青松

Visual C++ 6.0 编程经验和技巧

杜经农 陈淑贤 / 编著

青岛出版社

Visual C++ 6.0 编程经验和技巧

杜经农 陈淑贤 编著

青岛出版社

鲁新登字 08 号

图书在版编目(CIP)数据

Visual C++ 6.0 编程经验和技巧/杜经农等编著 .- 青岛：青岛出版社，1999.9
ISBN 7-5436-2029-4
Ⅰ. V…
Ⅱ. 杜…
Ⅲ. C 语言-程序设计
Ⅳ. TP312

中国版本图书馆 CIP 数据核字（1999）第 29574 号

书　　名	Visual C++ 6.0 编程经验和技巧
编 著 者	杜经农 陈淑贤
出版发行	青岛出版社
社　　址	青岛市徐州路 77 号（266071）
邮购电话	（0532）5835844 5814750 5814611-20
责任编辑	樊建修　金利鹏
装帧设计	申　尧
印　　刷	山东莒县印刷厂
出版日期	1999 年 9 月第 1 版，1999 年 9 月第 1 次印刷
开　　本	16 开（787×1092 毫米）
印　　张	11.5
字　　数	250 千
印　　数	1—5000
ISBN	7—5436—2029—4/TP・235
定　　价	19.00 元

第一章　类和 MFC 基本类库

在本书开始时，首先要向您介绍类和面向对象的编程方法。如果您是 C++编程老手，已完全熟悉这些概念，那么您可以大概浏览本章，否则您应该仔细阅读这部分内容。因为本书是以面向对象的原则来组织的，我们谈用 VC++编程实质上是使用 MFC 类编程。

第一节　类的定义

“类”如同结构一样，是一种数据类型。不过它与结构相比是有区别的，类是包含成员变量和成员函数的由用户定义的数据类型。类的成员变量可以是任意数据类型，包括用户定义的其他类或结构。成员函数用于操纵成员变量来执行一定的操作。当我们自己定义了一个类时，实质上是定义了一种新的数据类型。

类的定义可用 Class 语句，其格式如下：

Class［类名］
{
［存取标识符］
定义成员
定义成员
……
……
};

类的成员可以是成员变量或成员函数。其存取标识符决定了该成员是在类的内部还是在外部被调用。存取标识符有三个：Public、Protected 和 Private，我们将在后面介绍它们的含义。下面我们先来定义一个类，并在后面讨论它的各个组成部分，以帮助大家了解类的概念。

```
Class man
{
 Public:
  int m_WorkDag;
  char m_Name[8];
 Protected:
  int m_YofBirthday;
 Private:
  float m_Money;
 Public:
 Man(int workday, int   birthday, char * name)
{
```

```
 m_WorkDay=workday;
 m_YofBirthday=birthday;
 strcpy(m_Name,name);
}
~Man(){}
int   getpayment()
{
 return   m_WorkDay*3;
}
};
```

如上我们定义了一个类(Man)，接下来让我们来看看它的内部结构并学习一些相关概念。

第二节　构造函数和析构函数

在类 Man 的内部有两个用于特殊目的的成员函数。一个是 Man()，它是一个构造函数。当您声明一个类型为 Man 的新变量时，会自动调用该构造函数。如果您在构造函数中加入了初始化代码，那么您在声明一个该类型的新变量时，类的构造函数就自动对该类进行了初始化。如：

Man　LiMing(6，1972，"liming")；

这条语句声明了一个 Man 类的变量 LiMing，同时传给构造函数三个参数：6、1972 和 liming。构造函数执行后，LiMing 的成员变量 m_workday 初始化为 6，m_YofBirthday 初始化为 1972，m_Name 初始化为 liming。构造函数不能有返回值，所以无需在其前加 void，构造函数的名字应与类名相同。

另外，Man 类中还有一个函数～Man()，它是一个析构函数。当类变量 LiMing 超出其作用域时，C++会自动调用其析构函数。析构函数既不带参数，也不返回任何值。它必须与类同名且其前有一个～符号。如果类中的构造函数和析构函数不含任何代码、任何事也不做，可以不必创建它们。

其他成员函数

Main 类中还有一个成员函数 getpayment( )。成员函数也用点(.)操作符来调用。如：

```
Man   LiMing(6，1972，"LiMing")；
int payment；
payment=LiMing.getpayment( )；
```

第三节　类的成员变量

一个类可以有一个或多个成员变量，成员变量可以是任意定义的数据类型。Man 类中有四个成员变量，分别是 m_WorkDay、m_Name[8]、m_YofBirthday、m_Money。类的成员变量用点(.)操作符来存取。如下段程序所示。

```
Man  LiMing(6，1972，"LiMing");
int payment;
payment=LiMing.m_WorkDay*3;
```

在我们声明了一个 Man 类的变量 LiMing 后(有时也称“声明 Man 类的一个实例 LiMing”，或称为“声明 Man 类的一个对象 LiMing”)，即可调用其成员变量 LiMing.m_WorkDay。

第四节　存取标识符

我们注意到在声明类的成员时首先应有一个存取标识符。什么叫存取标识符呢？存取标识符是定义类的可存取性的操作符。存取标识符后面应有一个冒号(：)。一旦在类定义中出现一个存取标识符，其后所有的类成员都被规定了相应的存取属性。存取标识符一直有效直至另一个存取标识符出现或类定义终止。存取标识符有三个，分别是 Public、Private 和 Protected。下面我们介绍 Public 和 Private 存取标识符。关于 Protected 存取标识符，我们在谈到类的层次时将介绍。

一、Public 存取标识符

Public 存取标识符指明其后出现的是类的公有成员。对于类的公有成员来说，可以在类定义的外部调用它。如以下语句：

```
main()
{
 Man LiMing(6，1972，"LiMing");
 Cout<<LiMing.m_Name;
}
```

上述语句是正确的，因为 m_Name 是类 Man 的公有成员。

二、Private 存取标识符

Private 存取标识符指明其后出现的是类的私有成员。如果一个类的成员规定为 Private，不管它是一个变量还是函数，在类的外部都不能调用它，下面的语句段是错误的：

```
main()
{
 Man LiMing(6，1972，"LiMing");
 Cout<<LiMing.m_Money;
}
```

因为 m_Money 是类的一个私有成员变量，所以不能在类的外部 main 函数中调用类变量 LiMing。但是在类的内部，类的成员函数中是可以用它的。

有些变量，假如您出于安全性考虑，希望只有得到验证后才能在类的外部改变它，那么可以定义它为私有变量。同样，类的某些成员函数仅用于完成内部操作，类的对象不需要调

用它，较好的办法就是将它定义为私有。对于类的私有变量，如果需要改变它可以在类中定义公有成员函数来存取它。例如给类 Man 增加以下公有成员函数：

```
Public:
Void   SaveMoney(float money)
{
 if(money>m_Money)
 { m_Money=money;}
}

float   GetMoney(char * name)
{
 if(strcmp(name, " LiMing " )= =0)
 { return   m_Money; }
 else
 {return 0.0;}
}
```

这两个公有成员函数可以在类的外部操作变量，它们根据传递给自己的参数值来决定如何操作类的私有成员 m_Money。

第五节　类指针

类是一种数据类型。因此它和 C++语言的内部数据类型一样，可以声明该类型的指针。例如以下语句：

```
Man   LiMing(6，1972，" LiMing " );
Man*pLiMing=&LiMing;
```

第二条语句声明了一个指向 LiMing 的 Man 类型的指针 pLiMing。如果要用指针来存取类的成员，应使用指针操作符(->)，如下面的例子。

```
int   payment=pLiMing->getpayment();
int   birthday=pLiMing->m_YofBirthday;
```

要声明一个类指针还有另一种办法，即使用 new 操作符，观察如下语句段。

```
Man*pLiMing;
pLiMing=new   Man(6，1972，" LiMing " );
```

此语句段同样生成了 Man 类的一个新对象，并将该对象的地址存于 PLiMing 指针中。要注意的是，new 操作符的作用是为对象分配内存。在对象超出其作用域后，对象占用的内存并未释放，因此当您不需要该对象时，必须用 delete 操作符释放该对象占用的内存：

```
delete   pLiMing;
```

第六节　类变量的传递

当我们向某函数传递变量时可以有三种方法：复制传递、按地址传递和引用传递。这和 C++的内部变量类型一样。下面，我们来比较一下这三种传递方式的不同。

以下是复制传递的例子，复制传递也称为按值传递。我们设想有一个函数 ChangWorkday，我们向它传递一个 Man 类的参数 theman。

```
void ChangeWorkday(Man  theman)
{
 theman.m_WorkDay=5;
}
```

我们现在调用该函数，来改变一个 Man 类对象的 m_WorkDay 成员变量。

```
Man  LiMing(6, 1972, "LiMing");
ChangeWorkday(LiMing);
```

复制传递时，类的变量 LiMing 被复制了一份传递给函数。这种复制的代价是昂贵的，因为复制一个类变量时，类的所有成员变量都要复制，传给被调函数的是类变量的一份复制品，这将导致程序运行速度的降低。使用复制的办法传递参数时，当被函数返回后，对形参做的改变不会影响到实参的值。这是因为函数返回时，类变量的复制品被取消，类变量还是原来的值。所以上述语句段达不到它的目的。Changeworkday(LiMing)函数执行完后，LiMing.m_Workday 的值仍是 6，而不会变成 5。

通过以上讨论，我们可以发现复制传递有两个问题：

- 影响程序执行速度。
- 如果您想在被调函数中改变实参，复制传递无法做到。

要解决这两个问题，我们必须使用另两种传递参数的办法：地址传递和引用传递。首先介绍地址传递。

我们定义如下函数：

```
Void  Changeworkday(Man *theman)
{
 theman->m_WorkDay=5;
 return;
}
```

然后调用它：

```
Man  LiMing(6, 1972, "LiMing");
Changeworkday(&LiMing);
```

此时，传递给函数 Changeworkday 的是类的变量 LiMing 的地址(&LiMing)，而不是复制了的变量。同时存储类成员变量 LiMing.m_WorkDay 的内存地址处的内容也由 6 变成了 5。所以，实参 LiMing 的成员变量也 m_WorkDay 也由 6 变成了 5。

“引用”是 C++的一种新的变量类型，引用传递的效果和按地址传递一样，但是它的语法结构更清楚易懂。函数应如下所示：

```
Void   Changeworkday(Man & theman)
{
  theman.m_WorkDay=5;
}
```

调用该函数的语法如下：

```
Man   LiMing(6，1972，"LiMing");
Changeworkday(LiMing);
```

可以看出：调用函数时语法与复制传递一样，只是在声明函数原型时在形参前加一个&符号。

第七节　函数的重载

一个类的某个成员函数可以有多个版本，它们都有相同的函数名，只是参数不同，这称为函数重载。除了析构函数外，类的其他成员函数均可以重载。C++根据您传递给函数的参数来决定该使用函数的哪个版本。在 MFC 中您可以看到许多重载函数，它允许您以很多不同的方式来使用函数，这样就方便了您的编程。

重载函数较常见的是用在构造函数上，下面看看我们如何重载 Man 类的构造函数。

```
Class   Man
{
 Public:
 int   m_WorkDay;
 char   m_Name[8];

 Protected:
 int   m_YofBirthday;

 Private:
 float   m_Money

 Public:
 Man()
{
 m_WorkeDay=5;
 strcpy(m_Name, "WangHui");
 m_YofBirthday=1977;
}

Man(int workday，int birthday，char * name)
{
 m_WorkDay=workday;
```

```
 m_YofBirthday=birthday;
 strcpy(m_Name,name);
}

Man(Man & theman)
{
 m_WorkDay=theman.m_WorkDay;
 m_YofBirtday=theman.m_YofBirthday;
 strcpy(m_Name,theman.m_Name);
}
~Man(){}
};
```

我们看到，上述 Man 类的构造函数被重载了，它有三个版本。第一个 Man()不带任何参数，它称为缺省构造函数。在声明一个类变量时如果不传给它任何参数，将会调用缺省构造函数。例如，下面两个声明都会引起缺省构造函数的调用。

```
Man   aman;
Man   aman();
```

第二个构造函数有三个参数，如果我们声明类对象时，传给它对应的参数将会引起对这个构造函数的调用，如下所示：

```
Man   LiMing(6，1972，"LiMing");
```

第三个构造函数有一个参数。这个参数很特殊，它是同一类的引用变量。这样的构造函数称为复制构造函数。如果定义了一个类的复制构造函数，那么 C++在需要复制类变量时会调用该复制构造函数。例如当您以复制传递的方式将类变量传给某函数时。

第八节　使用多个文件来定义一个类

要定义一个类还有另一种方法。在 Class 语句中，我们仅声明类的成员变量和类的成员函数，而在 Class 语句外编写类的成员函数的实际代码。如下所示，我们用这种方式来定义 Man 类。

```
Class   Man
{
 Public:
   int m_WorkDay;
……其余成员变量
Public:
 Man(int workday，int birthday，char * name);
……其余成员函数
};
Man::Man(int workday，int birthday，char * name)
{
 m_WorkDay=workday;
 m_YofBirthday=birthday;
```

```
 strcpy(m_Name, name);
}
……其余成员函数
```

上述定义中有两点需要强调：

① 声明类的成员时，不管是声明一个成员变量还是声明一个成员函数的原型，其语句段末尾都要使用分号(；)。

② 在 Class 语句外定义类的成员函数代码时，每个成员函数前都要加上类名和把类名与函数名分开的双冒号(::)。这个双冒号是作用域分辨操作符，该操作符表明函数是该类的一个成员函数。在 Class 语句内部定义类的成员函数时，不用这个操作符。

使用这种方式定义类时，通常是把 Class 语句放入一个头文件中，如 man.h 文件。有时也称类的接口。把成员函数的实现代码放在一个 CPP 文件中，这个文件通常被称为类的“实现文件”。要注意的是，类的实现文件中要用 #include 语句将其头文件包含进来。如下所示，我们在两个文件中定义了 Man 类。

```
//man.h   Man 类的头文件
Class    man
{
 Public:
 int m_WorkDay;
 char m_Name[8];
 Protected:
 int m_YofBirthday;
 ~Man();
 Public:
 Man(int workday,int birthday, char * name);
 int getpayment();
};

//man.cpp Man 类的实现文件
#include "man.h"
Man::Man(int workday, int birthday, char * name)
{
 m_WorkDay=workday;
 m_YofBirthday=birthday;
 strcpy(m_Name,name);
}
Man::~Man(){}
int Man::getpayment()
{
 return   m_WorkDay*3;
}
```

用多文件来定义类可以方便的组成庞大的类库；通过将接口和实现代码分开可以保证类库的安全性。微软的基本类库(MFC)就是用这种方式来组织的。

类除了有强大的内部结构外还有一些其他的特性，如继承性和多态性。它们是组成类库的基础，也是面向对象程序设计方法的核心。下面我们来分别介绍。

第九节　类的继承性

如果您想扩充现有类的性能，给它加一些新的成员变量和成员函数。您可能会想到直接修改类文件，即类的头文件和实现文件。但这样做有很多不便之处，例如，您在修改时可能会无意改动了类的原有代码。更有可能的是，该类是别人所写，您只有头文件而没有实现文件的原代码(实现文件已被编译成二进制的类库了)。

为解决这样的问题，C++给类规定了一种特性，即继承性。一个类可以由另一个类继承而来，我们也称这个类从其他类派生出来。派生出的类叫子类，而原始的类叫父类。子类可以从父类中继承一切特性，包括其所有的成员函数和成员变量。假设我们从 Man 中派生出另一个类 YangMan，则其语法格式如下所示：

```
Class YangMan：Public　Man
{
.....
};
```

这个语句使 Man 类派生出了 YangMan 类。基类与派生类之间的冒号表明：冒号前的类是由冒号后的类派生出来的。Public 关键字表明：在 Man 中被定义成公有的成员变量和成员函数对 YangMan 类而言也是公有的。如果您声明了一个变量为 YangMan 类型，则这个变量不但可以存取 YangMan 类的可存取成员，也可以存取基类变量能够存取的成员。下面，我们写一段实际的代码来说明这些问题。

```
Class YangMan：Public　Man
{
 Public：
 int m_Cloths；

 Private：
 Char m_Dream[50]；

 Public：
 YangMan(int work，int birth，char *myname)：Man(work，birth，myname){}
 int　getage()
 {
 int age=1999-m_Birthday；
 return　age；
 }
};
```

现在我们来看看派生类的结构，同时学习两个新知识。

一、构造函数

在我们从 Man 类派生出 YangMan 类时，使用了如下形式的构造函数：

YangMan(int work，int birth，char*myname)：Man(work，birth，myname){}

这个构造函数中使用了冒号(：)操作符，这个冒号表明：在任何时候需调用派生类 YangMan 的构造函数时，要先用传递给 YangMan 类构造函数的参数来调用 Man 类的构造函数，接着再执行自己的构造函数。下列语句将引起基类构造函数的调用：

```
YangMan  XiaoLi(5，1978，"XiaoLi");
```

二、Protected 存取标识符

前面我们谈到，基类的公有成员对派生类来说也是公有的。不但可以在类的内部，也可以在类的外部使用。但对于基类的私有成员来说，派生的类是不能使用的，不但在类的外部不能使用，就是在派生类的内部，如派生类的成员函数中也不能使用。

我们知道类还有一种存取标识符，即 Protected(保护)存取标识符。当您定义了某个类成员为 Protected 存取类型时，您可以在该类的派生类的内部使用它，例如在派生类的成员函数内使用。但是不论是基类还是派生类的变量，都不能在类的外部使用它。下面我们列表来说明这三种存取标识符定义的存取属性。

三种存取标识符的存取属性

存取标识符	基类的存取权力	派生类的存取权力
Public (公有)	可在类的内部和外部使用自己的公有成员	可在类的内部和外部使用基类的公有成员
Protected (保护)	可在内部使用，不能在外部使用自己的保护成员	在内部使用基类的保护成员，不能在外部使用
Private (私有)	可在内部使用自己的私有成员，不能在外部使用	不论在内部还是外部，都不能使用基类的私有成员

第十节　虚函数

我们在派生类中可以定义与基类中一样的成员函数，这称之为重载成员函数。要在派生类中重载一个基类函数，可以在派生类中说明一个相同的函数原型，并在函数体中写入不同的代码。如下所示：

```
Class  Man
{
 Public:
 int Get Payment(){   return m_WorkDay*30;  }
          ……
};
Class  YangMan: Public  Man
{
 Public:
 int GetPayment(){ return m_WorkDay*20;  }//重载函数
          ……
};
```

如果您定义了一个派生类的变量，并使用该变量调用 GetPayment()函数。那么，调用的将是派生类中定义的成员函数。如下面的语句：

```
int   Payment
YangMan   XiaoLi(5，1978，" XiaoLi " );
Payment=XiaoLi.GetPayment();
```

第三条语句执行完后，Payment 中的值将是 100(5×20)，而不是 150(5×30)。

接下来我们讲解类的多态性。一个类的可以派生出多个子类，这些子类都具有一些相同的成员。但是，每个子类也可能有自己特有的成员，也可能重载基类的成员。现在，假设我们从 Man 中派生出了两个子类 YangMan 和 OldMan，这两个子类都重载了基类的 GetPayment 成员函数，如下所示。

```
Class   YangMan：Public   Man
{
 Public：
 int GetPayment(){ return m_WorkDay*20；}
          ……
};

Class   OldMan：Public   Man
{
 Public：
 int GetPayment(){ return m_WorkDay*40；}
          ……
};
```

在派生类与基类之间有一种“IS-A”关系。也就是说：一个派生类是一个基类。如“一个 YangMan 即是一个(is-a)Man”。但反过来是不行的，您不能说一个 Man 即是一个 YangMan。我们知道 C++中不允许某一种类型的数组去存储另一种类型的变量。但是，由于派生类与基类有 IS-A 关系，因此我们可以用一个基类的数组来存储两个派生类的变量。如下所示：

```
Man* theman[2]；
YangMan   XiaoLi(5，1978，" XiaoLi " );
OldMan   LaoWang(5，1945，" LaoWang " );
theman[0]=&XiaoLi；
theman[1]=&LaoWang；
```

现在 theman[0]指针指向 XiaoLi，而 theman[1]指针指向 LaoWang，假如我们想用这个指针数组来输出每个人的 payment 值。我们当然想到要用一个 for 循环来遍历数组，如下：

```
for(int i=0；i<2；i++)
{
 Cout<<theman[i]->GetPayment()<<" \n " ；
}
```

那么，输出应该是怎样的呢？由于 GetPaymet()函数在派生类中被重载过，我们会认为将调用派生类的函数来输出。但实际上不是这样，输出如下：

```
150
150
```

可见，每次循环都调用了基类成员函数 GetPayment。当用数组指针来调用重载函数时，即使指针指向派生类，调用的仍是基类的相应函数。但是这可不是我们想要的效果，我们希望此时能调用派生类的重载函数。要达到这个目的，C++引入了虚函数的概念。

如果您要定义一个成员函数为虚函数，只需在该函数的原型前加一个 Virtual 关键词即可。如下所示，我们将 Man 类的 GetPayment 函数定义成为虚函数。

```
Class  Man
{
 Virtual  int  GetPayment()
          ......
};
```

将 Man 类的 GetPayment 函数定义为虚函数后，再运行前面的代码段。此时数组指针将会调用派生类的重载函数 GetPayment，输出应为：

100

200

完成此类操作的能力称为多态性(polymorphism)。C++正是通过虚函数的引入实现了类的多态性。

本章我们讨论了类的继承性和多态性，这两个特性使类具有了层次，一个类可以派生出它的子类，而这个子类又可以再派生出它的子类。子类能继承父类的一些特性，但子类又有自己独特的特性。我们使用继承性和多态性的技术可以构建出丰富而强大的类库，我们在VC++编程中使用的 MFC 类库就是构建在这两种技术上的。

第十一节　面向对象的程序设计

正如 C 语言将问题分解成多个函数一样。在面向对象的程序设计中，一个完整的程序就是一些解决某个问题的对象的集合。什么是对象呢，对象就是某个类的变量。例如，当您定义 mystring 为 MFC 类库中 Cstring 类的变量时，就称为：定义了 Cstring 类的一个对象 mystring。面向对象的编程方法对问题进行了抽象和封装，允许您从问题的实质来考虑如何解决，而不是按计算机处理事物的方式来思考问题。在早期的机器语言和汇编语言中，您必须记住一条条的计算机指令。在考虑一个问题时，必须知道计算机内部发生了什么操作。这样，您被迫从计算机的角度来思考问题。在随后发展的高级语言中，如 FORTRAN 语言允许您以数学术语而不是计算机术语在计算机上解决数学问题；COBOL 语言允许您用自然语言来编写程序，它们都是一种抽象。而在类中，它以一种更强大的方式将数据和函数抽象地结合起来并封装在类的内部。Windows 系统是很复杂的基于消息处理的图形系统，要使用普通的 C 语言对 Windows 系统编程将是一件繁琐的事情。您要面对的 Windows API 函数也许有成打的参数，而整个程序的结构也很繁琐，必须写上许多行例行的代码。而在 VC++中您可以使用微软已编好的 MFC 类库。MFC 经过了精心的设计，抽象和封装了 Windows API 函数。同时，VC++提供了一个可以定制的程序框架，您所要做的事只是在框架中运用各种对象来解决某一问题。

在 VC++中，您应该学会用对象的观点来思考问题。例如：当您要在屏幕上输出时，应考虑用视图类；要存取磁盘文件时，应考虑用文档类。面向对象的程序设计方法的另一个好处是可以方便地重用代码。通过类的继承性和多态性，您可以从 MFC 基本类库中派生自己的类，并方便地对其功能进行重新定义、增强和改动。这样，您的类不仅有其 MFC 基类的所有功能，还能完成自己的特殊任务。如果微软对 MFC 基类的功能进行了扩充，那么您无需任何编程，您自己的类的相应功能也得到了扩充。本书举了大量例子、示范如何从 MFC 中派生自己的类，同时指导您在以后的编程中如何方便地使用这些类。

第十二节　什么是类库

就像函数库一样，类也可以汇集成库。类库的实现代码已经编译并保存在以.LIB 为扩展名的库文件中。类库的接口，即定义类的头文件，保存在扩展名为.H 的文件中。

为什么要将类汇集成类库呢？类的成员函数能封装 C++的代码完成特定的操作。例如微软提供的 CString 类就封装了所有处理字符串的功能。这样可以使您通过声明一个 CString 类，来方便地处理字符串。所以说类库能够简化操作，使您能更方便地设计程序。此外，由于类具有继承性，所以您可以扩充或修改库的功能而不必改变或拷贝任何原代码。在以后的章节中，我们将演示如何扩充 MFC 库的功能。要在程序中使用一个类库，应在程序中包含其头文件(.H)，同时应告诉编译器，在编译时要链接到相应的库文件上(.LIB)。

第十三节　MFC 库

使用 VC++编程，实际上就是使用 MFC 库编程，它可以大大简化 Windows 程序设计。MFC 库是微软编写的一套基本类库，MFC 就是 Microsoft Foundation Classes(微软基本类)的缩写。MFC 是微软专门为封装 Microsoft Windows API 而设计的。API 是 Application Programming Interface 的缩写，即应用程序编程接口。Windows API 是非常复杂的，微软为其编写的程序员指南就有三大本厚厚的书。微软根据通常程序要完成的功能将它们组织成不同的类，每个类都能完成一定的功能。在您进行程序设计时，通过组合使用这些类，最终得到了您想要的、能完成特定操作的程序，这就是 VC++编程的实质。当然，如果您愿意的话，也能在 VC++编程中使用 API 函数或普通的 C++函数。实际上，由于 MFC 并没有封装所有的 Windows API 函数。因此您进行 VC++程序设计时，有时会被迫直接使用 Windows API 函数来完成某些特殊的任务。

几乎所有的 MFC 类都是从一个名为 CObject 的基类中派生出来的。CObject 包括了一些您可能会在自己的类中使用的特征，如串行化输入和输出，它用来读写磁盘。当您使用 VC++的应用程序向导(AppWizard)来为您生成一个应用程序框架时，它使用了几个用于程序基本结构的类。这些类组合在一起形成了一个基本的体系结构并支持整个程序设计以及协调数据存取、给用户的显示和命令发送的工作。下面我们来谈这些类。

一、应用程序框架类

(1) CWinApp

所有的 MFC 应用程序都必须有一个而且只有一个 CWinApp 对象。该类中封装了应用程序的所有基本要素。它包括了您已熟悉的 main()函数的等价形式。实际上 Windows 应用程序并没有一个 main()函数。Windows 程序的主函数叫做 winmain()。winmain()在 Window 应用程序中所起的作用与 main()在 C 语言程序中所起的作用完全相同。但是使用了 MFC 以后就不再编写 winmain()了，这是因为 winmain()已经封装在类 CWinApp 类当中了。

(2) CCmdTarget

Windows 是一种消息驱动的环境。用户的动作会产生一条反应什么动作被截取的消息，而 Windows 程序则通过接收消息和根据消息的内容采取相应的动作来与用户相互作用。MFC 应用程序接收的消息大多数都是所谓的命令消息(Command Messages)，因此，所有那些接收消息并处理消息类的基类就取名为 CCmdTarget(Class Command Target)。决不能在程序中直接使用 CCmdTarget，而只能使用那些由 CCmdTarget 派生出来的类。

(3) CDocument

所有应用程序都必须有数据，否则它就没有存在的意义。大多数的应用程序都需要某种方法来管理数据，诸如从磁盘文件之类的数据源读出数据及向磁盘或其他介质写入数据。CDocument 的作用便是协调应用程序中所有的数据管理任务。不要直接使用 CDocument，最好是从 CDocument 派生出自己特殊的类版本以适合自己的应用程序的需要。

(4) CView

大多数应用程序都必须有一种方法来显示其数据，并要能随着输入而变动，即与数据交互作用。CView 是派生出您自己的数据视图类的一个基类，它包含了可帮助您进行用户和应用程序数据之间的交互管理所必需的功能。

(5) CDocTemplate

该类派生出两个子类：CSingleDocTemplate 和 CMultiDocTemplate。这些模板(Template)类协调新文档和新视图的创建。如果应用程序是一个单文档界面(SDI)应用程序，就应使用 CSingleDocTemplate；如果应用程序是一个多文档界面(MDI)应用程序，就应使用 CMultiDocTemplate。

SDI 应用程序一次只能打开一个文档，并且只有一个主显示窗口。Windows Notepad 应用程序就是一种 SDI 应用程序。MDI 应用程序则可以同时有多个文档，并能同时在多个窗口中对打开的多个文档进行显示。

AppWizard 通过从上述类中派生出用于程序的子类来构建您的应用程序框架。关于应用程序框架的具体结构，我们将在第五章中向您具体阐述。

二、管理 Windows 界面和显示的类

Windows 是一种高度可视化的面向对象的环境。因此 Windows 有许多类型的对象需要显示。在 Windows 中的各种类型的显示对象都必须出现在一个矩形区域内，这个矩形区域就是通常所说的窗口。MFC 中包括有众多支持各种固有 Windows 可视对象显示的类，还包括在屏幕或其他输出设备上绘图的类。MFC 中有用于显示实际的窗口、菜单、工具栏、对话框和应用程序中的各种控件。这些类中也包括了诸如画笔、画刷和设备描述表等用于绘画实际窗口中内容的各种对象。

(1) CWnd

CWnd 是所有 MFC 中各种窗口的主要基类。MFC 中已有大量从 CWnd 类派生出来的