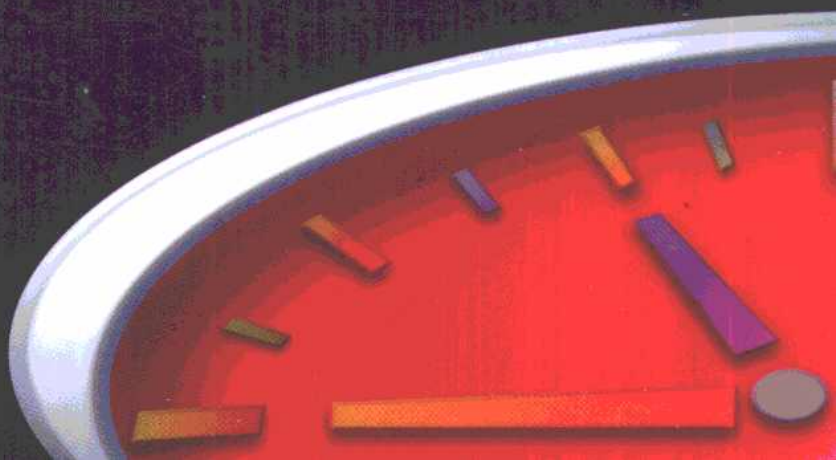
PTR
PH

计算机技术
译林
精选系列

C++面向对象高效编程

〔美〕Kayshav Dattatri 著
潇湘工作室 译



人民邮电出版社
www.pptph.com.cn

计算机技术译林精选系列

C++面向对象高效编程

[美] Kayshav Dattatri 著

潇湘工作室 译

人民邮电出版社

计算机技术译林精选系列

C++ 面向对象高效编程

◆ 著　　[美] Kayshav Dattatri
译　　潇湘工作室
责任编辑　俞　彬

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　　电子函件　315@ pptph.com.cn
网址　http://www.pptph.com.cn
北京汉魂图文设计有限公司制作
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本:787×1092　1/16
印张:48.25
字数:1 205千字　　2000年9月第1版
印数:1－4 000册　　2000年9月北京第1次印刷

著作权合同登记　图字:01－1999－3326号
ISBN 7-115-08592-7/TP·1679

定价:79.00元

前　言

现在，面向对象软件编程已经出现了，而且已经成为开发软件的首选方法，社会对优秀的面向对象软件开发人员、设计人员和系统设计人员的需求不断增加。为了在面向对象编程（OOP）领域中获得成功，人们必须忘却在多年面向过程的编程中所养成的一些习惯，并学习分析问题的一些新方法。

面向对象编程要求非常熟悉一些基本范例，也就是概念。理解这些范例是在OO 软件领域中打下牢固基础的基本要求。支持 OOP 的任何语言都必须支持这些基本范例。换句话说，学习高效 OOP 就是学习由许多语言（如 C++，Eiffel，SmallTalk，Java 等）所支持的强大的范例。本书的第一个目标就是让你理解面向对象编程的基本概念和原则，而不过多深入到语言的语法问题。这将在第一部分概念、实践和应用中介绍。

掌握支持 OOP 的语言的语法和学习 OOP 的概念并不一样。对基本 OOP 范例一无所知的人，可能会成为 C++或者 Java 的佼佼者。换句话说，任何理解了OOP 基本概念的人都可以在任何支持 OOP 的语言中有效地使用这些概念。而且，他也知道何时加入特定的概念。与此类似，任何掌握了链接列表概念的人都会发现，这是在 Pascal，C 或 Modula-2 中实现链接列表的基本知识。举个例子来说，如果你知道如何游泳，就可以在湖泊、池塘或者游泳池中游泳。语言仅仅是一个载体，它可以帮助你达到最后的目标。

学习 OOP 概念仅仅是第一个里程碑，而不是任何编程人员和设计人员的最终目标。他们必须可以将这些概念应用于他们的领域或者专门领域的问题之中。财政计划人员可能希望开发一个对象框架，以管理个人资金；一个商店可能希望开发应用程序，以管理它的存货。但是，在所有这些不同的领域中，应用 OOP 概念并不是一项简单的工作。解决玩具教科书的问题可能会非常容易，但是，要解决针对某个领域的问题和建立系统并不是件轻松的事情。学习来自专业领域的特定例子（例如文件系统、汽车代理管理、桌面出版、航班计划系统等）将会有所帮助。很明显，我在这些领域中都不是专家，而且大多数读者也不会发现它们非常有趣，因为他们并不熟悉这些领域。这是在编写面向对象设计的时候作者所面临的传统问题。然而，有一些经验方法，可以帮助任何软件专家解决复杂的问题，而无论问题所涉及的领域是什么。一个人必须明确知道何时使用什么工具。在本书的第二部分，我们用一些简单的例子说明了一些高级 OOP 技巧。在这部分中讲述了使用 OOP 进行高效设计的功能强大的策略。专业设计人员并不满足于仅仅学习诀窍和策略，他们还希望了解每种技术的利弊、替换方法和移植的效果及效率等诸如此类的问题。本书详细讨论了各种技术的所有必须和重要的主题。学习这些内容后，毫无经验的年轻人就会成为专家。

本书的重点是概念，而不是语言的语法，尽管我们讲述了许多语言问题。在所有主要的问题中，我们讨论了 C++和其他面向对象语言之间的差别。来自其他 OOP 语言的功能部件将和那些 C++中的进行对比，以让你理解 OOP 的概念（有时，可以帮助理解语言设计）。这些内容深入而广泛地讲述了 OOP 领域。例子是用 C++编写的，但这并不意味着 C++是唯一的选择。有许多其他同等优秀（如果不是更好）的 OOP 语言。对不同语言的设计目的进行深入了解是非常好的。考虑到那些已经熟悉 C++以外的其他 OOP 语言的读者，我们主要概念的细节将首先从 C++的角度开始讨论，然后对 Eiffel 和 SmallTalk 进行讨论。

由于 C++越来越流行（因为它是具有丰富的功能部件集合、严格的静态类型检查并支持编写工业等级的软件），理想情况下，一个人必须熟悉 C++和 OOP 的概念。然而，仅仅学习 C++语法并不是特别有趣，我们将瞄准一个更高的目标。为开拓 C++在 OOP 中使用的所有潜力，有很多特殊的模式、技术和技巧，它们让面向对象软件开发人员可以完全控制他们的程序。当前，在 C++世界中，还有许多时间测试技术。而且，有时候它有助于我们了解为什么某种功能部件在某种语言中是以这种方式出现的。它让程序员更好地理解语言（有时候是语言的设计者）。它也帮助我们更加有效地使用语言。在第一部分中的后面章节中，我们讲述了那些内容。在本书中，提供了许多强大的 C++策略和战术。

初学者应该从第一部分开始阅读，它讲述了范例、理论和应用程序。在第一部分中，章节是按照资料的相关性和难度进行组织的。第 1 章是初步的介绍，其后的第 2 章对数据抽象进行了彻底的讨论。在第 2 章，我们还讲述了统一建模语言和 Booch 方法。第 3 章和第 4 章对对象模型和良好的接口设计进一步进行讨论。在第 5 章和第 6 章中，我们讲述了继承及其特性，其后的第 7 章和第 8 章介绍了一些简单的问题。在第 9 章中说明了一般的编程问题。最后，第 10 章阐述了异常的处理方法。第二部分中的第 11 章力图说明建立强大的抽象概念的策略方法，而第 12 章则介绍了建立强大继承概念的策略方法。最后，第 13 章简要研究了 C++对象模型。

你可以以随机的方式阅读任何章节，但是，因为某些例子来自前面的四个章节，所以，这样读的话，无法保持较好的连续性。第二部分中的章节是独立的。

在全书内容中，我避免提供枯燥的实现代码，以保持重点突出和简洁。大多数实现代码是非常乏味的。实现代码仅仅是在它确实对所进行的讨论有所帮助的情况下，才会加入。

本书可以作为面向对象编程的介绍或者高级课本，而且，对于第一次接触 OOP 的编程人员和学生非常有帮助。本书的第二部分对于已经熟悉 OOP 概念的编程人员非常有用。熟悉其他 OOP（C++之外）语言的程序员，最好首先阅读第一部分，然后重点阅读第二部分。本书的第一部分可以作为研究生的 OOP 课程的介绍，而第一部分和第二部分一起可以用于关于 OOP 和 C++这方面的研究生课程。本书的大多数内容会极大地帮助初学者、中等程序员和设计人员，而老练的专家也会从第一部分和第二部分的许多章节中获益。在你已经阅读大部分章节之后，本书也可以作为很好的参考书。

本书假设你熟悉基本的 C++语法，而且具有普通的编程经验（课堂内容就已经足够了）。关于语法方面，我们将详细讨论 C++语言更复杂的一些部分（例如模板、异常和继承）。

在第 2 版中，我们更新了一些代码示例，以符合 ANSI C++标准，而且重新编写了一些内容，以更加清楚地说明概念。我们还对在第 1 版中发现的一些错误做了更正。

·作者·

第一部分

概念、实践和应用

第 1 章　什么是面向对象编程

最近，软件行业中的几乎每个人都被面向对象编程所吸引，甚至经理、主管和市场人员都喜欢对象技术。几乎没有什么东西比面向对象更好。看起来，面向对象软件似乎成为每个人孜孜以求的东西。人们可能会奇怪，面向对象到底是什么，它与我们已经使用了数十年的方法有何不同。软件开发人员可能会有被忽视的感觉，而且认为，因为面向对象的出现，他们的许多经验以及历尽艰辛获得的技术变得没有用处。在这种情况下，理解下列内容会有所帮助：

- 究竟什么是面向对象软件开发？
- 它的优点是什么？
- 它和进行软件开发的传统方法有什么不同？
- 它对传统软件开发技能有什么影响？
- 如何成为面向对象编程人员？

1.1 背　　景

程序员开发软件的历史已经有数十年了，他们使用各种不同的编程语言，例如 Algol，COBOL，Lisp，C，Pascal 等等，来实现各种规模的系统——从非常小的程序到大型系统。非常小的程序指小程序，例如汉诺塔的解决方法、纸牌游戏、简单的快速排序实现方法等等，我们编写这些程序作为课程指定的家庭作业或者仅仅用于学习。这些程序并没有任何商业价值。它们帮助我们学习新的概念和语言。相反，大型系统是指涉及到大型问题，例如库存控制、字处理、医院病人管理、天气预报、个人资金管理的软件系统。这样的系统需要由编程人员和设计人员组成的小组协同工作才能实现。并且，这些系统由公司出售，用以赚取利润。在设计和实现小型程序中学到的经验和教训对于解决大型问题会有帮助。在日常生活中，我们使用以各种语言实现的系统。我们也在使用这些语言和系统的经验中学到了许多知识。所以，为什么我们要转换到不同的编程方式？继续阅读下面的内容，在阅读完下面的内容之后，答案应该是显而易见的。

1.1.1　面向对象编程示例

在提出一个问题后（例如，问题的口头或书面说明），如何使用某种语言（例如 C）

设计和实现解决这个问题的方案呢？将它分解为多个便于处理的部分，这些部分便是模块。然后，我们设计许多数据结构，并在其中保存数据，我们还实现许多函数（也称为过程，或者例程）以对这些数据进行操作。函数修改数据结构，将它们保存到文件中并打印数据。我们对系统的所有了解都转换成了一组函数。我们工作的重点就是这些函数，因为没有它们，就无法完成任何有用的操作。这种编程方法称为面向过程编程，在这种方法中，函数是根本要点。之所以这样称呼它，是因为它的重点在于过程。它以函数的形式来思考问题，所以，它也称为问题的功能分解。

注意：

在C和C++中，术语过程、函数、子程序和例程之间没有差别。然而，在Pascal、Modula-2和Eiffel中，术语函数是指返回计算值的例程，术语过程是指接收某些参数和执行某个操作，但是不向调用者返回任何值的例程。在本书中，术语过程、函数和例程将互换使用，它们的含义是相同的。Algol、Fortran、Pascal和C等编程语言都称为过程语言。

然而，当我们更详细地研究这种实现时，我们发现，数据结构中信息对我们更加重要。让我们最感兴趣的是保存在那些数据结构中的值，它已经处于次要地位。过程只是一些修改数据结构的简单工具，在没有这些数据结构的情况下，过程不能做任何有用的事情。我们花费了大量时间来设计这些过程，甚至在关键不在这里的时候也把主要精力放在这些过程上。而且，在正在运行的程序中，这些过程中的代码永远不会改变。正是不同数据结构中的数据在程序的生存期内不断改变。从这种意义上说，过程是非常没有趣味的，因为它们是静态的。我们来举一个简单的例子，想像一个银行系统，在这里，客户可以有各种不同的银行帐号（例如存款帐号、支票帐号和贷款帐号），允许客户存款、提款和在帐号之间转帐。如果这个系统用C实现，我们可能看到这样的一组过程[1]。

```
typedef unsigned long AccountNum;
typedef int bool;

bool MakeDeposit(AccountNum whichAccount, float amount);
float WithDraw(AccountNum whichAccount, float howmuch);
bool Transfer(AccountNum form, AccountNum to,
              float howmuch);
```

AccontNum可以仅仅是一个正整数。我们可以用一个简单的数据结构来管理帐号：

```
// Just a plain bank account record

struct Account {
  char* name;       /* name of account holder */
  AccountNum   accountId;
  /* and many other details */
  float      balance;
```

[1]不要过多注意语法。需要注意的是过程和参数的概念。每种语言都有自己的语法，用以进行声明。

```
    float     interestYTD; /* year to data interest */
    char      accountType; /* Savings, Checking, Loan etc */
}
```

我们可以用下面的函数创建 Account 结构的一个实例，从而为客户创建一个帐号。

```
AccountNum CreateNewAccount(const char name [ ], char typeOfAccount);
```

1.1.2　银行帐号的表示

这个函数返回新帐号的编号。当我们研究这种解决方案时，很清楚，客户对了解帐号中的资金和从中得到的利息更加感兴趣，而不是允许他们具有存款和提款的功能。客户将帐号视为他们辛辛苦苦赚来的钱的安全天堂。事实上，他们并不注意如何进行提款或者如何存款；他们仅仅需要一种方便的方法，以进行这些普通的操作。但是，我们作为程序员，将注意力完全集中在编写函数 MakeDeposit（以及其他）中毫无趣味的代码上，并创建一个小的数据结构来管理数据。换句话说，我们将注意力放置在客户几乎不关心的问题上。而且，在客户和他们的银行帐号之间没有特殊的关系。客户仅仅被当作一系列的字符和数字，而帐号操作是在不关心谁拥有帐号和其中所包含的内容的情况下对数据进行操作的。银行帐号的功能仅仅是一个数字——帐号数字。

1.1.3　银行帐号的安全

进一步可以注意到，任何人（或者任何其他程序或者程序员）都可以创建一个帐号，并操作它。因为帐号作为一段数据而保存，任何可以访问银行帐号记录的人都可以修改它（甚至非法的）和提款。这是帐号被当作一系列字符和整数所造成的后果。并没有任何规则规定银行帐号必须仅仅由可以信任的银行职员修改，即使存在这样的规则，那么谁执行它呢？语言（例如 C 或者 Pascal）并不能做到这一点，因为他们并不知道银行帐号和普通整数之间的区别。

如果我们希望打印客户的帐号，那么我们将加入新函数：

```
PrintAccount(Account thisAccount);
```

它将进行打印。但是，函数必须知道正在打印何种类型的帐号（支票、存款等）。这很容易——只需查看 accountType 字段中的值。假设开始时我们有三种帐号（支票、存款和贷款），PrintAccount()函数理解这些类型，因为它将在这个函数的代码中进行硬编码。到目前为止，一切都顺利。现在让我们假设加入了一个新的帐号类型——retirement_account。如果碰巧传递 retirement_account 给 PrintAccount，会出现什么情况呢？它将无法正常工作。我们可能看见这样的错误消息：

```
"Unknown acount type- Cannot Pring"
```

或者更糟糕：

```
"Illegal account type - Call Supervisor:"
```

之所以会出现这种情况，是因为帐号的类型在系统中采用的是硬编码方式，因此不能修改，除非修改源代码，重新编译并再次链接。所以，如果我们加入了一个新帐号类型，我们需要修改与该信息有关的所有函数，并重新进行编译、链接和测试过程。所有这些都是枯燥的，而且易于出现错误。我们为什么会遇到这种问题？答案仍然是因为这样的一个事实，即函数和数据结构被当做是没有任何共同点的脱节的实体。因而，函数难以理解数据结构中的改动。我们愿意以这样的一种方式建立系统，即加入到系统中的每一个新帐号类型都不会影响任何其他帐号类型，而且不会引起对代码进行重新编译。换句话说，对现存的实现方案进行改进是非常困难的。

我们所讲述的所有这些问题都可以归结于，在最初解决方案中，重点的安排是错误的。我们将重点（错误）地放置在我们认为很重要的函数上，而彻底忽略了对客户（例如银行客户）而言更为重要的数据。换句话说，我们完全专注于如何完成它，即使看起来更应该将重点放在做什么上时也是如此。这就是面向对象编程和面向过程编程方法的区别。

1.1.4 用面向对象编程方法解决问题

如果用面向对象编程技术解决这个银行帐号问题，我们的注意力应该放在银行帐号上。我们首先注意客户希望对这个帐号进行什么操作，对他们而言，最重要的是什么等问题。简而言之，在面向对象编程方法中，重点是被操作的数据，而不是实现那些操作的过程。我们试图找到用户（在这个例子中是银行客户）可能对这个数据——银行帐号所进行的操作——然后，我们开始提供那些基本操作。而且，数据和操作并不像以前那样被当做彼此独立的实体。它们作为一个整体。数据带有一组操作，以便用户对它们执行一些有意义的操作。同时，数据本身并不会由任何外部程序或者过程进行直接访问。修改银行帐号内的数据的唯一方法就是使用为那个目的而提供的操作。由银行帐号提供的操作代表用户修改那个数据。现在，我们可以说，银行帐号是一个类，而我们可以创建任何数量的银行帐号实例，其中的每一个都是银行帐号类的对象。因而，面向对象编程是这样的一种编程方法，即程序由合作对象（就是类的实例）构成。而且，类本身可能利用继承关系而相关联[1]。

这里的关键是类和对象的概念。类是一个实体，它拥有一组对象的共同属性（特性）。对象是类的实例。类的所有对象具有它们所属类声明的相同结构和特性。你可以把类看做是切甜饼的工具，而甜饼就是切甜饼的刀具创建的实例。这个比喻可能有些粗糙，但是，它仍然和从类中创建对象的过程类似。切甜饼的工具决定了甜饼的大小和形状（而不是味道）。与此类似，类确定了所创建的对象的大小和特性。在问题的面向对象解决方案中，任何东西都是类的对象[2]。用这种方式看问题，在面向对象编程中，我们仅仅以类和对象以及类之间的关系和对象之间的关系这些方式来思考问题[3]。Shalaer-Mellor 用了一种非常好的方法来说明面向过程和 OOP 之间的区别——功能分解准确地按照顺序提出了系统设

[1] 实例将在下一章中详细讲述。

[2] 如果我们认为 main()程序类似一个 root 对象，所有的程序都从这里开始，这在 C++中几乎是可能的。

[3] 在一些其他不著名的语言中，对象可以创建其他类。

计中的三个基本元素（算法、流程控制和数据描述），而在 OOP 中顺序完全相反。

现在，我们重新回到银行帐号的问题上来，银行帐号是我们注意的重点，而且它成为一个类。这里是 C++中的 BankAccount 类的框架。需要再次说明的是，请不要过多担心语法。

```
class BankAccount {
    public:
    // Many details omitted for simplicity
    void MakeDeposit(float amount);
    float WithDraw();
    bool Transfer(BankAccount& to, float amount);
    float GetBalance() const;
private:
    // Implementation data for the private use of the BankAccount class
    float       balance;
    float       interestYTD;
    char*    owner;
    int      account_number;
};
```

对客户（例如，任何希望打开银行帐号和使用它的人）而言，最重要的就是这个类所允许的 Public：操作（在图 1-1 中以粗体表示）。private:段中的声明并不能由这些客户访问。它是只能在内部使用的操作。例如，MakdDeposit 操作的实现方式如下。需再次说明的是，请不要管 C++语法。

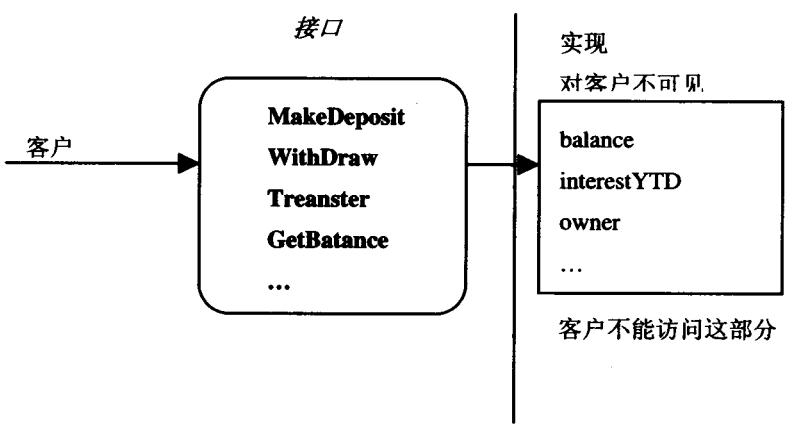


图 1-1

```
// Implementation of an operation of class BankAccount
void BankAccount::MakeDeposit(float amount)
{
    if (amount > 0.0
```

```
            balance = balance + amount;
    }
```

只有在类 BankAccount 内部声明的操作才能访问私有成员 balance（和其他私有成员），它们不能在类外部使用。我们将这种私有数据称为封装数据。这种在类中隐藏某些东西的方式称为数据封装。第 2 章将详细介绍数据封装。

1.2　理解对象模型

使用面向对象编程（OOP）的最大困难之一就是理解类和对象的概念。为了成功地使用 OOP，必须很好地理解这些概念。

OOP 中的基本实体是类。我们继续以银行帐号为例，每个 BankAccount 对象都有相同的结构和特性。因而，所有的 BankAccount 对象可以使用 MakeDeposit 以及类中声明的其他操作（例如，我们可以在类的任何对象上使用任何操作）。同时，可以确保所有的对象都有它们自己的私有数据副本（balance、acount_number 等）。对象之间的差别在于包含在私有数据中的值。例如，BankAccount 对象 alpha 的 balance 字段的值可能为 500，而另一个 BankAccount 对象 beta 的值可能为 10000，如图 1-2 所示。

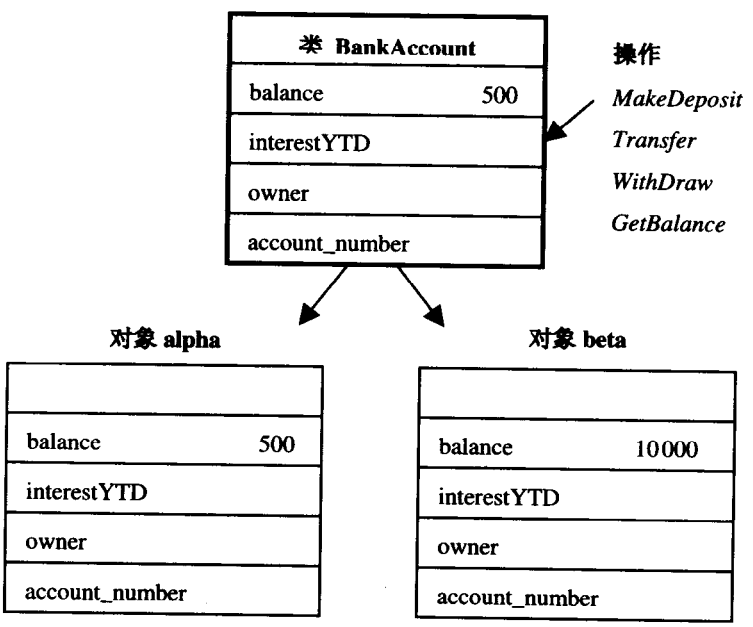


图 1-2

对象是类的实例（如甜饼）。类仅仅可以在程序的源代码中看见，而对象是类的活的实例，它参与到运行的程序中。对象占据内存空间。我们可以接触和感觉对象的存在。创建类的对象的过程也称为实例化对象。

在面向过程的编程方法中，我们总是讨论调用过程；我们总是以“调用过程 X（或者

只是 X）和调用 Y”来谈论问题（在这里，X 和 Y 都是过程）。但是，在 OOP 方法中，我们永远不会谈论“调用 X 或者调用 Y”；相反，我们说“对一些对象中调用 X”。如果我们有类 BankAccount 的对象 myAccount，我们说“在对象 myAccount 中调用过程 MakeDeposit。”换句话说，过程是在对象上调用的。没有对象，我们不能简单地调用过程。我们永远不会说“完成这个操作”；我们总是必须说“在那个对象上完成这个操作”。任何操作都应用针对对象，而且操作将对对象进行有用的操作（例如存款）。完全不存在修改其他人的数据的可能性。

在面向过程编程的模型下，当程序员查看数据结构时，很难感觉它将完成什么工作以及它将如何使用。而且，我们并不清楚它的含义。数据结构的目的、用法和限制都难以理解，因为数据结构并不是一个智能的实体。正确使用它们所需要的信息被掩埋在其他地方的函数集合之下。面向对象编程方法不会有这个缺点，因为任何人可以对类进行的操作都清楚地在类中以公共操作的形式进行了声明，并没有独立的数据结构存在。事实上，客户看不见独立的数据结构。客户所看见的所有东西就是类和一组清晰定义的操作。数据结构（隐藏在类内部）和对它的操作已成为一个整体。换句话说，客户对类可以进行的操作都在类本身中进行了详细的说明，客户并没有必要在别的地方查找这些信息。

1.3 术　　语

现在可以对类中的操作和数据使用正确的术语（参见表 1-1）。

表 1-1　　不同语言中使用的术语

定义	C++术语	Smalltalk 术语	Eiffel 术语
一组相似对象的说明	类	类	类
一组私有数据和函数	对象	对象或实例	对象或实例
高级、通用类	基类	超类	父类
低级、专用类	派生类	子类	子类
重新使用设计和代码的方法	继承	继承	继承
独立调用类型	多态	多态	多态
请求对象执行它的某个操作	成员函数调用	消息调用	操作调用
如何执行操作之一	成员函数实现	方法	例程（过程或者函数）
私有数据	数据成员	实例变量	字段、属性

C++　在 C++中，类内部的函数称为成员函数，而变量称为数据成员。函数和普通的函数类似，但它们属于一个类，因此是那个类的成员函数。同样，变量保存了属于某个对象的数据，因此它们是数据成员。

EIFFEL　Eiffel（和 Ada）将函数称为操作，而将变量称为属性。Eiffle 将类中的变量称为属性，而将对象中的变量称为字段。函数是一些操作，因为它们由客户使用，以对象进行操作。变量称为字段，因为对象与 Pacsal 中的记录类似。

SMALLTALK　在 Smalltalk 中，函数称为消息，而变量称为实例变量。

1.4 理解消息、方法和实例变量

类的任何用户（通常是另一个程序，甚至是另一个类）都称为类的客户。客户使用成员函数（消息）对类的对象进行有用的操作。我们稍后将会看见，客户可能仅仅创建类的对象，并使用那些对象，或者也可以创建一个新类，方法是从现存类继承。

SMALLTALK 在 Smalltalk 中，调用对象的接口函数（成员函数）被视作向对象发送消息。这看起来是合适的。我们向 BankAccount 对象发送 MakeDeposit 消息，要求它接收存款。向对象发送消息将导致执行那个对象中的方法（例如，当我们发送消息的时候，对象执行特定的方法或函数）。换句话说，对象对消息做出响应。消息仅仅是客户所看见的一个名称，而且名称绑定到（可能在运行时）接收这个消息的对象内的这个消息的正确实现（方法）上。类的每个实例（例如，每个对象）都包含实例变量的单独副本，如图 1-2 所示。

注意：

术语方法（和消息）、操作和成员函数在本书中将互换使用。它们的含义实质上相同。

1.4.1 对象中包含的内容

每一个创建的（或者实例化）的对象都得到自己的数据成员副本。数据成员（静态数据成员除外）都不是共享的。我们将在后面看见，仅仅静态数据成员（在 C++下）可以在运行程序中的类的所有对象之间共享。Smalltalk 也支持共享数据成员[1]。什么是成员函数？是否每一个对象也得到每个成员函数中的代码副本？很明显，不是这样。每个对象可以对类中声明的所有成员函数做出响应，但是，对象本身并不包含实现代码的副本。至少在一个正在运行的程序（进程或者任务）中，仅仅存在成员函数实现代码的一个副本。无论在进程中创建了多少个类的对象，成员函数的代码均不会复制。代码在类的所有对象之间共享。为便于理解，你可以想象类的实现代码驻留在一个库中。许多实现可能进一步优化这个性能，它们可能只为整个系统的所有实现代码保留一份副本。这通常利用动态共享库完成。所有这样的细节是操作系统特定的。例如，我们可以使用类 Card 来表示纸牌游戏中的一张牌，如下所示：

```
enum Suit { Clubs, Diamond, Heart, Spade, Unknown};
enum Rank { Two, Three, Four, Five, Six, Seven, Eight,
            Nine, Ten, Jack, Queen, King, Ace, Invalid };
enum Color { Red, Black};

class Card {
   public:
```

[1] 在讨论类的对象之间的共享时，我们将会看到有关的更多细节。

```
        void FaceUp();      // Show card face up
        void FaceDown();        // Make it face down
                // And many other member functions
        Card(Rank r, Suit s); // Create a card with these specifications
    private:
            Rank    cardRank
            Suit        cardSuit;
            Color   cardColor;
};
```

如果我们创建 52 张纸牌，那么，在纸牌中将有类 Card 的 52 个对象，每一个对象都有自己的数据成员 cardRank, cardSuit 和 cardColor 的副本。

```
Card myDeck[52];//Create a standard deck of 52 cards
```

可以单独操作纸牌中的每张牌。我们也可以按照下列方式实例化一些纸牌对象，整个过程如图 1-3 所示：

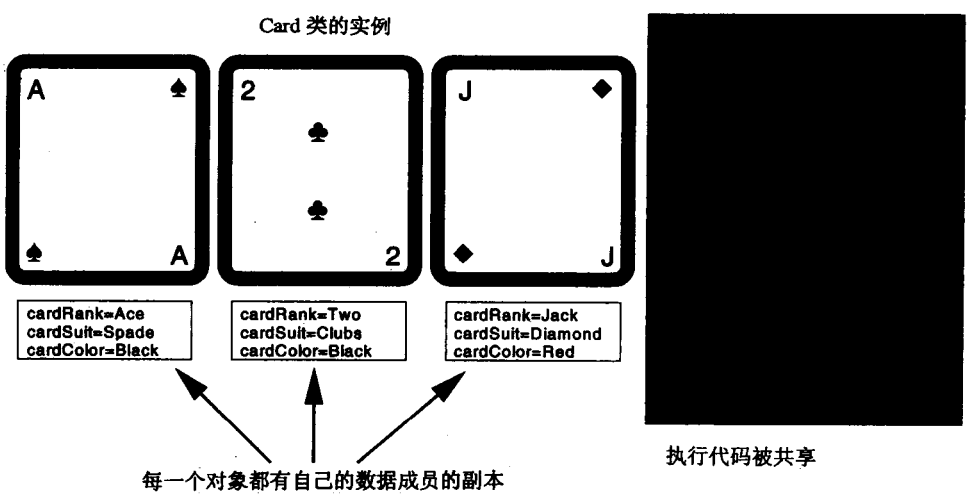


图 1-3

```
Card        spade_Ace(Ace, Spade); // Ace of Spades
Card        clubs_2(Two, Clubs)1 // Two of Clubs
Card        diamond_Jack(Jack, Diamond); // Jack of Diamonds
```

1.4.2　实例化（或创建）对象

一旦设计了一个类并实现了它，则希望使用那个类的对象的程序员就可以在代码中实例化它们。各种语言支持实例化对象的方式是不同的。在 C++中，对象的实例化看起来是一个简单的声明，如下所示：