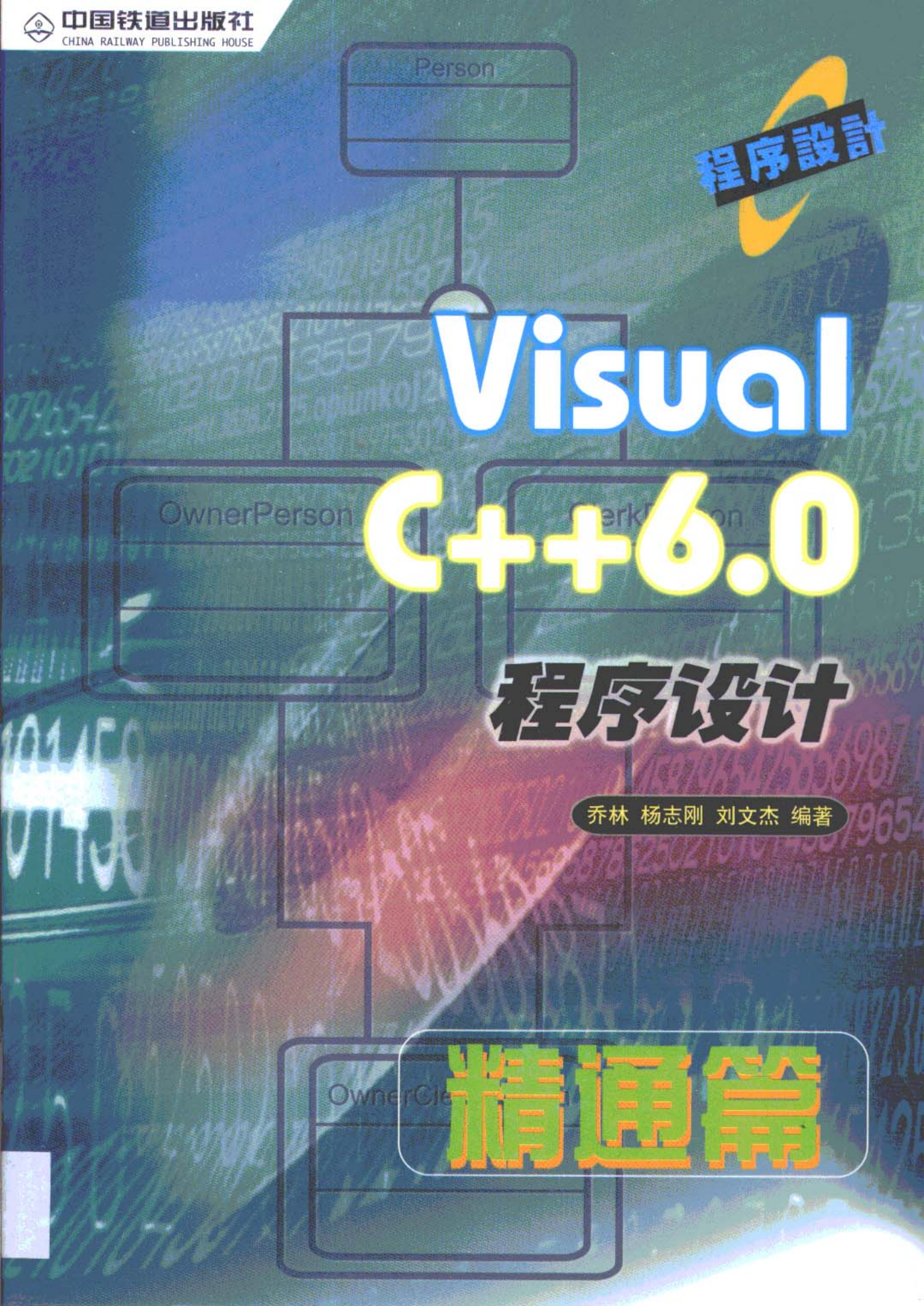
中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE



程序設計

Visual C++6.0 程序设计

乔林 杨志刚 刘文杰 编著

精通篇

Visual C++ 6.0程序设计

精通篇

乔 林 杨志刚 刘文杰 编著

中国铁道出版社

1999年·北京

(京)新登字 063 号

内 容 简 介

本书用多个程序实例介绍了 Visual C++程序的编制方法。内容包括：类型转换、OOP 程序设计、继承与重载、多态性与动态联编、列表与视图、格式化文本、属性编辑、打印、多任务与多线程等内容。

本书适合计算机软件开发人员和一般计算机爱好者使用。

图书在版编目（CIP）数据

Visual C++ 6.0 程序设计：精通篇/乔林编著.-北京：中国铁道出版社，1999.10
ISBN 7-113-03545-0

Ⅰ.V… Ⅱ. 乔… Ⅲ.C 语言-程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（1999）第 64607 号

书　　名：Visual C++6.0 程序设计——精通篇
作　　者：乔林　杨志刚　刘文杰　编著
出版发行：中国铁道出版社（100054，北京市宣武区右安门西街 8 号）
策划编辑：严晓舟
责任编辑：刘　波
特邀编辑：曹宝惠
封面设计：新创工作室　冯龙彬
印　　刷：北京市兴顺印刷厂
开　　本：787×1092　1/16　印张：29.75　字数：719 千
版　　本：1999 年 11 月第 1 版　1999 年 11 月第 1 次印刷
印　　数：0001—5000 册
书　　号：ISBN 7-113-03545-0/TP·407
定　　价：47.00 元

第 1 章

强制类型转换与模板

本章讨论 C++ 新增的部分特征。C++ 是 C 的超集，它的最成功之处是为支持面向对象的程序设计而提供的语言成份，这使 C++ 程序员可以进行面向对象的程序设计。除函数名重载和函数模板之外，这些新增的特征并不是为了支持面向对象的程序设计，而是为了消除 C 中的不安全因素。

1.1 强制类型转换

在 C 中，将一种类型的值转换为另一种类型 T 的值的强制类型转换使用表达式：

```
(T) 运算表达式   //  例如 (int)x 和 (unsigned int)x
```

在 C++ 中，强制类型转换可以使用下面的表达形式：

```
T(运算表达式)   //  例如 int(x) 和 (unsigned int)(x)
```

这样的强制类型转换更类似于函数调用。因为强制类型转换本身隐含着从一种类型的值向另一种不同类型值的映射，这与函数的作用是一样的，所以上述 C++ 表达形式要更为合理一些。

1.1.1 关键字 static_cast

C++ 操作符 static_cast 提供了一种新的进行强制类型转换的表达形式，使用该操作符进行强制类型转换的语法为：

```
static_cast<T>(运算表达式)
```

它将“运算表达式”的值转换为 T 类型的值，这种类型转换只适用于表达式的类型与类型 T 之间有适应关系的情形。static_cast 的类型转换能力虽然较弱，但是更安全。此时，编译器将会指出不恰当的类型转换。下面是使用该操作符进行类型转换的例子：

```
void *pch;
static_cast<char *>(pch);
```

在 C++ 中，T * 与 void * 相互之间是适应的，基本数据类型中的整型与字符类型、整型与实数类型之间是适应的。我们可以使用该操作符进行此类强制类型转换，有关继承情况下的类型适应关系将在第 3 章中讨论。

1.1.2 关键字 reinterpret_cast

C++ 的操作符 reinterpret_cast 用于将一种类型的值解释为另一种类型的值，在它做这种转换时，不改变值的位序列，这是与 static_cast 不同的地方。在必要的时候，static_cast 要将值的位序列变换成目标类型所要求的形式，例如将一个 int 类型的值转换为 float 类型。

reinterpret_cast 操作符的使用语法为：

```
reinterpret_cast<T>(运算表达式)
```

这里，T 可以是指针类型、引用、算术类型或指向函数的指针类型。使用该操作符，指针可以被解释为整数类型的值，整数类型的值可以被解释为指针。在将指针转换为整数，再将结果转换为指针时，转换结果与原指针相同。该操作符用于替代 C 中与实现相关的或不安全的强制转换。例如：

```
char *pch;
struct TAGA
{
  char pch[10];
};
TAGA *ptaga;
pch = reinterpret_cast<char *pch>(ptaga);
```

在 C++ 中，reinterpret_cast 是最强有力的类型转换操作符。如果使用不当，会给程序造成严重的运行时错误。

1.1.3 关键字 const_cast

此外，C++ 还提供了操作符 const_cast，此操作符用于从某个类型中删除或添加 const 或 volatile 修饰符。它的使用语法为：

```
const_cast<T>(运算表达式)
```

设“运算表达式”的类型为 V，则 T 必须是由 V 经 const/volatile 修饰符修饰之后得到的派生类型，或反之。该强制类型转换表达式的结果类型是 T。

1.1.4　关键字 mutable

mutable 为一类存储类关键字，它用于处理对象 const 状态的一种特殊情况。Mutable 只能用于处理类的非 const 和非 static 声明的数据成员。任何声明为 mutable 的数据成员都可以使用 const 类型的成员函数进行更新。例如，下述代码在 Visual C++ 6.0 中运行完全正常：

```
class X
{
  public:
    bool GetFlag() const
    {
      m_accessCount++;
      return m_flag;
    }
  private:
    bool m_flag;
    mutable int m_accessCount;
};
```

数据成员 m_accessCount 声明为 mutable 类型的整数，因此可以通过 const 类型的成员函数 GetFlag 进行更新。

此外，Visual C++ 6.0 还提供了另外一个强制类型转换操作符 dynamic_cast，有关这个强制类型转换操作符的详细内容，请读者参阅第 4 章的讨论。

1.2　创建堆对象

在程序中通过声明语句来建立对象，完全限制了创建或删除对象的时机，以及在特定时刻程序中所存在的对象的个数，这使得程序不适应许多稍稍复杂的应用场合。C++ 的 new 操作符能够使程序在运行时根据程序的需要随时创建对象。new 操作符的使用形式为：

```
new T( 初始值列表 )
```

new 操作符的任务是在堆中建立一个 T 类型的对象，上述表达式的值是指向所创建的对象的指针值，类型为“T *”。括号中给出对象创建时的初始值，如果省去括号和括号中的初始值，则对象被创建时的初始值是任意的。为能够在程序中存取 new 所创建的对象，必须在 T * 类型的一个指针对象中存储指向所创建的对象的指针值。例如，下面的语句创建一个初值为 0x76 的 char 类型的对象，并将指向该对象的指针值存储于指针对象 pch 中：

```
char *pch = new char( 0x76 )
```

堆对象在不使用时使用操作符 delete 删除。例如，下面的语句删除 pch 所指向的对象：

```
delete pch;
```

操作符 new[] 用于创建数组类型的对象，其使用语法为：

```
new T( E )
```

其中，T 是一个类型表达式。E 是一个运算表达式，类型为 size_t（与 unsigned 同义）。上述表达式的求值结果在堆中创建一个数组对象，数组对象的元素对象的个数等于表达式 E 的值，每个元素对象的类型为 T。

上述表达式的值是指向所创建的数组对象的第一个元素对象的指针值，类型为“T *”。该指针值应被存储在某个“T *”类型的指针对象中，以便程序能够存取 new 所创建的数组对象的每一个元素对象。

当使用 new[] 创建数组对象时，不能为数组对象指定初始值，其初始值是任意值。使用 new[] 创建的数组对象使用操作符 delete[] 删除。例如：

```
delete[] pch;
```

delete[] 操作符的使用要求与 delete 操作符一样。

1.3 函数模板

本节讨论 C++ 的高级特性——函数重载与模板。

1.3.1 函数重载

函数重载指的是一个标识符可以被用作多个函数抽象的名字。当一个标识符被用作多个函数的函数名时，必须能够通过函数的参数个数或参数类型使这些函数区别开来，这才是合法的重载。例如：

```
void swap( int& a, int& b );
{
  int n = a;
  a = b;
  b = n;
}

void swap( float& a, float& b );
```

```
{
  float n = a;
  a = b;
  b = n;
}
```

在上面声明的两个函数中，标识符 swap 被用作两个函数的名字。这两个函数的参数个数相同，但参数类型不同，所以，这是合法的重载。当调用重载函数时，编译器通过参数个数或类型来确定调用哪个函数。

在重载函数的定义和使用中，需要注意下面几个问题。

首先，要确保不同的参数类型确实不同。例如下述代码是错误的：

```
typedef float real;
float swap( float& a, float& b );
float swap( real& a, real& b );
```

因为 real 是 float 类型的另一个名字，在调用函数 swap 时，编译器不能确定应调用哪个函数，所以上面的重载函数声明是错误的。

其次，函数的返回类型不能用于区分重载函数。例如：

```
int f( int a );
void f( int a );
```

如果存在函数调用表达式 f(1)，则在没有确定应调用哪个函数时，该表达式的类型也是错误的。

T const * 与 T *，以及 T & 与 T const & 是不同的类型，所以如果使用它们进行函数重载，是可以区别的。例如，

```
void Print( const char * );
void Print( char * );
```

都是合法的重载声明。但 T const 与 T，以及 T & 与 T 不能用于区分重载函数。

部分时候使用 long 或 double 区分重载函数会产生错误。例如，设有下述重载函数：

```
int f( long a );
void f( double a );
```

此时，如果有函数调用语句：

```
f( 10 );
```

编译器不知道如何对 10 进行类型转换，因此将给出错误信息。正确的表达形式是：

```
f( 10L );
```

或

```
f( 10.0 );
```

读者要注意的是，在 Visual C++ 6.0 中，一个浮点常量的类型是 double，一个 float 类型的常量应表示为 10.0F；而一个整常量的类型按其值的大小可能是 int、unsigned 或 long 类型，这取决于按所给出的次序，哪种类型的值域能表示该整常量。

当重载函数声明了缺省参数值时，一定要注意这些重载函数的可区分性。例如：

```
int f( int a );
void f( int a, int b = 1 );
```

考察下面的函数调用表达式：

```
f( 10 );
```

从语法上说，这个调用表达式可以调用上述两个函数中的任何一个。由于出现了这种二义性，所以函数 f 的重载声明是错误的。

在面向对象的程序设计中，名字的重载是为了表达行为共性的一种重要手段。因此让重载函数执行不同的操作是一种非常不好的习惯。例如，如果一个名为 swap 的函数用于交换两个值，而另一个名为 swap 的函数则用于比较两个值的大小，这种设计思想是非常错误的。

多态性是面向对象的程序设计中最重要的术语之一。多态性的一般含义是指，某一论域中的元素可以有多种解释。例如，函数重载的标识符可以用来代表多个函数抽象，这种多态称为一名多用多态。在类与对象的设计中还存在其他形式的多态，我们将在后面的章节详细讨论。

1.3.2 函数模板

考察上节重载函数 swap 的定义。这些函数的参数个数相同，实现代码也相同，但只是参数类型不同。Visual C++ 6.0 提供了处理这类问题的最有力的工具——函数模板。函数模板声明了一组函数，因此，函数模板是编写通用函数的有力工具。

一个函数模板声明由关键字 template 引出。其一般形式为：

```
template <模板参数列表> 函数声明
```

模板参数列表是由“class 标识符”或“typename 标识符”这样的项组成的列表，表中每个项用逗号隔开，class 或 typename 后的标识符代表类型参数。读者要注意的是，关键字 typename 是新增的，它用于替换较早的在 template 中使用的关键字 class，这避免了程序员对 class 的使用可能造成的迷惑。

例如，下面的函数模板定义了通用函数 swap：

```
template <typename T> void swap( T& a, T& b );
{
  T n = a;
  a = b;
  b = n;
}
```

此时如果存在两个函数调用

```
int a, b;
float c, d;
swap( a, b );
swap( d, e );
```

编译器将从模板生成函数定义：

```
void swap( int& a, int& b );
{
  int n = a;
  a = b;
  b = n;
}

void swap( float& a, float& b );
{
  float n = a;
  a = b;
  b = n;
}
```

以便表达式 swap(a, b) 和 swap(c, d) 能被求值。

因此，一个函数模板告诉 C++ 编译器一个特定类型的函效应该如何建立。当需要建立这个函数时，C++ 编译器从函数模板中自动生成这个函数。根据函被模板生成的函数称为模板函数。函数模板的使用简化了重载函数的定义。

考虑下面的函数模板。

```
template < typename T > T min( T a, T b )
{
  return a < b ? a : b;
}
```

当有函数调用

```
min( 126, 230 );
```

时，编译器从函数模板生成类型为

```
int min( int, int );
```

的模板函数。但对于下面的程序代码，编译器将给出错误信息：

```
int i;
char c;
min( i, c );
```

在上述函数调用语句中，编译器无法从函数模板生成形式为 min(int, char) 的函数。

若所定义的函数模板需要不同个数的参数，则可以声明为：

```
template < typename T > T min( T a, T b )
{
  return a < b ? a : b;
}

template < typename T > T min( T a, T b, T c )
{
  T t = min( a, b );
  return min( t, c );
}
```

当以下述方式使用函数模板时：

```
void f( int x, int y, int z )
{
  min( x, y, z );
}
```

其中的函数调用表达式将导致编译器使用第二个函数模板生成类型为

```
int min( int, int, int );
```

的函数。由于该函数的实现使用了第一个函数模板，因此，该函数的生成又导致函数

```
int min( int, int );
```

的生成。

通过显式声明一个非模板函数，能够阻止编译器根据函数模板为特定类型的函数生成模板函数。例如：

```
template < typename T > T min( T a, T b )
{
  return a < b ? a : b;
}

int min( int a, int b )
{
  return a < b ? a : b;
}
```

该函数声明语句使编译器不再产生此种类型的模板函数，所声明的函数将与模板函数一同参与重载区分。例如：

```
void f(int i, char c, float f )
{
  min(i, i );  // 使用模板函数 min( int, int )
  min( f, f );  // 使用模板函数 min( float, float )
  min( i, c );  // 使用非模板函数 min( int, int )，c 被自动转换成 int 类型的值。min( i, c ) 能够正确编译。
  //  如果没有显式声明此函数，则 Visual C++ 6.0 会产生一个类型不匹配的编译错误。
}
```

在一些情况下，通过显式声明一个非模板函数来阻止编译器从函数模板中生成模板函数是必须的。例如，在函数模板 min 的定义中，操作符“<”对字符串的比较将产生不期望的结果，为此，我们可以做这样的定义：

```
#include <string.h>

template < typename T> T min( T a, T b )
{
  return a < b ? a : b;
}

char * min( char *a, char *b )
{
  return strcmp( a, b ) > 0 ? a : b;
}

CString min( CString a, CString b )
{
  return a < b;
}

int caller()
{
  char *szStr1 = "ABCDEFG";
  char *szStr2 = "ABCDEFH";
  min( szStr1, szStr2 );
  CString Str1( "ABCDEFG" );
  CString Str2( "ABCDEFH" );
  min( Str1, Str2 );
  return( 0 );
}
```

在上面的程序中，当对 szStr1 和 szStr2 进行比较时，将使用函数 min(char *, char *)；当对 Str1 和 Str2 进行比较时，将使用函数 min(CString, CString)。它们都不会使用从函数模板生成的模板函数。

当存在同名的函数模板和非模板函数时，编译器使用下述原则确定应调用的重载函数：

查找完全匹配的函数，如果找到，则调用之；否则查找某个函数模板，该模板可生成一个与调用完全匹配的模板函数；否则使用正常的重载函数解决方案。

如果仍不能确定调用哪个重载函数，则给出错误信息。

一个函数模板的“模板参数列表”中出现的参数必须被用于声明形参。只有这样，编译器才能够通过函数调用为模板参数推导出实现类型。因此，下述两个函数模板声明都是错误的。

```
template < typename T > T f()
{
// ......
}

template < typename T > void g()
{
  T a;
// ......
}
```

为使 C++ 编译器能够根据函数模板中生成模板函数，编译器必须在该函数调用之前知道该函数模板的声明。因此，模板声明通常出现在头文件中。

读者要注意的是，如果在函数模板中声明了静态对象，则每一个生成的模板函数都是有对应的静态对象。例如：

```
template < typename T > void g( T *p )
{
  static T t;
// ......
}

void f( int a, char *b )
{
  f( &a );  // 带有一个 int 类型的静态对象 t
  f( &b );  // 带有一个 char * 类型的静态对象 t
}
```

除函数模板之外，Visual C++ 6.0 还实现了类型模板。有关类型模板的内容，请读者参阅第 2 章。

1.4　小　结

本章讨论了 C++ 对 C 的改良和扩充，这使 C++ 作为 C 来使用时比 C 更方便安全，本章主要包括两个部分，其一是强制类型转换，其二是函数模板。基于抽象数据类型和类的概念，

函数抽象在 C++ 面向对象的程序设计中没有独立存在的理由，只能在抽象数据类型的实现中找到它们。

C++ 保持了与 C 的高度兼容性。在 C 中的做法在 C++ 中仍可以使用，C++ 程序也可以使用已有的 C 库，这使 C 程序员可以很容易地转向 C++。和 C 相比，C++ 提供了更强的类型检查能力，这减少了 C++ 软件系统中的类型错误。因而，C++ 给人的第一印象是一个更好的 C。

然而，C++ 最重要的方面是支持面向对象的程序设计。C++ 既支持过程程序设计方法，也支持新的面向对象的程序设计方法。C++ 的这种双重性或称混合性，给 C++ 程序员提出了特殊的挑战，即不仅要熟悉 C++ 的语法，而且也应自觉地熟悉面向对象的方法——一种解决问题和思考问题的新方法。这也意味着，在使用 C++ 设计面向对象的风格的程序时，在 C 中所习惯使用的思维方法在 C++ 中已不适用。要真正发挥面向对象的 C++ 的强有力特点，最重要的是要理解 C++ 所支持的程序设计原理，而不仅仅是 C++ 的语法。在 C++ 中，实现这种程序设计原理的关键机制是类——一种封装数据元素和对这些数据元素进行操作的函数的用户定义类型。另外需要指出的是，C++ 语言仅仅是一种工具，使用 C++ 的理由不是针对实际的语言本身，而是由于系统的分析、设计和实现被紧密地联系在一起，在分析、设计和实现之间有良好的过渡。

第 2 章

再谈面向对象程序设计

我们知道，当使用 Visual C++ 6.0 的 AppWizard 创建应用程序框架时，Visual C++ 6.0 自动生成的应用程序框架是由多个类组成的。事实正是如此，C++ 程序和 C 程序的本质不同在于，C++ 程序是由一组类组成的，而 C 程序则时由一组函数组成的。当设计 C 程序时，我们的任务设计完成特定功能的函数；而在设计 C++ 程序时，我们的任务则是设计完成特定功能的类。类是语言中表达抽象的有力工具。

封装和数据隐藏是面向对象范型的核心。类（class）是 C++ 用于封装和数据隐藏的工具，类将一个操作集与一个数据结构紧密地结合在一起。本章讨论 C++ 的类特征，其中部分内容在《Visual C++ 6.0 程序设计——入门篇》中有简要的讨论，我们这里将深入展开之。

2.1 类与对象

在 C++ 中，类的对象在程序语句中使用对象声明语句来声明，例如，要声明 TPoint 类的对象，我们可以使用语句：

```
POINT thePoint;
thePoint.X = 100;
thePoint.Y = 200;
TPoint MyPoint( thePoint );
```

其中，MyPoint 为所声明对象的标识符，TPoint 是 MyPoint 的类型。

2.1.1 类的构造

现在给出一个例子来说明类的构造。考虑屏幕上的点，为了与 Visual C++ 6.0 实现的类 CPoint 相区别，我们使用类 TPoint 类来描述一类点对象：

```
// MyPoint.h
class TPoint
{
public:
  TPoint( POINT thePoint );
  int GetDimX();
  int GetDimY();
```

```
    void Move( int xOffset, int yOffset );
    void MoveTo( TPoint thePoint );
  private:
    POINT Point;
  };
```

这是类的规范说明部分，上面的 TPoint 类定义了点的行为，即可以移动、可以获取点位置坐标，这些操作在公有部分声明。成员函数 TPoint 与类同名，它称为构造函数，在创建对象时起特殊的作用——初始化对象的数据成员。

这个类描述了点的数据结构表示，即使用 TPoint 类型的数据成员记录点的 X 坐标和 Y 坐标。数据的表示保存在类的私有部分。

下面是 TPoint 类的实现部分：

```
// MyPoint.cpp
#include "MyPoint.h"
TPoint::TPoint( POINT thePoint )
{
  Point.X = thePoint.X;
  Point.Y = thePoint.Y;
}

int TPoint::GetDimX()
{
  return Point.X;
}

int TPoint::GetDimY()
{
  return Point.Y;
}

void TPoint::Move( int xOffset, int yOffset )
{
  Point.X += xOffset;
  Point.Y += yOffset;
}

void TPoint::MoveTo( TPoint thePoint )
{
```