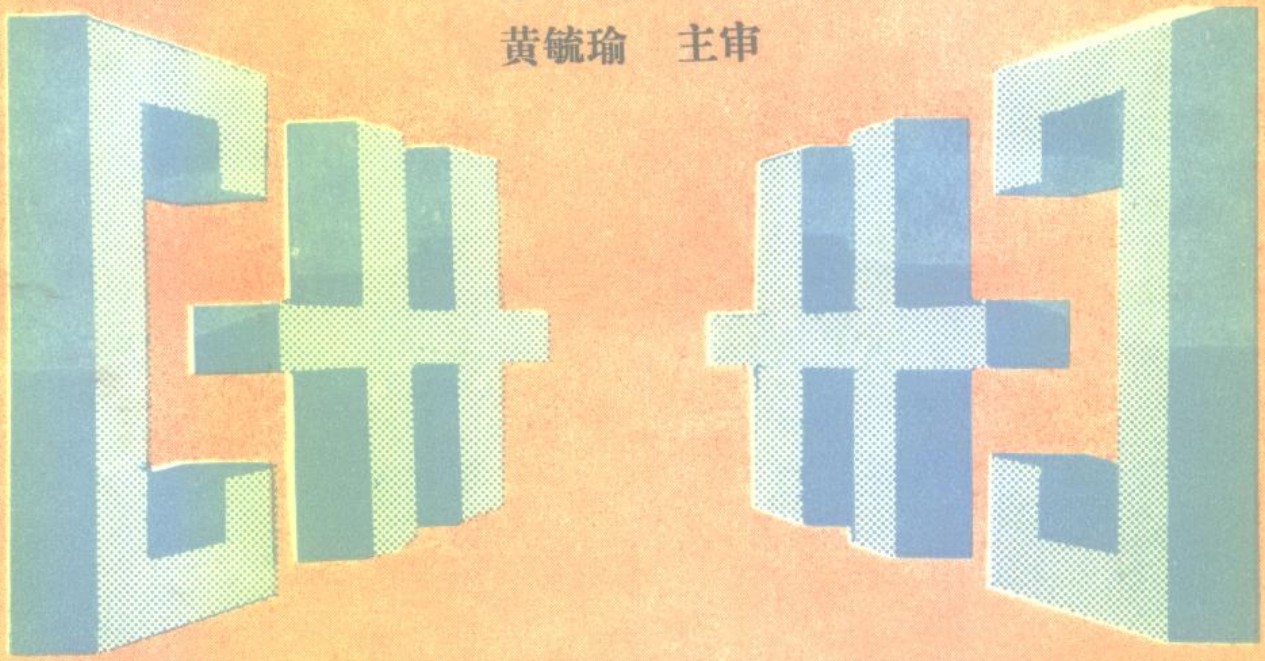
深入C++编程

李志刚
于 炜 编著
黄毓瑜 主审



北京航空航天大学出版社

深入C++编程

李志刚
于 炜 编著
黄毓瑜 主审

北京航空航天大学出版社

(京)新登字 166 号

内 容 简 介

本书讲述的内容介于程序设计方法和语言之间，着重介绍用 C++ 具体实现面向对象的思想。在介绍面向对象程序设计的基本概念和特点的基础上，详细讲解了面向对象程序设计的过程及方法，具体分析一个图形软件的设计过程，并给出较详细的程序框架。书中论述的所有内容都结合具体的 C++ 程序，深入浅出，使读者能了解到面向对象技术的本质，迅速掌握用 C++ 进行面向对象程序设计的思路和方法。

初学者可从书中学习到面向对象程序设计的思想和基本理论；对已有一定 C++ 编程经验的读者，会对 C++ 特性及实现有更深入的理解。

●书　　名：深入 C++ 编程
SHENRU C++ BIANCHENG
●编 著 者：李志刚　于炜
●责任编辑：王小青
●出 版 者：北京航空航天大学出版社(100083)
●印 刷 者：[illegible]
●发　　行：新华书店总店科技发行所
●经　　售：[illegible]
●开　　本：787×1092　1/16
●印　　张：9
●字　　数：227 千字
●印　　数：6000 册
●版　　次：1994 年 12 月第 1 版
●印　　次：1994 年 12 月第 1 次印刷
●书　　号：ISBN 7-81012-537-0/TP・143
●定　　价：8.80 元

前　言

随着计算机软件设计方法的不断提高和改进，面向对象的程序设计在我国也逐渐流行起来，C++ 语言作为 C 语言的超集，对面向对象的程序设计提供了强有力的支持，受到越来越多软件设计人员的青睐。但是笔者常常听到有人抱怨，他们面对一大堆 C++ 程序员手册、编程指南等参考资料，学习了 C++语言和面向对象程序设计思想，却不知如何下手编程，无法用类来实现现实的模型。甚至有些程序员仍然用结构化的程序设计思想来编写 C++程序，有些程序员设计的类及类构造明显违背了面向对象程序设计的原则，没有发挥出 C++ 潜在的威力。

基于上述考虑，本书从面向对象程序设计的思想出发，说明了 C++ 编译器怎样实现对面向对象程序设计的支持以及面向对象程序设计的基本概念和特性，通过分析一个图形软件的设计过程，重点介绍使用 C++ 具体实现面向对象程序设计的思路和方法。书中重点讲解了下列问题：软件的面向对象设计，软件的可重用性及维护性，数据抽象，继承性，多态性以及 C++的相关内容，如包容类、函数模板、重载等。

初学者可以从书中了解到面向对象程序设计的思想和基本理论，通过套用现成的编程方法，编写出合乎面向对象程序设计原则的程序，少走弯路。对于已有一定 C++ 编程经验的读者，本书会使他们对 C++ 特性及实现有更深的理解，在编程中更充分地发挥 C++ 的潜在威力。

由于作者水平有限，书中难免有错误和不当之处，欢迎广大读者批评指正。

本书在成书的过程中，得到黄毓瑜教授的支持并主审了该书，张勇智、李缙海、赵宇东、王小平同志为本书的编写提供了建议和帮助，在此表示衷心的感谢。

在本书的出版过程中，得到了北航出版社王小青同志耐心细致的指导和帮助，在此表示非常感谢。

编著者

1994 年 8 月

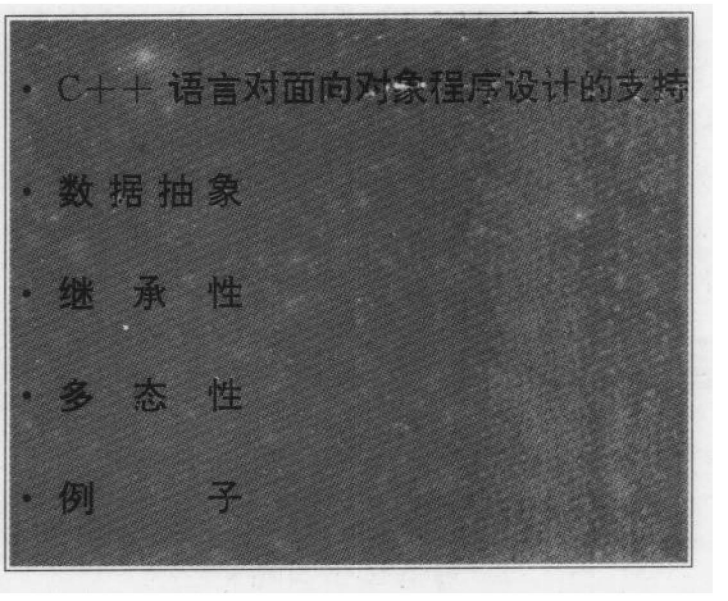
目　录

第一部分　面向对象程序设计基础

第二部分　面向对象程序设计方法

第一部分

面向对象程序设计基础



第一章　C++ 语言对面向对象程序设计的支持

C++ 是一种广为流行的面向对象(object oriented)的程序设计语言。在学习使用 C++ 编程之前，应先理解面向对象程序设计的基础理论和 C++ 对面向对象程序设计的支持特性。

本章从面向对象程序设计的角度简略回顾 C++ 较 C 语言的扩充部分，在后续的章节中，将分专题进行论述。掌握和理解本章的内容是读者使用 C++ 编写面向对象程序的起点。

1.1　面向对象程序设计与 C++ 语言

20 世纪 80 年代，结构化程序设计是最主要、最通用的程序设计方法。对于它，Wirth 曾经说过如下一段话："我们对付复杂问题的最重要的办法是抽象，所以，对一个复杂的问题，不应马上用计算机指令、数字或逻辑字来表示，而应该用较为自然的抽象语句来表示，从而得出抽象程序。抽象程序对抽象的数据进行某些特定的运算，并用某些合适的记号（可能是自然语言）来表示。对抽象程序作进一步的分解，并进入下一层的抽象，这样的细化过程一直进行下去，直到程序能被计算机接受为止。此时的程序可能是用某种高级语言（如 C 语言）或机器指令书写的。"在结构化程序设计中，是用函数来完成这种抽象过程的。自顶向下逐步求精的结构化程序设计方法是把一个问题分成若干子问题，依次进行下去，直到子问题很易把握为止。体现在程序设计语言中（如 C 语言），即为高层函数包含低层函数，而低层函数又包含更低层函数。换一种角度，可以说低层函数含有高层函数的特征，或说低层函数"继承"了高层函数的某些特征。结构化程序设计就是通过函数来体现设计的层次性。

随着工程规模的扩大，软件复杂性也增加了。为了缩短软件的研制时间，提高软件开发效率，一种新的编程方法应运而生，它就是面向对象的程序设计方法，简称 OOP。面向对象的程序设计吸取了结构化程序设计中的精华，是一种试图模仿人们建立现实世界模型的程序设计方法，它利用了人们对事物分类和抽象的自然倾向，引进了类的概念，具有数据抽象、继承性和多态性的特点。

与结构化程序设计不同的是，面向对象的程序设计是用类来抽象代表现实的实体，用类之间的继承关系来代表设计的抽象过程。函数只是对数据的操作，没有数据的概念，而类是数据和数据操作的集合，是非常贴近现实实体的表示形式，并且类之间的继承关系可以是设计的抽象过程的显式表示形式，这是面向对象的程序设计越来越流行的根本原因。

C++ 语言是在结构化程序设计语言—— C 语言的基础上发展起来的一种混合型编程语言。C++ 较 C 语言最显著的进步是提供了对面向对象程序设计的支持。它主要体现在以下几点：

- 引入类的概念，使用户可以对数据及对数据操作进行封装，实现数据隐藏。这是C++最重要的特征，是面向对象程序设计的基础。
- 对类之间继承性的支持，使建立类之间的层次关系成为可能 。此为设计面向对象程序的出发点。
- 引入虚函数，支持多态性，极大方便了面向对象程序设计。
- 引入类模板，在此基础上可建立参数类型及包容类。极大地简化了面向对象程序设计。

此外C++ 提供的函数模板及重载等功能，也为面向对象程序的开发提供了很大的方便。

下面对上述内容分别加以简略介绍。具体语言方面的细节可参阅有关的C++ 手册。

1.2 类

在C++ 中，类的基本形式为：

```
class  类名
       {
       存取描述符   数据集合 ;
       存取描述符   对数据操作集合( 函数集合 ) ;
       };
```

其中的存取描述符可以有以下三种形式 ：

- (私有类型) private
- (保护类型) protected
- (公有类型) public

它们分别代表了三种使用范围 ：

- 类本身
- 继承类
- 一般用户

从C++ 语言本身来说，用户可以用上述存取描述符修饰用户所建类中的任意数据和函数，但从面向对象程序设计的角度考虑，用户使用存取描述符时应遵循下列原则 ：

- 数据的存取权应尽量为私有的(private)。
- 函数集合中提供给外部调用的函数为公共的(public)。
- 只是提供给继承类调用的函数为保护型的(protected)。

- 其它函数为私有的(private)。

依据上述原则建立的类，可以达到数据隐藏的目的，从而为程序的维护及重用创造条件。下面以一个例子来进行说明：

```
class Object
    {
  private :
          int state ; // state >= 0
  public :
     int GetState( void ) const
        {
         return state ;
         }
          BOOL SetState( int curState )
                {
                 if( curState < 0 )
                   return FALSE ;
                else
                 {
                   state = curState ;
                return TRUE ;
                }
            }
     } ;
```

在上面的 Object 类中，由于数据 state 是私有的，从而限制了对它的访问，只能通过函数 GetState() 及函数 SetState() 来进行。直接对私有数据 state 进行操作将导致语法错误 ：

```
......
Object a ;
a . SetState( 6 ) ; // 正确 !
a . state = 6     ; // 语法错误 !
......
```

这样把数据 state 隐藏起来，使程序的其他部分“看不到它”。数据隐藏的优点，在语法上保证了数据不会被程序的其他部分意外修改，若想修改只能通过成员函数进行。当对象的数据处于不正确状态时，也可以知道只可能是调用的函数出了问题。如在上例中 state>= 0 总是成立的，但若数据 state 是公有的 ，那么下列语句虽在语法上是正确的，却会导致对象的数据处于不正确的状态。

```
.....
a . state = -3 ; // 使对象的数据处于不正确的状态
.....
```

从实例可以看出，通过使数据为私有类型，可以实现数据隐藏，从而把错误限制在局部。数据隐藏还有一个好处，它能使对象的外部接口和具体的执行过程分离，即访问对象的程序只需了解对象的公共函数的功能，而不必了解其具体的执行过程是什么。如上例所示：用户只需了解函数 GetState，SetState 的功能及用法，对于具体这两个函数如何执行，数据怎样存储不必了解。这就为程序的局部修改创造了条件。也就是说，若修改 GetState，SetState 两个函数的具体算法或者数据的存储形式，而保持两个函数的功能和使用方式不变，那么程序的其他部分将不需要做任何改动。

利用 C++ 提供的类的概念，可以方便地将它作为现实中实体的模型。用数据来代表实体的状态，用供外部调用的成员函数作为实体的外部表现 ，或称为与外部的接口，通过类把数据和对数据的操作（包括和外部的接口）封装在一起，作为一个整体，直接和现实中的模型相对应。

类是面向对象程序设计的“基石”，再依据 C++ 提供的继承性机制，读者就可以搭起整个应用程序的框架。

1.3 类的友元

friend 修饰符可以修饰一个类或一个函数。被修饰的类或函数，称为某类的友元。

在 C++ 中，具有 friend 修饰符的类和函数同所在类中的其它类和函数有相同的存取权限；另一方面，具有 friend 修饰符的类和函数是在外部定义的。这样，对数据隐藏是一种破坏。为提高程序的维护性，在使用 friend 修饰符时，应尽量使它修饰的类和函数对数据的存取通过该类的其它函数，不要直接操纵数据。如下例：

```
class student{
        private :
                char name[20];
                char class[20];
        public:
                char * GetName (void) const;
                void PutName( char * ) ;
                .....
        friend void Inspector( student ) ;
            } ;
```

在上面的程序中，函数 Inspector 的功能是对类 student 中的数据进行“视察”，并做相应的处理，形式如下：

```
void Inspector( student P )
{
......
(直接使用) P—> name ;
......
}
```

可以看到，在友元函数 Inspector 中，对类 student 的数据直接进行了操作。这样，当类 student 进行修改时，函数 Inspector 也要相应地进行修改。例如，把变量名 name 改为 Name ，那么 Inspector 中的函数也要进行相应修改。但若通过类 student 的成员函数 GetName 来执行，就不会发生这种情况：

```
void Inspector( student P )
{
......
(间接使用)P—>GetName();
......
}
```

在修改类 student 中的数据时，只要保持类 student 的接口不变，那么友元函数 Inspector 将无需做任何修改。这就为程序的维护提供了极大的便利。

1.4 继承性

C++ 语言中类的继承表示形式为 ：

```
class 子类名 ： 存取限定符 父类名,.....
        {
        存取权描述符 数据集合 ;
        存取权描述符 对数据操作集和( 函数集合 ) ;
        };
```

其中存取限定符有下列两种形式 ：

- private
- public

它们分别可实现如下功能 ：

- private 基类的 protected 及 public 数据及成员函数在派生类中成为 private 的。

（缺省形式）

- public 基类的 protected 数据及成员函数在派生类中仍为 protected 的，基类的 public 数据及成员函数在派生类中仍为 public 的。

从 C++ 语言角度来讲，在用户的应用程序中，通过使用 private 存取限定符，可以实现派生类对基类的性质约束，即对基类的性质加以限制或删除。使用 public 存取限定符，可以实现派生类对基类的性质扩展，即增加基类的性质。下面分别就这两种情况加以说明。

1.4.1 性质约束

考察下面的链表程序，其中类 Node 用来定义链表的节点，在链表类中 AddNode 函数和 DeTach 函数分别用于在当前节点（由 cur 所指向）处添加和删除节点，GetCur 函数和 SetCur 函数用于获取和设置当前指针 cur。除用 SetCur 函数进行设置外，一般情况下 cur 总是指向最后添加的节点。

```
typedef int NodeElement ;
class Node
    {
    private :
        NodeElement value ;
        Mode * next ;
    public :
        Node( NodeElement val ) ;
        Node * GetNext( void ) const          ; // 取得下一节点
        NodeElement GetValue( void ) const    ; // 返回当前节点的值
        void LinkNext( Node * )               ; //连接下一节点
        } ;

class List
    {
    private :
        Node * head ; // 链表头指针
        Node * cur  ; // 链表当前指针
    public
        List() ;
        BOOL IsEmpty( void )              ; // 若 TRUE，链表空
                                            // 若 FALSE，链表非空
        BOOL IsFull( void )               ;// 若 TRUE，系统无空间可供利用
                                            // 若 FALSE，系统还有空间可供利用
        void AddNode( Node * )            ; // 添加节点
        void AddNode( NodeElement )       ; // 申请空间，赋值，并添加节点
```

```
        Node * DeTach( int mark)          ; // 把节点移出链表 ,mark == 1
                                            // 释放所占空间,否则不释放
        Node * GetCur( void ) const       ; // 取得当前链表位置
        BOOL SetCur( int num)             ; // 设置当前链表位置
};
```

在上面的链表程序中,对节点的访问比较自由,用户可删除任意节点,也可在任意节点后插入新节点。若要在链表类的基础上建立一个堆栈类,由于堆栈只允许在其顶部进行 pop 和 push 操作,因此堆栈类必须对其基类——链表类进行限制,使其操作只能在链表的尾部进行,这可以用私有继承来实现。

```
class Stack : List
    {
    public :
            Stack() ;
            void Push( NodeElement node )
                {
                AddNode( node ) ;
                }
            NodeElement Pop( void ) ;
            List :: IsEmpty ;
            List :: IsFull ;
};
NodeElement Stack :: Pop( void )
{
NodeElement value ;
Node * node ;

node = DeTach( 0 ) ;
value = node -> GetValue() ;
free ( node ) ;
return value ;
}
```

注： List :: IsEmpty ; 语句使 List 类中的 IsEmpty 程序在 Stack 类中为外部可见。

在上面程序中,如果用户使用 public 继承:

```
class Stack : public List
        {
        public :
            Stack() ;
```

```
        void Push( Node * node )
            {
            AddNode( node ) ;
            }
    Node * Pop( void )
        {
        return DeTach( 0 ) ;
        }
// List :: IsEmpty ;
// List :: IsFull ;
} ;
```

那么下述使用 Stack 类的方法对前述两种继承方式都是正确的，程序都能正常工作：

```
Node node ;
Node * fpNode ;
  ......
Push( &node ) ;
fpNode = Pop() ;
  ......
```

但若在用户程序中使用了如下语句：

```
Node node ;
Node * fpNode ;
  ......
SetCur( 5 ) ;
Push( &node ) ;
SetCur( 6 ) ;
fpNode = Pop() ;
  ......
```

对 public 继承来讲，上述语句虽然在语法上都是正确的，但会导致整个程序一团糟。而对于 private 继承，SetCur(5) 及 SetCur(6) 在语法上是错误的，在编译阶段就不会通过，从而在语法上保证了程序的可靠性。

1.4.2 性质扩展

假定有一个描述学生基本状况的类：

```
class student
```

```
    {
    private :
            char name[20] ; // 学生名字
            char cls[20]   ; // 学生所在班级
  public :
          student() ;
          char * GetName( void ) const ;
          void                          PutName( char * ) ;
          char * GetCls ( void ) const   ;
          void * PutCls ( char * cls ) ;
} ;
```

现要在其基础上建立一个描述班长状况的类。对于班长，除具有一般学生的性质外，假定它还应有描述整个班的情况的数据，如学生总数，那么可以如下建类 ：

```
class monitor : public student
    {
    private :
          int studentNum ; // 班里学生总数
    public :
          monitor() ;
          int GetStudentNum( void ) const ;
          void PutStudentNum( int ) ;
} ;
```

通过继承类 student ，类 monitor 可以充分利用类 student 的数据和函数，对类 monitor 来说，其供用户调用的函数除在本类内定义的外，还包括从类 student 中继承来的公有函数。从用户使用的角度看，类 monitor 扩充了类 student 的功能和性质。

该节只是初步论述了类的继承性，更深层次的内容见第三章。

1.5 多态性

多态性对面向对象程序设计，主要是指程序在运行时刻，根据具体情况，选择相应的执行操作。多态性是一种迟后联编技术，在 C++ 中主要通过把成员函数声明为 virtual 来实现。

```
class ..... {
      .....
      virtual 函数 ; // 虚函数
      .....
      } ;
```