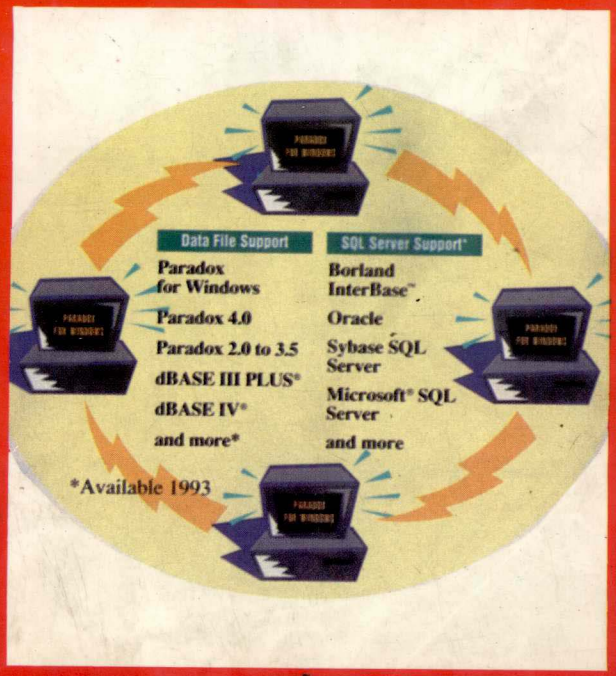
计算机语言技术系列丛书

Borland C++

BORLAND C++

面向对象程序设计进阶教程



柏雪枫编著

学苑出版社

计算机语言技术系列丛书

Borland C++面向对象程序设计进阶教程

柏雪枫 编著
亦 鸥 审校

学苑出版社
1994

(京)新登字 151 号

内 容 简 介

本书是学习和使用 Borland C++进行面向对象程序设计的高级读物，书中介绍了用 Borland C++进行面向对象程序设计的有关要点。

本书对从事软件设计、开发和应用的技术人员具有重要的参考价值。

需要本书的用户，可与北京 8721 信箱联系，邮码 100080，电话 2562329。

计算机语言技术系列丛书

Borland C++面向对象程序设计进阶教程

编　　者：柏雪枫
审　　校：亦　鸥
责任编辑：徐建军
出版发行：学苑出版社　　**邮政编码**：100032
社　　址：北京市西城区成方街 33 号
印　　刷：施园印刷厂
开　　本：787×1092　1/16
印　　张：24.375　　**字数**：565 千字
印　　数：1～5000 册
版　　次：1994 年 2 月北京第 1 版第 1 次
I S B N：7-5077-0776-8/TP・8
本册定价：27.00元

学苑版图书印、装错误可随时退换

目 录

此为试读,需要完整PDF请访问: www.ertongbook.com

第一章　C++类

C++最初被称为“具有类的C”。类(class)概念，是C++的中心内容，给语言以独特的效力。Borland C++支持所有最新的分类定义功能。本章对C++类的基本原则做了一个完整的介绍。

用户应明白，C++不是一种适用于初学者的编程语言。对于初学者即便是基本的C++也包含某些相当复杂的C概念。因此，从本章开始的内容是为那些至少对标准C和它的术语有一定经验的用户设计的。

要了解Borland C++，用户必须要了解C++类(class)。类，就是用户定义的类型。做为一种“类型”的“类”，这一思想较我们早先看到过的typedef更进了一步。记住，typedef只不过是由C提供的一个同义词功能。一个C++类说明允许定义一种新的对象——一个类对象——带有用户所确定的性质。C数据类型与C++类之间的区别是如此之大，以致于要分别用数据对象以及类对象这两个术语来区别这两个概念。

1.1　C和C++派生类型的比较

在标准的C中，派生类型(数组、结构和联合、函数类型和指针类型)都是由基本数据类型派生的。通过基本类型对象被一并放在一组里(如在一个结构里)或以一个新的方式使用一个基本对象类型(如为一个整数安排一个指针)，可以从一个简单类型派生一个复杂得多的类型。读者也可用typedef关键词把一个新的类型名与派生的对象联系起来。

基于两个原因，我们可认为结构是一个特别有用和有趣的派生数据类型。首先，该结构允许用户将具有不同基本类型的(甚至是其他结构)的数据对象集合在一起，并将结果做为一个实体来对待。标准的C结构是一个非常有用的类型，即使不向其中加入其他功能也是如此。第二点，数据结构是C++类的首要基础。读完本章后读者会明白这意味着什么。

然而，有些事情是不能用标准的C结构来处理的。不允许直接将一个结构做为一个表达式的一部分(虽然可写出一个函数将两个结构“加”起来)，并且不能在一个结构里定义某一函数(虽然用户可以在结构里为某一函数说明一个指针)，用户可定义一个C++类以便让该类中的对象去做以上两件事，甚至更多。

1.1.1　在C++里重新定义派生

用标准的C程序，用户可派生预先定义的数据类型的新分组和新用途，但不能改变它的性质——任何预先定义的类型的基本行为。因此，C程序的类型定义功能只不过是某些已存在的东西上应用新名字(同义词)的一种方法。

C++不仅允许用户定义一个类的类型的存在和内容，也要求用户定义类型的行为。因此一种类就可认为是一个数据类型，但一个类并不真是一个派生类型。一个类就是一个新的

类型，如将C++类说成是用户定义类型而不是派生类型可能更合适。

用户可使用标准C程序的功能来模仿C++类对象的功能，但C在能力和简洁上绝不能与类对象相比较。用户可写一个函数将两个整数数组加起来，但是，标准C程序不允许数组名称的以下用法：

```
int a[10];
int b[10];
int c[10];
...
c=a + b;    /* ILLEGAL IN STANDARD C */
```

用户可设计一个C++类，这样，它会支持对该类的对象使用加法运算符。以下的代码段简要说明了如何为数组类的对象定义一个加法运算。

```
class arrayplus { //skeleton class declaration
  int date[10];
public:
  arrayplus operator+(arrayplus&);
  ...
};
...
arrayplus arrayplus::operator+(arrayplus& op2)
{
  int i;
  arrayplus hold; // define a temporary object

for ( i=0; i<10; ++i )
    hold.date[i]=date[i] + op2.data[i];
  return hold;                    //return by value
}
...
arrayplus a,b,c; // Define some class objects
c=a + b; // this is legal,now
```

这个代码段含有本章和以后几章里将说明的一些新功能。用户应该可以在不详细了解C++语法的情况下，在这个代码段中发现三件事：

1. C++类定义了新(用户-定义)类型的内容和行为。在前面的代码段里，arrayplus类既定义了一个数组的数据元素的结构，也定义了将两个arrayplus对象加起来的方法。该方法(method)(从面向对象的编程系统中借用的一个技术名词)是以一个成员函数的形式给出的

(成员函数是一个实际上属于该类的函数)。本例中的成员函数是 operator+()。

2. 类的说明可以也确实有密切联系的函数用来控制该类的行为。标准的 C 程序结构可包含某一函数的指针,但没嵌入函数定义。C 结构也不能像类一样拥有一个函数。下节,用户将了解函数是如何被与类相联系起来的。

3. C++的注解与 C 注解极不相同。C++注解由两个连续斜线引入,仅被源行的终点所结束。因此,C++注解一定是源行里的最后一件事或是源行中唯一的一件事。

用户可以看到,用户可安置一个类的说明语句以便在一个表达式里直接使用含那个类(也就是拥有用户定义的类型)的对象。虽然控制一个类的对象的行为并不是用类定义所能做的一切,但却是用它所能做的最重要的事情之一。

1.1.2 C++封装

现在我们应该明白,类是被用来将一个对象的数据结构和控制该对象的数据的方法绑在一起。这就模仿了对从 OOPS 的普通理论中自然生成的对象的定义。将数据和方法绑在一起叫封装(encapsulation)。

封装是 C++程序设计中的一个重要部分。封装完成了三件事:(1)隐藏复杂性;(2)不鼓励程序员更改已经生效的代码;(3)促进对以前发展过的代码的重新使用。

C++用结构来封装类的对象(用 struct 或 class 两个关键字中的一个)。一般的 C 结构含属于该结构的数据成员,而且成员不能脱离结构而被访问。类的说明进了一步,它允许使用数据和成员函数,这两者都属于类。

这样的话,C++用 struct 类型做为类的对象的基础,但 C++结构和类还可做许多 C 结构所做不到的事。函数的所有权如数据给予 C++一定的增大"维数",如图 1.1 所示。

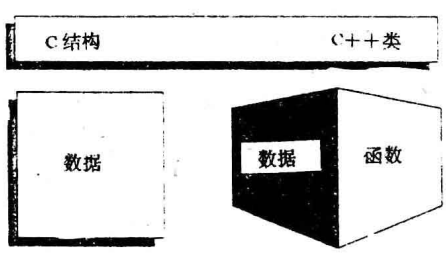


图 1.1 C 结构拥有数据成员,但 C++类既拥有数据成员也拥有成员函数

C++类用封装以两种方式来隐藏复杂性:隐藏内部数据结构的函数,提供一个用户(程序员)接口,而且它不要求类对象的内部工作的知识。这就像用户不必知道一个浮点数的位结构,或者是当浮点数被乘在一起时幕后所发生的事情,为使用类对象用户也不必关注于该类对象的内部结构或内部函数一样。用户只需将对象的公共成员函数调至对象的"接口"。封装的概念如图 1.2 所示。

封装不鼓励程序员对数据结构和已经是功能的函数进行不必要的干涉,对一个普通的程序员来说采用简捷的方法迟早会成为一种无法抗拒的欲望(包括我也包括你)。用简捷的方法更改数据的危险在于用户可能会忘记过程中的某些必要的部分,也可能搞错数据。

此为试读,需要完整PDF请访问: www.ertongbook.com

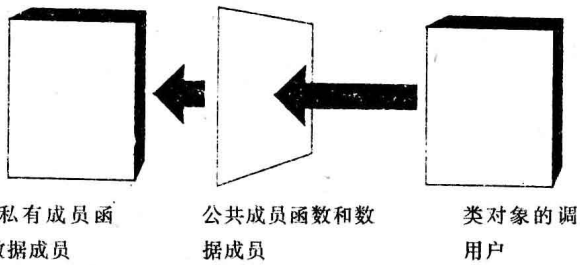


图 1.2　类对象的 C++封装为该对象的用户隐藏了目标的内部工作情况

当被允许修改的数据结构和功能函数因封装而处于视界之外时，用户干涉复杂结构的愿望就降低了。虽然刚开始封装看起来像是一个令人讨厌的限制，但经验证明这样做确实有好处。封装的确减小了数据结构意外损坏的事件。最后，调试大项目的必要性会大大地减小(但折衷的结果是调试会变得更困难)。

封装也可以以一种迂回的方式提供一种操纵数据结构的简捷方法。因为类对象是自含式，所以这种简捷法是可行的：用户不再需要经常地回忆起操纵类对象的复杂性了，对象自己就会注意去处理数据结构。

最后，被封装对象的自含式的性质鼓励应用以及再用已开发的代码，因为类对象倾向于可靠的和有能力的，因此 C++类是容易重用的，因为与普通的 C 程序比，在 C++类里写复杂的代码要容易的多(封装的另一个好处)，所以它们是可靠的。因此用户更倾向于写那些已预见到的类的所有用法的类说明。

1.1.3　用 struct 来说明类

在前面的内容里，用户已了解到一个类对象的公共成员函数提供了一个该对象用户的“接口”(interface)(这就是说，一个类对象的公共成员函数可被想使用这一对象的其他函数调用)。将公共成员函数想象成一个类对象的接口是一个很有用的想法，但 C++不像其他的面向对象的语言一样有正式的“实现性定义”或“接口模块”。

像 simula 这样的语言确有一个接口定义可将用户定义的对象放在主程序外的内存里。这样处理对象接口的优点是对类对象做修改时不会影响主程序。当类定义发生变化时不必重编译。

然而，在访问每一个类对象时一定要使用的形式接口模块有很大的缺陷：它速度慢。执行速度实际上是 OOPS 语言不能迅速普及的主要原因。

C++通过用和处理内部对象类型相同的方式来处理用户定义的对象以克服速度障碍。编译器通常知道一个类对象的尺寸和内部结构，而程序员却可能做不到这一点。但这样处理类就意味着在编译时必须要了解有关类的一切。对类代码的解释不是在运行时进行的。

因此，如果类(甚至是私有的，不可见的部分)发生了任何变化，使用这个类的任何程序都要被重编译或重新连接，否则这个程序将沿用类的旧版本，但这种模仿有一大优点，它允许主程序直接访问对象，它是与产生区标、快速运行期的代码的原始 C 的哲学有关系的。除

非真有必要的话用户不必对一个对象使用指针(或其他额外的接口)。

虽然C++并不在使用类对象的路径上产生人为的障碍,但它在决定不属于某一类的函数可以访问该类的哪一个数据成员和成员函数时确实遵守一定的规则。一个类对象可能有私有和公有两种成员:

- 私有数据成员和成员函数只适用于该类的成员函数和那些做为该类的友元的函数。当类用 class 关键字来说明时,类的成员缺省为私有的。用户可允许类成员为缺省以进行 private 访问或可将它们定义为私有成员来隐藏一个类对象的复杂性。用 class 关键字来说明类将在下节说明,友元函数将在本章结尾讨论。
- 公共数据成员和成员函数可被任何函数使用。当使用 struct 或 union 关键字来说明类时,类的成员约定为公共的。本节的后面部分讨论了用 struct 和 union 来说明一个类。

用 struct 或 union 关键字来说明一个类是能迅速而直接访问一个类和它的成员的最直接的方法。一个 C++ struct 的所有成员(数据和函数)允许从程序任一处的公共访问。缺省时,用 union 说明的类也允许公共访问;而且一个额外的特点是一次只有一个成员是有效的(与作为同一存储区的覆盖的联合的概念一致)。以下的讨论仅适用于结构,但同样的规则也可用于联合。

在C++中,一个结构是一个类,同时,也是一个真结构(避免了C中的许多兼容性问题)。比较C++结构与C结构对理解C++变化的额外功能是有帮助的。记住说明一个结构对象的一般语法:

$$\text{struct } tag_{opt}\{member\text{-}List_{opt}\}\ identifier\text{-}List_{opt};$$

用户应熟悉 struct 语法。可以像在普通的C里一样使用它,也可在C++用它定义一个真实类的类型。当用户用 struct 来说明一个类时,会出现一些新问题。程序 1.1 显示了其中的一些。在继续之前仔细查看一下代码。

程序 1.1　structob.cpp:用 struct 来说明一个类

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   struct myclass {
5      int a;
6      myclass();
7      void showclass();
8   }my_obj;
9
10  myclass::myclass()
11  {
12     a=37; // initialize the object
13  }
14
```

```
15  void myclass::showclass()
16  {
17    printf( "The value of my object is %d\n",a);
18  }
19
20  void main()
21  {
22    //---- Prove that everything in a class
23    //---- declared with struct is accessible
24
25    my_obj.a=64; //ref. just like a C struct
26    my_obj.showclass();
27  }
```

程序 1.1 中所列程序的某些部分对用户来说是没有问题的。第 1 和第 2 行里的 #include 语句是老朋友了，main()函数的一般设计也是如此。

用户对 Turbo C 早期版本的熟悉性可能会发现一些新东西，如在第 20 行。main()函数被说明为 void，许多有经验的程序员习惯于根本就不用返回类型说明 main()。然而，为 main()定义一个返回类型的失败，导致 Borland C++产生了一个编译时警告信息。用户如不介意警告信息，当然能不用返回类型对 main()进行编码。

结构成员的引用，在 25、26 行里也是大家所熟悉的。一个结构数据成员引用的语法与在普通的 C 中一样。但如仔细观察 26 行，可看到这里的引用是对一个成员函数——不能加入 C 结构中的东西。

在程序 1.1 中的结构说明还有几个方面是用户所不熟悉的。首先看一看 4—8 行。结构是在这些行里被说明，但要记住的是这是 C++，struct 说明实际上定义了一个类。

特别要注意的是第 6 行和第 7 行含有类的成员函数的函数原型。函数原型在 C 结构中是绝对禁止的。有几点关于成员函数说明的东西从样本代码看是不明了的：

- C++类可以说明特殊的函数——构造函数和析构函数—他们在程序中别的地方是不允许的。构造函数被用来设置和初始化类对象，而析构函数被用来破坏类对象(这就是说在需要的时候释放他们的内存)。这些函数将在本章“提供构造和析构函数”一节中讨论，在第五章中也将讨论。
- 要将类成员函数和类关联起来需要特殊的方法。关联一个成员函数和它的类的语法与用户在 C 中所学到的一切是不同的。设定成员函数的细节在本章后面“派生类的成员函数”一节中给出。
- C++中格式 function()的函数原型决不意味着允许函数定义去假设任何可能的函数参数这样一种老式的(Kemighan 和 Ritchie)实践。在老式的(K&R)C 中，一个空白的形式参数表意味着相应的函数定义可说明任何适于说明的参数。用户仅需要确认那个函数的函数调用确实传送了预期次数和类型的变元。

在 C++中，空白形式参数表与 ANSI 标准语法 fuction(void)意思相同。换句话说，变

元是禁止的。这一原则适用于任一在C++程序中说明的任何函数。

·C++认为老式的函数定义为一个年代错误。以下形式的函数定义过去是很普遍的：

```
sum(a,b)
int a;
int b;
{
    return a+b;
}
```

实际上，ANSI C 和 C++都允许这种形式的函数定义，这两类编译器均认为这种实践是危险的，而且数据溢出。此外，C++不允许用这种语法来定义类成员函数；用户必须对成员函数使用全函数原形。

在程序 1.1 中 10—18 行中含有定义类成员函数的代码。类的构造函数(那些特殊函数中的一种)，出现在 10—13 行。这个特殊的构造函数是个简单的例子：它初始化整数成员变量 a。类的成员函数的语法的独特之处在于不允许返回类型；而且构造函数的名称与类名称是完全一样的。这一成员函数最有趣的地方在于它访问成员变量 a 的方式。因为这个函数是类的一个成员，这一函数不要求使用结构成员操作符(.)，而在 main()中这个操作符是必须的。

其他成员函数 showclass()，出现于 15—18 行。它唯一的目的就是显示出成员变量 a 的值。与构造函数一样，showclass()最独特的特点是它不需要使用结构成员操作符来访问成员变量。

在程序 1.1 中要注意的最后一件事是第 8 行里的一个 struct 对象 my_obj 的直接定义，同时还有 struct 的说明。虽然这样一个定义是完全合法的，类对象几乎从不以这种方式在 C++中被定义(不管是否使用 class 或 struct 关键字)，正常用法如下：

```
struct myclass { // Declare the class here
  int a;
  myclass();
  void showclass();
};
  ...
void main()
{
  myclass my_obj; // Declare the class OBJECT here
  my_obj.a = 64;
  my_obj.showclass();
}
```

使用局部对象定义的主要原因(这就像其他成员变量一样,有 auto 位移量)是避免使用全局变量。因为C++类对象比标准的C数据对象大的多。全局定义的类对象在程序中占了很多的内存。目的是要让类对象尽可能而且尽快地消失。使用局部对象的其他原因包括公布性能和收集无用单元。类对象的行为和性能将在第八章中"控制对象的行为与性能"这一节中讨论。

在程序 1.1 中的其他部分也是值得讨论的。对所有类的公用的元素将在下一节讨论。从这一点看,类要用 class 关键字来说明。这并不意味着用 struct 说明的类是不需要的——只不过它们还不普遍。

1.2 说明C++类

用 class 关键字说明的类的成员(包括数据和函数)在缺省时是私有的;这就是说,他们只能被类的成员函数(和友元函数)访问。缺省时,类成员为私有属性这一事实给类的设计带来了意想不到的问题。这一结果需要更多的解释。

理解成员的公有或私有性质的基本原则是私有属性控制对成员对象的存储,而不是成员对象的可见性。为帮助用户理解存储控制和可见性(即传统的C作用域)之间的差别,这里举出两个C++程序。其中的一个运行,但不是按所期望的样;而另一个甚至不能正确地编译。

第一个程序,名为 access.cpp,有 cp1 和 cp2 两个类。类 cp2 中的成员函数试图访问类 cp1 中的成员变量 A。程序示于程序 1.2。

程序 1.2 access.cpp:访问控制不能影响这个程序,但可见性却影响

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int A;    // this is a global data object
5
6  class cp1 {
7     int A; //this is a member data object
8  public:
9     cp1() {A=37 ;}
10 };
11
12 class cp2 {
13 public:
14    void show() { printf("%d\n",A);}
15 };
16
```

```
17  void main()
18  {
19      cp1 X;
20      cp2 Y;
21
22      Y.show(); // this gets to global a
23  }
```

在程序 1.2 中，类 cp1 与 cp2 是不相关的；它们是独立的类。因此，当 cp2 的成员函数 show()(写作 cp2::show())试图访问一个名为 A 的变量(14 行)，它是全局变量 A(第 4 行)，而且对该函数而言既可见也可被访问，而不是在 cp1 中的预期变量。

可存储性甚至对于程序 1.2 中的 cp1 成员变量 A 也是不可能的。这个变量对函数 cp2::show()是完全不可见的。用户可能会通过重写 14 行和使用作用域分辨运算符，来强制实现对变量 cp1::A 的访问：

```
14    void show() {printf ("%d\n",cp1::A);}
```

然而，用户如果真的这样做的话，编译器便错误地假设用户指的是一个位字段对象并产生一个错误信息(作用域分辨将在后面的几处内容中讨论，包括第三章中的“作用域分辨运算符的应用”这一节)。总的来说，对成员变量 A 的存取是由以下几个因素控制的：

- 一个 C++结构和类有它自己的作用域。因此，当编译器在 cp2 的一个成员函数里找到一个变量标识符 A 的引用时，编译器首先为标识符搜寻 cp2 的作用域，在那未发现它。
- C 的常规作用域原则要求为所提到的标识符搜索参数的外层作用域。然而，外层作用域不包括 cp1 的成员，因为那个类有它自己的作用域(这就是为什么 cp1::A 对 cp2 完全不可见)。结果是找到并使用了 A 的全局复本。

如果改写程序 1.2，使得类 cp2 是由类 cp1 派生的，访问控制便成了支配因素。派生类(driving class)(类继承)在第七章中讨论。从现在起，要着重了解的有关派生类的事是派生类继承(携带)了其父类(派生类的类)的许多特征。如正确地说明，派生类也可访问父类的公共成员。改写的程序见程序 1.3。

程序 1.3　access2.cpp：因为 C++structs(包括类)有它们自己的作用域，因此存储控制在这里是一个重要问题。

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int A; // this is a global data object
5
6   class cp1 {
7       int A; // this is a member data object
```


此为试读，需要完整PDF请访问：www.ertongbook.com