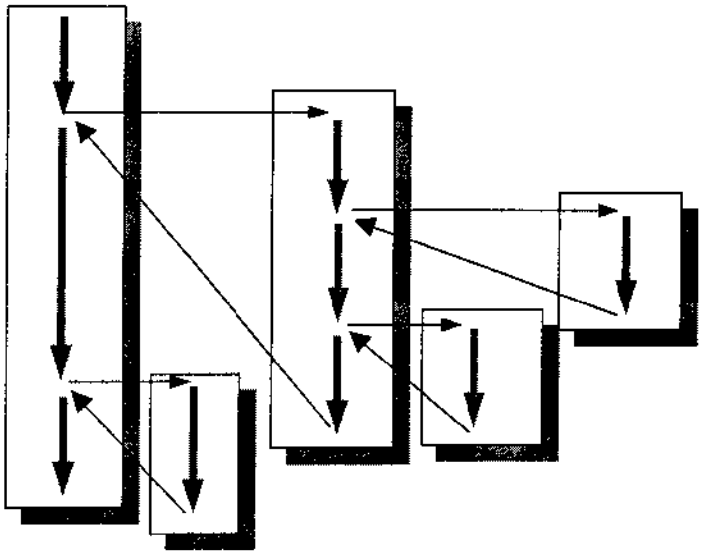

图 18.1.2　由函数构成的程序基本结构

另一方面，经由各种函数的累积，可以赋予程序处理更复杂问题的能力。同一类的函数更可以聚集成“模块”(module)，让程序的结构更有组织。各函数间通过严谨的输出输入数据管理，以及严格规范控制结构 (特别是不允许使用不受约束的 goto 指令)，以有系统的方式安排各程序模块；这种设计程序的方法统称为“结构化程序设计”(structured programming)。程序语言 Pascal 就是在这种想法下诞生的计算机语言，曾一度广受学校重视，被用为训练结构化程序设计的入门语言。

“函数”和“算法”的概念与计算机硬件的运作方式非常一致：预先规划好的处理步骤配合依序输入的数据，得到期望的输出数据。非常多的程序都以这种方式写成，而且可以顺利运作，这种依照处理步骤设计程序的方式称为“以过程为基础的程序设计”(procedural-based programming)。

以“面向对象”的概念作为程序设计的指导原则，则是由于实际的需要而被广泛重视的新趋势。虽然“面向对象”这个名词听起来有点神秘，事实上，我们每个人原本就是以“面向对象”的方式来认识和理解这个世界的。不管是自然界的虫、鱼、鸟、兽，人造的车、房屋、用具，或是虚拟的各种计算机游戏内的角色、武器，都可以被视为是具有各种属性

（attributes），能够有各种特殊行为 (behaviors) 的对象（objects）。面向对象的程序设计精神就是在程序中保留各种对象的属性，规划出使用者与对象，以及对象与对象间的交互关系，以完成程序数据处理的功能；而不是只重视处理问题的步骤。

面向对象程序设计 (object-oriented programming, 简称 OOP) 的原始动机有两个来源：一个来源是科学家在计算机内部仿真各种生物交互的行为、群落的消长以及物体间互相碰撞的现象 (例如空难或地面交通事故的仿真) 所得到的结论。另一个来源则是 Windows 操作系统的诞生。因为在 Windows 操作系统 之下，我们可以不断地开启一连串彼此非常相像，都有标题栏、工具列及滚动条等基本设施的窗口，面每个窗口内都可进行不同的工作，显示不同的界面。这些窗口程序的设计工作如果用传统的“结构式”或“程序式”的设计方法必须使用较复杂的技巧才能完成，如果以“对象”的观点来设计则非常契合。

18.2 抽象化和数据的隐藏

抽象化 (Abstraction)

面向对象程序设计（OOP）的第一个重要概念就是“数据的抽象化”。这句话的意思是：在思考问题和规划程序的时候，先不去考虑程序代码的细节，而集中注意在下述三个要点上：

(1) 在整个系统里面，有哪些参与的对象。

(2) 这些对象有什么属性。

(3) 这些对象如何和外界交互。

封装（Encapsulation）

形成对象的同时，我们同时也界定了对象与外界的内外区隔（这就是道家和佛家所指的烦恼根源）。至于对象的属性、行为等实现的细节则被封装在对象的内侧。外侧的使用者和其它的对象只能经由原先规划好的接口和对象交互。

通常我们可以用一个蛋的三重构造来比拟一个对象，如图 18.2.1 所示：

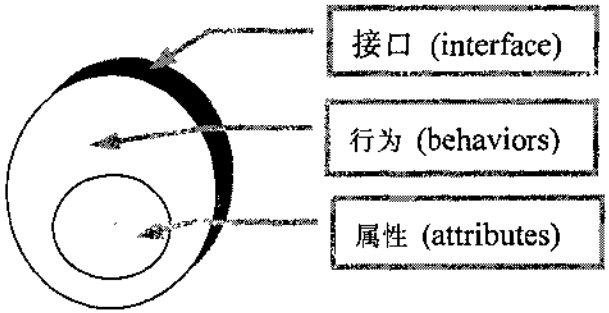

图 18.2.1　对象的三重构造

1. 属性（attributes）就好比蛋黄，隐藏在中心，不能直接碰触，它代表了对象的状态。
2. 接口（interface）好比蛋壳，可以和外界直接接触。
3. 行为（behaviors）好比蛋白部分，它可以经由接口与外界交互而改变内部的特征值，并把反应经由接口表现出来。

面向对象程序设计的好处

经由“面向对象”的思考方式，许多数据和函数都被重新安排，隶属于个别的“对象”之内。除了有机的归类之外，这些数据和函数的取用也分别受到限制，只有透过特定的接口才允许触发行为，进而改变对象内部代表状态的数据。

表面上看来，使用面向对象的方式开发程序带来了种种函数调用和变量存取的限制，然而这些限制使得大型程序的开发获得以下的好处：

1. 提高程序的再利用率

   大型程序并不是由全新的程序代码逐行构成，而是由既有的可用程序重新修订组合而成。如果程序以面向对象的方式写成，由于分隔清楚，功能内涵，最能够发挥提纲挈领的属性。而且由于严谨的接口限制，避免了数据和函数间不受控制的交互，有利于程序之再利用和修订。

2. 问题和程序间更直接的对应关系

   在早期由于软硬件的限制，设计程序时必须将待处理的数据和算法区分开来，同时通过函数将算法包装起来，以提高程序的结构性。而“面向对象”的设计

方式，让实际问题里（称为问题领域）的每个具体或抽象的对象，都可在程序里面找到相对应的对象，内部含有数据以及处理这些数据的方法。这种直接的对应关系让程序在规划阶段可以采取比较直观的方式进行；继之而来的程序设计和调试阶段，也可因为这种密切的对应关系而变得更有组织，更容易理解，也更易修正。

18.3 对象(object)与类(class)的关系

任何问题所涉及的对象，不管它是具体的还是抽象的，都具有以下两个属性：

1. 状态 (state)
2. 行为 (behavior)

在面向对象的结构下，问题领域的对象与程序内部的对象间有如表 18.3.1 的对应关系：

表 18.3.1 问题领域对象与程序内部的对象间的对应关系

问题领域的对象 (Problem space objects)	软件对象 (Software objects)
状态 (states)	数据成员 (data members)
行为 (behaviors)	成员函数 (member functions)

以一部电梯为例，它的状态包括目前所在的楼层位置，要前往的目标楼层，目前的速度，内部乘客人数，总重等等与运作有关的细节。而它的行为则有：接受按钮调用，接受目的楼层选择，以及驱动升降等功能，可以与外界交互或改变状态。

以面向对象的术语来说，与行为对应的成员函数又称为方法（methods）；它是所有能够改变数据成员的操作，具体的呈现对象的行为。

与一个对象交互时，必须传送消息（messages）。一个消息必须包含以下三个要素：

1. 接受此消息的对象名称。
2. 所要触动的方法。
3. 执行此方法所需要的参数。

我们会在本节稍后的范例中就消息作更具体的说明。除了必须声明目标对象的名称外，事实上传送消息相当于调用成员函数。

类 (class)

在我们所熟知的世界里，很少有哪种对象是唯一存在，找不到同类的。比如说，某人新买的汽车，就是世界上很多汽车其中的一部。使用面向对象的术语，我们说这部汽车是一种称之为“汽车”的类所产生的一个实例（instance）。

类（class）是某类对象的蓝图（或典范），定义了所有这类对象拥有的数据成员和成员函数。对于每个对象而言，它内部的数据成员与其它对象的数据成员是独立的，可以拥有不同的值，因此允许个别差异的存在。而同一类的对象则共享成员函数，因此具有共同的传送消息方式。

事实上，类就是一个广义的数据类型。我们已经熟悉了如何使用 int 和 char 等基本的数据类型来定义各种变量，也在第 15 章学过如何以 struct 和 enum 来定义合乎自己需要的数据类型。其中 **struct** 可以包括不同种类的基本数据类型，而类 (class) 更进一步把成员函数都包装在里面。

在表 18.3.2 里，我们以一个简单的 struct 和 class 的数据类型定义为例，比较它们和基本数据类型 int 在语法上的异同：

表 18.3.2　int、struct 和 class 数据类型定义的比较

数据类型	int	struct	class

定义数据类型的语法	（不需要）	struct Member
		{
		  int Age;
		  char Name[20];
		};
			class Account
			{
			  private:
			  static float Rate;
			  int Balance;
			  char Id[20];
			  public:
			  Account();
			  ~Account();
			  void Deposit(int);
			  void WithDraw(int);
			  int CheckBalance();
			};

定义实例的语法	int N;	Member Ma;	Account SomeOne;

public 和 private 成员

我们注意到，比起 struct， class 除了拥有成员函数外，它的定义内还有 **public** (公开的) 和 **private** (私用的) 两个关键词.

在上例中，“private:”之后的成员 (包括数据成员和成员函数) 只有同一个 class 内的成员函数 （亦即 Deposit()，WithRow()和 CheckBalance()三个成员函数） 才可以取用和调用。而在“public:”之后的成员则开放给程序内的所有语句。

依据 C++ 的标准语法，struct 除了内部成员的默认开放程度都是 public 以外，它和 class 没有任何差异。事实上，如果我们在声明成员时，都明确加上“public:”和“private:”的标志，则 **struct** 和 **class** 两个关键词是可以互换的。

接口 （Interface）

以表 18.3.2 的例子而言，存款余额（Balance）被设定为 private，因此必须透过被设定为 public 的函数成员才能存取. 这些函数成员，包括 Deposit()，WithDraw()和 CheckBalance()，称为这个 class 的接口。

成员函数的实现（Implementation of member functions）

在表 18.3.2 里，所有的函数成员都只是函数的声明 (prototypes)，真正的函数内容有三种实现方式：

1. 在 **class** 的本体之内实现 **inline** 成员函数
   如果成员函数的内容非常简短，我们可以将它的定义内容直接放在 class 的本体 (body，也就是类名称之后的大括号) 之中，这样做的效果会和 inline 函数一样，在编译时成员函数直接在程序内展开加入。因此在这个情况下“inline”这个关键词可有可无。例如：

   ```
   inline int CheckBalance() { return Balance;}
   ```

2. 在 **class** 的本体之外实现 **inline** 成员函数
   我们也可以在 class 的本体之外实现 inline 成员函数。这个时候，必须在传回数据类型之前加上关键词“inline”，然后在成员函数的名称之前加上它所隶属的类名称，以及范围确认运算子 (scope resolution operator) “::”，这个

语法和第 16 章介绍的名称空间内函数的定义使用资格修饰符 (qualifier) 是完全一致的。例如：

```
inline void Account::Deposit(int CashInput)
        { Balance += CashInput;}
```

3. 在 class 的本体之外实现成员函数

通常在 class 的本体之内成员函数都只有函数的声明（prototype），详细的函数内容放在 class 的本体之外。这个语法和 inline 成员函数的定义是完全一样的，只是少了关键词“inline”而已。例如：

```
void Account::WithDraw(int Cash)
  {
     if (Cash > Balance)
       {
          cerr <<  "" << endl;
          return;
       }
          Balance -= Cash;
          return;
  }
```

成员函数的调用

只有“public 成员函数”才可以被程序内的其它语句调用，“private 成员函数”只能被同一类内的成员函数调用。调用 public 成员函数是外界与对象交互的唯一方式，也是传送消息（message）给对象的方式。成员函数调用的语法是在函数名称前加上对象的名称和成员运算符号“.”；这个语法基本上和上一章 struct 成员的存取语法是一致的。如图 18.3.1 所示：

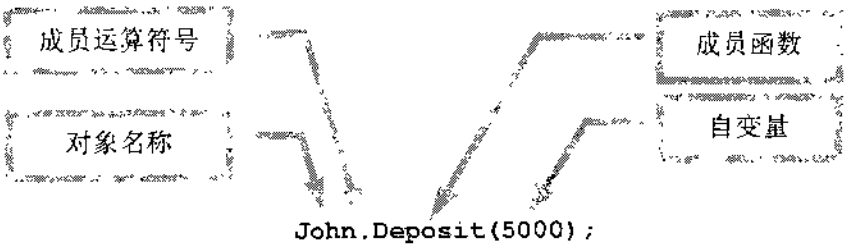

图 18.3.1 成员函数的调用语法

const 成员函数

有些成员函数，例如本例的 CheckBalance()，在执行时不应更动数据成员。为了避免不小心改变数据成员，可以将其声明为“const 成员函数”。其语法是将关键词 const 放在参数列的括号后面。例如：

```
int CheckBalance() const;
```

它的实现部分也要改成：

```
int Account::CheckBalance() const
  { return Balance;}
```

如果是 inline 成员函数，则写成以下的语句：

```
inline int Account::CheckBalance() const
        { return Balance;}
```

构造函数 (constructor) 与析构函数 (destructor)

在 **Account** 类的声明中，有两个成员函数非常特别，分别是：

```
Account();
~Account();
```

前者称为构造函数 (constructor)，后者称为析构函数 (destructor)。它们有两个特征是其它成员函数所没有的：

1. 成员函数的名称与类名称相同。
2. 没有传回数据类型（return data type），连 void 也没有。

如果我们没有特别为类声明构造函数和析构函数，则编译器会为每一个类制作一对没有自变量的默认函数。以本例而言，它们分别是

```
Account()  和  ~Account();
```

当我们进行对象的定义时，例如:

```
Account  SomeBody;
```

则默认的构造函数 Account() 就会被调用，只是由编译器提供的默认构造函数没有实质内容，因此没有产生任何影响。我们也可以自己建立没有参数的构造函数来对对象进行初始化的动作。例如:

```
Account::Account() {Balance = 0;}
```

这时候，编译器将不会再制造一个空的构造函数，由于没有自变量，它仍然称为“默认的构造函数”(default constructor)。

应用 C++ 允许函数重载（overload）的属性，我们可以同时另外建立一个可以接受参数的构造函数。例如:

```
Account::Account(int N) {Balance = N;}
```

以便在声明对象时一并给予初始值。

具有初始化功能的默认构造函数

具有初始化功能的默认构造函数 (default constructor) 有两种写法:

(1) 标准成员函数的语法。例如:

```
Account::Account() {Balance = 0;}
```

(2) 具有初始功能的缩写语法。

采用这种语法时，数据成员初始化的语句不写在函数本体的大括号内，而是写成:

构造函数名称(): 数据成员名称（初始值）{ }

例如：

```
Account::Account(): Balance(0) {}
```

如果有两个以上的数据成员需要初始化，则可以一并完成：

```
Account::Account(): A1(5), A2(1){}
```

(1) 和 (2) 这两种语法不一定要用在类本体外，也可以在类本体内以下列两种语法定义：

```
Account() {Balance = 0;}
Account(): Balance(0) {}
```

至于析构函数 (destructor) 的名称，是由一个波浪符号“~”加在类名称之前所构成，它不接受任何参数。只有由编译器所自动建立的析构函数才称为“默认的析构函数”(default constructor)。析构函数会在对象离开它的作用范围（scope）时自动被调用。例如，某个名叫 John 的对象在区块内定义，则在该区块结束时，析构函数就会自动被调用：

```
// ... 其它语句
{
  Account John;       // 在区块内定义名叫 John 的对象
  // John.Account()在此处自动被调用
  // ... 其它语句
}                  // John.~Account()在此处自动被调用
// ... 其它语句
```

析构函数的功能是担任回收存储资源的工作。我们在以下各章中要介绍向量和矩阵，它们会占用较大的存储空间，如果不在每一阶段都回收内存，数千次的重复运算很快就会将所有的存储资源耗尽。因此，析构函数的设计在那些场合就会变得非常重要。

对于数组而言，程序设计者必须在析构函数内使用 8.6 节所介绍的关键词 delete 来达到这个目的。我们会在第 22 章时继续讨论析构函数的相关问题。

提示

对于内建数据类型 (例如 int, float, char 等) 所定义的区域变量，则不需要借助析构函数，即可在离开作用范围 (scope) 时自动从堆栈 (stacks) 回收存储资源。

static 成员

在 Account 这个类中，我们注意到 Rate 之前有个修饰词 **static**。表示所有由 Account 所定义的任何对象，都共享这个数据成员。也就是说，如果定义了三个对象：

```
Account John, Cathy, William;
```

则 John, Cathy 和 William 都共享数据成员 Rate，而且如果 Rate 遭到修改，所有的对象都会因而受到影响。

除了共享数值的用途外，**static** 数据成员也常被拿来累计由类所定义的对象数目。

对象占有的内存大小

一个对象所占有的内存大小并不包括成员函数和 **static** 数据成员，而是由其余数据成员 (包括 public 和 private 数据成员) 的大小总和来决定。

18.4　以对象为基础的银行账户操作程序范例

在本节，我们将以一个完整的程序展示 18.3 节所述建立类和定义对象的各种语法。这个程序的功能是用来建立银行账户 (account)，并经由存款(deposit) 或提款 (withdraw) 来更改存款余额 (balance)。

首先，我们将银行账户视为类 Account，使用这个类可以建立各个独立的对象，亦即专属于某人的账户。类 Account 与外界的交互必须透过 Deposit(), WithDraw(), CheckBalance()和 CheckRate()来进行。这四个成员函数就是类 Account 的接口。在每一个由类 Account 所定义的实例 (instances) 的内部都有 Balance, Id, Rate 和 Count

四个特征值，其中 Rate 和 Count 被所有实例共享，因此声明为 static 变量。

我们将类 Account 的组成和功能具体地画成图 18.4.1。

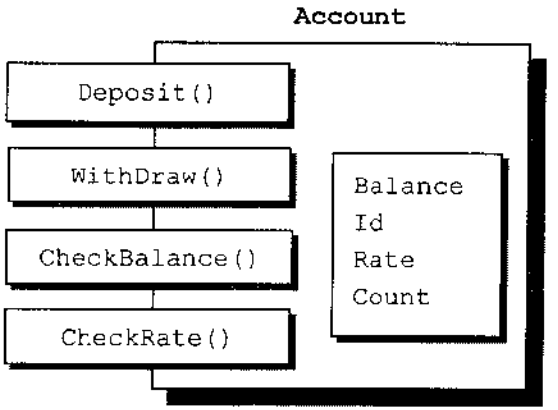

图 18.4.1　以对象的观点描述类 Account

虽然下面我们即将介绍的程序非常简短，我们还是将 class 的声明部分置于一个头文件中，而把实现部分放在另外一个文件中。这种做法在大型程序中尤其重要，因为 class 的声明部分是所谓的接口，告诉使用者如何与对象交互。class 的使用者并不需要知道实现部分的细节。

我们将程序区分为 AccountClass.h，AccountDef.cpp 和 AccountMain.cpp 三个文件：

(1) 头文件 AccountClass.h

头文件 AccountClass.h 声明类 Account，以及定义 Deposit() 和 CheckBalance()两个 inline 成员函数。成员函数 CheckRate()的定义很简短，因此直接放在类声明的本体内，因此也是 inline 成员函数。

此外，我们也在这个头文件内设定 static 变量 Rate 和 Count 的初始值。虽然这两个变量分别被设定为 public 和 private，它们的初始值设定语句是完全一样的。除了初始值设定外，由于变量 Rate 是 private，因此必须透过成员函数 CheckRate()才能取用它。

范例程序 文件 AccountClass.h

```
// AccountClass.h
#ifndef ACCOUNTCLASS_H
#define ACCOUNTCLASS_H

#include <iostream>
using std::cin;
using std::cout;
using std::endl;
using std::cerr;

//---- 声明类 Account ----------------------------
class Account
{
  private:
    static  float Rate;    // static 变量
    int           Balance;
    char        Id[20];

  public:
    Account();              // 默认构造函数
    Account(int);           // 构造函数(可设定初值)
    ~Account();             // 析构函数
    void Deposit (int);
    void WithDraw(int);
    int   CheckBalance() const;
    float CheckRate   () const
      {return Rate;}
    static int Count;       // static 变量
};
// -- 定义 inline 成员函数 Deposit() ----------------
inline void Account::Deposit(int CashInput)
  { Balance += CashInput;}

// -- 定义 inline 成员函数 CheckBalance() ------------
```

```
inline int Account::CheckBalance() const
{ return Balance;}


// -- 设定 static 变量初始值 -------------------------
float Account::Rate  = 5.8;
int   Account::Count = 0;

#endif
```

(2) 文件 AccountDef.cpp

文件 AccountDef.cpp 定义成员函数 WithDraw()，构造函数 Account()，以及析构函数 ~Account()。

为了能够知道构造函数和析构函数在何时作用，我们在构造函数和析构函数的实现内容加上了简短的输出消息。此外，构造函数和析构函数内分别对 static 变量 Count 做递增和递减运算，以掌握程序执行中对象的确实数量。

范例程序 文件 AccountDef.cpp

```
// AccountDef.cpp
#include "AccountClass.h"

// -- 定义成员函数 WithDraw() ------------------------
void Account::WithDraw(int Cash)
  {
    if (Cash > Balance)
      {
       cerr <<  "存款不足!" << endl;
       return;
      }
    Balance -= Cash;
    return;
  }
```