Development of Applications with VC/MFC

何友鸣

VC/MFC

应用程序开发

湖北科学技术出版社

前　　言

Visual C++ 6.0 是 Microsoft　Visual Studio 6.0 的成员之一,是公认的开发 Windows 程序的利器。即使是在微软推出 Visual Studio .Net 开发平台之后，仍然将 Visual C++作为重要的开发工具，而且提供了由 Visual C++ 6.0 向 VC++.Net 转化的途径及相应的工具。这充分说明了 Visual C++ 具有强大的生命力。

1. 为什么偏偏是 VC

对于计算机编程语言而言，C 语言拥有最大多数的用户群，很多人接受学校的计算机基础教育也是从学习 Turbo C 开始的。对于随后出现的面向对象的编程语言 C++，不少人最初是把它当作 C With Classes 来接受的(虽然随着学习的深入，这个观点就显得不那么正确了)。正是由于这样的原因，大家都觉得从 C 过渡到 C++是自然的。

Visual C++ 6.0 是微软的 Windows 平台下 C/C++编译器的最新最好的版本，而且它远非只是一个编译器。它还包括了微软基本类库（MFC Library），这是一个真正称得上应用程序框架（Application Framework）的东西，这使得开发 Windows 应用程序变得简单而高效；它还有复杂的资源编辑器，可以编辑对话框、菜单、工具栏、图标和其他许多 Windows 应用程序的组成元素；另外它的集成开发环境——Developer Studio，可以在编写 C++程序时对程序的结构进行可视化的管理，它的完全集成的 Debug 工具可以让你从各个角度来检查程序运行中的微小细节。上面提到的还只是 Visual C++ 6.0 的众多特点中的一部分。正是由于 Visual C++的这些特点使得它能开发出一流的 Windows 应用软件，在 C++的使用群中占据了相当的份额。

相比 VB、Delphi 以及 Borland C++ Builder 等快速开发工具（RAD），VC 的功能显得更为强大。尤其是 VC 是微软制作的产品，与 Windows 操作系统的结合更加紧密。

2. 对于 VC/MFC 的常见误解

VC 不是什么新的语言,而是开发平台，它使用的是 C/C++语言,而且几乎没有进行任何扩展。网上常看到有人问应该学 C/C++还是 VC，对于这个问题的正确回答应该是：如果你学 VC 你就必须得学 C/C++。

MFC 是一个“无所不包”的应用框架(Application Framework)，这大致相当于 Delphi 和 Borland C++ Builder 的 VCL 或是 OWL。而所谓的应用框架，基本上你可以认为，是一个完整的程序模型，譬如文件存取、打印预览、数据交换……，以及这些功能的使用接口（工具栏、状态栏、菜单、对话框）。

C++本来有着广泛的使用范围和良好的可移植性。微软在设计 MFC 时，为了最大限度地减少对 C++语言本身的扩展，不得不在原代码级上下工夫，以便能尽可能支持 ANSI 等标准，结果导致 MFC 的封装复杂而不直观(尤其是它对消息的封装，这在以后会进一步提到)。也正是由于这个原因，很多人抱怨 VC/MFC 不容易上手，这大概是为可移植性和兼容性所付的代价吧。

3. 阅读本书的知识前提和本书的特点

在阅读本书之前，你应该具备 C++语言的基本知识和面向对象的基本观念，并至少有过

一些 C++的编程经验。如果你还能具备一些 Windows 编程的基本知识和经验，那么就能更好的理解本书。为了让所有的读者有共同的知识基础，本书前面章节将会对 C++语言和 Win32 程序的基本设计原理作一个概略的回顾。

本书力求做到简洁生动，条理清晰。所采用的实例多与财经相关。

我们衷心希望在本书的指导下，你能喜欢使用 Visual C++ 6.0 编程，并且通过你不懈的编程实践加深对于 Windows 程序开发的认识。

4．本书的内容组织

本书内容安排如下：

第一章分三个部分分别介绍了 C++面向对象的语言特征 、Windows 程序的基本概念和作为应用程序框架的 MFC。这三部分内容可以作为我们开始本书 VC/MFC 学习的共同起点。

第二章介绍了 VC 的开发环境和主要的使用工具。中间贯穿了两个 Hello 实例，分别是 DOS-LIKE 程序和 Windows 程序。希望能通过这两个实例使您较快地从原先使用 Tubor C 的环境过渡到使用 VC 的集成开发环境。

第三章介绍了如何开发基于对话框的程序。我们尽量在较少的实例中让你接触到更多的常用控件。

第四章介绍了设备环境和 GDI 对象。这部分内容几乎所有的应用都会涉及到，尤其在绘图和打印中。

第五章综合叙述了窗口的组成元素，包括图形资源、菜单、工具栏和状态栏，以及视图和框架等。

第六章介绍了文档/视结构，两类文档/视结构应用--SDI 与 MDI，以及文档的读写。这一章最后包含了不少实例，分别实证不同类别的应用。

第七章介绍了三类增强的文档/视技术，包括多视图，切分窗口和打印与打印预览。

第八章介绍了 VC 开发数据库应用的基础知识，包括使用 MFC 中的 ODBC 类，以及使用 Record 视图。

第九章介绍了动态链接库的的创建和使用。

第十章 COM 是 Microsoft 的基于组件的软件解决方案的基础，它的应用非常广泛。本章最后还介绍了作为 COM 技术应用的 ActiveX 控件的开发。

撰写本书时，中南财经政法大学信息学院杨云彦教授给予了大力支持，并得到了曾庆伟教授和刘腾红教授的关心和指导。张骥、金大卫和胡爱钰等研究生参加了部分文字工作，何苗、金鑫和何蓉等同学作校对工作，院办公室李贤勇、沈彤和张进老师等给予帮助，在此一并表示感谢。我们还感谢武汉大学计算机学院李晓红老师、罗敏博士、湖北工学院罗晓兰、楚天金报电脑版夏文编辑对书稿的审阅和指导。

武汉联柯制冷设备工程有限公司及总经理李援朝先生，对本书的编写与出版给予的积极协助与支持，也表示衷心感谢。由于我们的水平和能力所限，书中存在缺点和纰漏等，承蒙读者批评指正。

何友鸣　芦涌　方辉云

2002 年 7 月 10 日

目 录

第一章 重要的知识前提

这里的知识前提包括三个部分，第一个部分是C++本身的重要性质，尤其是其中的支持面向对象程序设计的性质。C++掌握的好坏直接影响到 VC 的深入学习。第二部分简要地介绍一下 Windows 程序设计的特点，这与在 DOS 环境下的程序设计差别很大，可以说涉及到编程思想的转变。不完成这个转变，就很难发挥 VC 的强大功能。至于第三部分的 MFC，前言中已经提到这是随 VC 编译器的应用程序框架（Application Framework），它真正地把面向对象程序设计方式融入了 Windows 程序设计。

第一节 C++的重要性质

“面向对象”是软件界的一种潮流。面向对象的程序设计（Object Oriented Programming ）其实是一种观念，用什么语言实现它都可以。当然，面向对象的程序设计语言（Object Oriented Programming Language）是专门为面向对象观念而发展起来的，能更方便地实现面向对象的封装、继承、多态等特性。

使用类和对象的概念进行封装可以隐藏实现细节，使得代码模块化；继承则可以用来扩展已存在的代码模块（类）；它们的目的都是为了——代码重用。而多态则是为了实现另一个目的——接口重用。

C++语言，虽然不象 Small Talk 和 Java 是一种纯粹的面向对象的语言，但由于它站在 C 语言的肩膀上，因而成为了最重要的面向对象的程序设计语言。

面向对象的设计与面向过程的设计是有很大区别的，甚至可以说这是一种思维模式的转变。这就是为什么有的人使用 C++ 语言，仍然编不出面向对象的程序的原因。学习 C++ 语言不仅要懂得语法（这可以通过学习任何一本 C++ 的教材或专著来达到），更要懂得语意，尤其要认识 C++ 语言面向对象的特性和实现面向对象的方法。

一、类与对象

1. 基本概念

面向对象的观念把世界看成是由对象组成的。任何实际的物体和抽象的概念都可以看作是对象。属性和方法分别描述了对象的静态和动态特征。从数据处理的角度看，程序设计中的类和对象为我们提供了数据封装的工具。在面向对象的程序设计中，将数据（C++中称为成员变量——member variable）和对该数据进行合法操作的函数（C++中称为成员函数——member function）封装在一起作为一个类的定义，数据将被隐藏在封装体中，该封装体通过操作接口与外界交换信息。对象作为类的实例，被定义成某一给定类的变量。

C++中的类有些类似于 C 语言中的结构，但 C 语言中的结构里可以包含数据，而不能包

含函数。C++中的类则是数据和函数的封装体。在 C++中，也有结构，但这是作为一种特殊的类，它虽然可以包含函数，但是它没有私有或保护的成员。

C++类中包含私有、公有和保护成员

C++类中可定义三种不同访问控制权限的成员。一种是私有(Private)成员，只有在类中说明的函数才能访问该类的私有成员，而在该类外的函数不可以访问私有成员；另一种是公有(Public)成员，类外面也可访问公有成员，成为该类的接口；还有一种是保护(Protected)成员，这种成员只有该类的派生类可以访问，其余的在这个类外不能访问。

一般而言，成员变量尽量声明为 private，成员函数则通常声明为 public。正是通过这种手段，声明为 private 的数据，不允许外界随意存取，只能通过特定的接口来操作，从而满足对象的封装特性。

下面给出一个日期类定义的例子：

```
class tdate
{ public:
    void setdate(int y, int m, int d);
    int isleapyear();
    void print();
    private:
    int year, month, day;
};
//类的实现部分
void tdate::setdate(int y, int m, int d)
{   year = y;
    month = m;
    day = d;
}
//判断该年份是否为闰年
int tdate::isleapyear()
{   return(year%4==0 && year%100!=0) || (year%400==0);
}

void tdate::print();
{   cout<<year<<"."<<month<<"."<<day<<endl;
}
```

这里出现的作用域运算符::是用来标识某个成员函数是属于哪个类的。

该类的定义还可以如下所示：

```
class tdate
{   public:
      void setdate(int y, int m, int d)
```

```
    {year=y; month=m; day=d;}

  int isleapyear()
    {return(year%4==0 && year%100!=0) || (year%400==0);}

  void print()
    {cout<<year<<"."<<month<<"."<<day<<endl;}

private:
    int yeay, month, day;
}
```

这样对成员函数的实现(即函数的定义)都写在了类体内，因此类的实现部分被省略了。如果成员函数定义在类体外，则在函数头的前面要加上该函数所属类的标识，这时使用作用域运算符::。

2. C++中通过发送消息来处理对象

C++中是通过一种类似于函数调用的机制把消息发送到一个对象上。接受消息的对象根据所接收到的消息的性质来决定需要采取的行动，以响应这个消息。响应这些消息的是一系列的方法，方法实际上就是类中的成员函数。

C++中允许友元破坏封装性

类中的私有成员一般是不允许该类外面的任何函数访问的，但是友元便可打破这条禁令，它可以访问该类的私有成员(包含数据成员和成员函数)。友元可以是在类外定义的函数，也可以是在类外定义的整个类，前者称友元函数，后者称为友元类。

通过 C++的保留字 friend 可以声明友元。下面给出一个友元函数的示例：

```
class   num1
{int data1;
  public:
    fun1() {data1=10;}
    friend int fun_friend(fun1,fun2);
 }

    class   num2
    { int data2;
      public:
      friend int fun_friend(fun1,fun2);
    }

    int fun_friend(num1,num2)
    {   return(num1.data+num2.data2);
```

```
}
```

上面的例子中，num1，num2 这两个类中的私有（private）成员变量，可以通过友元函数 fun_friend 进行访问。

3. C++中的构造函数与析构函数

C++的所有类都有一个或多个被称为“构造函数”的特定成员函数，它们被用来对对象进行初始化。如果类的定义中没有指定构造函数，那么编译器将产生一个不带任何参数的缺省构造函数。当编译器调用该缺省构造函数时，编译器就会对应该类的对象分配空间，但不会对其内部类型的值进行初始化。如果要对对象内部类型进行明确的初始化，你可以添加新的构造函数。

构造函数的名字通常与类的名字相同，而且构造函数不返回任何值。

相对于构造函数，自然有析构函数，这是在对象行将毁灭但尚未毁灭的前一刻，最后执行的函数。执行析构函数可以释放这个对象所占据的内存空间。一个类只能有一个析构函数。如果用户省略了析构函数，编译器会自动为该类生成一个析构函数。它的函数名称必定要与类名称相同，再在前面加一个“~”符号。

```
class XY{
    public x,y;
    XY();       //default constructor
    XY(double xarg,double yarg);
    ~XY();      //destructor
}
XY::XY()
{ printf("XY default constructor called\n");
  x=y=0;
}
XY::XY(double xarg, double yarg)
{ printf("XY explicit constructor called\n");
  x=xarg;
  y=yarg;
}
XY::~XY()
{printf("XY destuctor called\n");
}
```

接下来你可以设计函数来使用 XY 类的对象，并观察构造函数和析构函数的执行时机，这里就不再详细叙述了。

4. 四种不同的对象生存方式

在 C++中，有四种方法可以产生一个对象，第一种方法是在堆栈（Stack）中产生对象：

```
void MyFunc()
{  CFoo foo;  //在堆栈中产生 foo 对象
   .....
 }
```

第二种方法是在堆（heap）中产生对象:

```
void MyFunc()
{ .....
  CFoo  *pFoo=new CFoo(); //在堆中产生对象
 }
```

第三种方法是产生一个全局对象，同时也必然是一个静态对象（在下一小节会讲到）:

```
CFoo foo;  //在任何函数范围之外做此操作
```

第四种方法是产生一个局部静态对象：

```
void  MyFunc()
{  static CFoo foo;  //在函数范围内产生一个静态对象
 }
```

前两种方法，都是在 MyFunc()函数内，通过调用构造函数来构造对象，所不同的是一个把对象创建在堆栈上，一个把对象创建在堆上。创建在堆栈上的对象，会在程序流程超出对象的生存范围时，自动执行析构函数。而创建在堆上的对象，必须使用 delete 才能析构该对象。

对于第三种方法，对象是在程序还没有到达程序进入点（main 或 winmain）之前，由 startup 码调用构造函数创建的。它在整个程序退出时，才会被析构。

第四种局部静态对象，只有一个实例（instance）产生。它的构造函数在控制权第一次转到其声明处（也就是在 MyFunc 第一次被调用时）时被调用。

5. C++中的 static 成员

静态成员的提出是为了解决数据共享的问题。实现共享有许多方法，如：设置全局性的变量或对象是一种方法。但是，过多的使用全局变量或对象将妨害面向对象的风格。

静态成员是类的所有对象中共享的成员，而不是某个对象的成员，所以在类中，使用静态数据成员可以实现多个对象之间的数据共享，并且不会破坏封装的原则。使用静态数据成员还可以节省内存，因为它是所有对象所公有的，因此，对多个对象来说，静态数据成员只存储一处，供所有对象共用。静态数据成员的值对每个对象都是一样，但它的值是可以更新的。只要对静态数据成员的值更新一次，保证所有对象存取更新后的相同的值，这样可以提高时间效率。

静态数据成员的使用方法和注意事项如下：

（1）静态数据成员在定义或声明时前面加关键字 static。

（2）静态成员初始化与一般数据成员初始化不同。静态数据成员初始化的格式如下：

<数据类型><类名>::<静态数据成员名>=<值>

初始化种需要注意以下三个方面：①初始化在类体外进行，而前面不加 static，以免与一般静态变量或对象相混淆；②初始化时不加该成员的访问权限控制符 private，public 等；③初始化时使用作用域运算符来标明它所属类，因为，静态数据成员是类的成员，而不是对象的成员。

（3）静态数据成员是静态存储的，它是静态生存期，必须对它进行初始化。

（4）引用静态数据成员时，采用如下格式：<类名>::<静态成员名>

下面给出一个静态数据成员应用的示例：

```
class Myclass
{ public:
    Myclass(int a, int b, int c);
    void GetSum();
 private:
     int A, B, C;
 static int Sum;
};
int Myclass::Sum = 0; //静态数据成员的初始化

Myclass::Myclass(int a, int b, int c)
{ A = a;
  B = b;
  C = c;
  Sum += A+B+C;
}

void Myclass::GetSum()
{   cout<<"Sum="<<Sum;
}

void main()
{Myclass M(3, 7, 10), N(14, 9, 11);
 M.GetSum();
 N.GetSum();
}
```

从输出结果可以看到 Sum 的值对 M 对象和对 N 对象都是相等的。这是因为在初始化 M 对象时，将 M 对象的三个 int 型数据成员的值求和后赋给了 Sum，于是 Sum 保存了该值。在初始化 N 对象时，对将 N 对象的三个 int 型数据成员的值求和后又加到 Sum 已有的值上，于是 Sum 将保存另后的值。所以，不论是通过对象 M 还是通过对象 N 来引用的值都是一样的，

即为 54。

静态成员函数和静态数据成员一样，它们都属于类的静态成员，它们都不是对象成员。因此，对静态成员的引用使用类名，而不是对象名。调用静态成员函数使用如下格式：

<类名>::<静态成员函数名>(<参数表>);

在静态成员函数的实现中不能直接引用类中声明的非静态成员，可以引用类中声明的静态成员。如果静态成员函数中要引用非静态成员时，可通过对象来引用。下面通过例子来说明这一点。

```
class M
{public:
      M(int a) { A=a; B+=a;}
 static void f1(M m);
 private:
       int A;
       static int B;
};

void M::f1(M m)
{   cout<<"A="<<m.A<<"B="<<m.B; //通过对象来引用
}

int M::B=0;
void main()
{   M P(5),Q(10);
    M::f1(P);      //调用时不用对象名
    M::f1(Q);
}
```

读者可以自行分析其结果。

6. this 指针

任何一个对象的成员函数都可以存取一个特殊的指针 this，它指向对象本身。实际上，这个 this 指针隐藏在成员函数的参数表中。

假设有这样一个 CShape 类，该类中有一个成员函数 setColor。

```
class CShape
{   ……
  private:
      int m_color;
  public:
     void setColor(int color) {m_color=color;}
```

```
}
```

在被编译器编译过后，实际上变成了：

```
class CShape
    {  ……
       private:
            int m_color;
       public:
           void setColor(int color, (CShape*)this)} {this->m_color=color;}
     }
```

正是通过这个 this 指针，同一个成员函数可以处理不同的对象。

二、基类与派生类

C++中可以允许单继承和多继承。一个类可以根据需要生成派生类。派生类继承了基类成员，包括成员变量和成员函数，另外派生类还可以定义自己所需的不包含在父类中的新成员。一个子类的每个对象包含有从父类那里继承来的数据成员以及自己所特有的数据成员。

下面仅对 C++单继承的有关问题作一简要介绍：

在单继承中，每个类可以有多个派生类，但是每个派生类只能有一个基类，从而形成树形结构。

派生类对于基类的缺省继承方式是 private。其它不同派生方式得到的派生类对基类成员的访问权限列于表 1-1。无论使用的是哪种继承方式，在基类中以 private 关键字进行限定的成员在派生类中都是不可以访问的，这和以 protected 关键字定义的成员有着很大的区别，因此，如果希望成员能够为派生类所访问，而同时又不希望被类外部的其它函数直接访问，那么应该使用的访问限定符是 protected，而不是 private。

表 1-1　不同派生方式得到的派生类对基类成员的访问权限

基类成员所使用的关键字	在派生类中基类的继承方式	派生类对基类成员访问权限
public(公有成员)	Public	相当于使用了 public 关键字
	Protected	相当于使用了 protected 关键字
	Private	相当于使用了 private 关键字
protected(受保护成员)	Public	相当于使用了 protected 关键字
	Protected	相当于使用了 protected 关键字
	Private	相当于使用了 private 关键字
private(私有成员)	Public	不可访问
	Protected	不可访问
	Private	不可访问

派生类的构造函数

我们已经知道，派生类的对象的数据结构是由基类中说明的数据成员和派生类中说明的数据成员共同构成。将派生类的对象中由基类中说明的数据成员和操作所构成的封装体称为基类子对象，它由基类中的构造函数进行初始化。

构造函数不能够被继承，因此，派生类的构造函数必须通过调用基类的构造函数来初始化基类子对象。所以，在定义派生类的构造函数时除了对自己的数据成员进行初始化外，还必须负责调用基类构造函数使基类数据成员得以初始化。如果派生类中还有子对象时，还应包含对子对象初始化的构造函数。

派生类构造函数的一般格式如下：

<派生类名>(<派生类构造函数总参数表>):<基类构造函数>(参数表 1),<子对象名>(<参数表 2>)

{

<派生类中数据成员初始化>

};

派生类构造函数的调用顺序如下：

- 基类的构造函数
- 子对象类的构造函数(如果有的话)
- 派生类构造函数

下面给出一个派生类构造函数的例子：

```
#include
class A
{
public:
    A() { a=0; cout<<"类 A 的缺省构造函数.\n"; }
    A(int i) { a=i; cout<<"类 A 的构造函数.\n"; }
    ~A() { cout<<"类 A 的析构函数.\n"; }
    void Print() const { cout<< "类 A 的 Print 函数.\n";}
    int Geta() { reutrn a; }
private:
     int a;
}

class B : public A
{
public:
    B() { b=0; cout<<"类 B 的缺省构造函数.\n"; }
    B(int i, int j, int k);
```

```
    ~B() { cout<<"类 B 的析构函数.\n"; }
    void Print();
private:
     int b;
     A aa;
}

B::B(int i, int j, int k):A(i), aa(j)
{
    b=k;
    cout<<"类 B 的构造函数.\n";
}
void B::Print()
{
     A::Print();
     cout<< "类 B 的 Print 函数.\n";
 }
void main()
{
 B bb[2];
 bb[0] = B(1, 2, 5);
 bb[1] = B(3, 4, 7);
 for(int i=0; i<2; i++)
    bb[i].Print();
}
```

派生类构造函数使用中应注意的问题:

①派生类构造函数的定义中可以省略对基类构造函数的调用，其条件是在基类中必须有缺省的构造函数或者根本没有定义构造函数。当然，基类中没有定义构造函数，派生类根本不必负责调用基类的析构函数。

②当基类的构造函数使用一个或多个参数时，则派生类必须定义构造函数，提供将参数传递给基类构造函数途径。在有的情况下，派生类构造函数的函数体可能为空，仅起到参数传递作用。

派生类中的析构函数

当对象被删除时，派生类的析构函数被执行。由于析构函数也不能被继承，因此在执行派生类的析构函数时，基类的析构函数也将被调用。执行顺序是先执行派生类的构造函数，再执行基类的析构函数，其顺序与执行构造函数时的顺序正好相反。