版权说明

本图书的版权属于北京智帆高科科技有限公司，作者保留该图书的所有权利。本图书在未来会作为纸版图书出版，一切对该图书的非法复制和其他商业行为都会追究法律责任！

以电子版发布此书的目的是提高国内 CAD 开发者的普遍水平，让入门者更快地能够编写代码，让初级程序员更快达到中高级程序员的水平。为国内 CAD 开发整体水平的提高，我们愿意尽一份微薄之力。

我们会不定期发布该图书的新章节，也会发布.NET ObjectARX 方面的电子图书，请留意我们在网站上的新公告。

我们非常愿意在 ObjectARX 编程站论坛（http://www.objectarx.net/bbs）与大家讨论书中的问题和该图书进一步写作的建议，也很欢迎大家与我们讨论其他的问题。

大家可以通过网站论坛与我们取得联系，也可以发送电子邮件到 zf9568@263.net 与我们沟通，祝大家读书愉快！

张帆

2007-7-15

定制开发项目合作

北京智帆高科科技有限公司是一家专门从事 CAD 二次开发的软件公司，公司成立两年以来，我们完成了一系列大规模的定制开发项目：

（1）机场场道、排水辅助设计系统；

（2）机场灯光辅助设计系统；

（3）公路桥梁养护巡检管理信息系统；

（4）工业厂房电力辅助设计系统；

（5）铁路车站信号系统配线设计；

（6）高速公路隧道设备布设系统；

（7）混凝土道面厚度计算系统；

（8）铁路运输列流图绘制系统。

所有项目均得到客户的满意评价。我们的优势在于：

（1）优秀的业务建模，将客户的业务模型成功地在计算机中表示出来；

（2）算法实现能力强，几乎所有的功能都进行优化处理，软件运行速度快；

（3）商品化的软件包装能力，软件的加密、安装包等完全具备。

欢迎每一位客户与我们合作开发适用您公司的软件，我们愿意与您共享下一个成功的案例。

联系方式：QQ 7413643，邮箱 zf9568@263.net。

更新记录

2009-8-26 更新：

- 8.3 节，可停靠窗体；
- 8.4 节，使用 MFC 创建工具栏。

第1章 ObjectARX 编程基础

初学 ObjectARX 编程，需要了解其开发和调试环境的构建，本章首先帮助读者建立起开发环境，进而介绍加载和运行 ObjectARX 应用程序的方法，最后仍然以 Hello,World 来作为进入 ObjectARX 开发的第一个程序。

1.1 ObjectARX 开发环境

1.1.1 说明

使用 ObjectARX，首先要确定你的目标平台，获得适当的开发环境。如果你是在 AutoCAD 2002 平台上开发，你就需要具备下面的工具和软件：

- AutoCAD 2002 中文版或英文版。
- VC++ 6.0 英文版。
- ObjectARX 2002 开发包。

一般来说，ObjectARX 开发包的版本和 AutoCAD 的版本是对应的。在开发工具方面，AutoCAD R14、AutoCAD 2000 和 AutoCAD 2002 平台上使用的开发工具是 VC++ 6.0，AutoCAD 2004 和 AutoCAD 2005 平台上使用的开发工具是 VC.NET 2002。

本节将要介绍 ObjectARX 开发环境的构建，包括开发包的获得、开发包的组成部分、ObjectARX 向导的安装，以及自动提示和显示 ObjectARX 关键字的方法。

1.1.2 思路

本章的内容不涉及具体的编程，此部分从略。

1.1.3 步骤

在开始本书的旅行之前，首先要指出，本书所有的程序都是基于 AutoCAD 2002 中文版，因此构建的开发环境为“VC++ 6.0 英文版＋AutoCAD 2002 中文版＋ObjectARX 2002”。按照下面的步骤，一步一步构建开发环境：

（1）安装 AutoCAD 2002 中文版和 VC++ 6.0 英文版。软件的安装可以参考相关软件的说明文件，不再介绍。

提示：微软公司在中国从未推出正式的 VC++ 6.0 中文版，如果你使用了某些外挂

的平台来实现 VC++的中文汉化，那么可能无法用本书所说的方法实现 ObjectARX 关键字的自动提示和高亮显示。

（2）获得ObjectARX开发包。可以到Autodesk公司的官方网站下载开发包，下载的页面是http://www.autodesk.com/adn。解压下载得到的压缩文件，能够得到下面几个文件夹：

- arxlabs：包含了 ObjectARX 的教程，和对应的示例文件。
- classmap：包含一个 DWG 图形，其中显示了 ObjectARX 类层次的结构。
- docs：包含所有的联机帮助文件。
- docsamps：包含在《ObjectARX 开发者向导》（在 docs 文件夹中，为英文的资料）中所提到的源代码和说明文件。
- inc：包含 ObjectARX 的头文件。
- lib：包含 ObjectARX 的库文件。
- redistrib：包含一些动态链接库（DLL），其中一些可能是运行 ObjectARX 应用程序所必需的。
- samples：包含了许多 ObjectARX 应用程序的例子。
- utils：包含扩展 ObjectARX 的应用程序，例如用于边界表示的 brep 程序。

提示：为便于访问，一般可以将 ObjectARX 2002 放置在驱动器的根目录下，例如 E:\ObjectARX 2002\等。

（3）安装ObjectARX开发向导。按照开发包中的路径\utils\ObjARXWiz\，找到一个名称为wizards.exe的自解压文件，将其解压到一个文件夹中，运行其中的WizardSetup.exe文件，系统弹出如图 1.1所示的对话框，单击【Install】按钮开始安装向导。

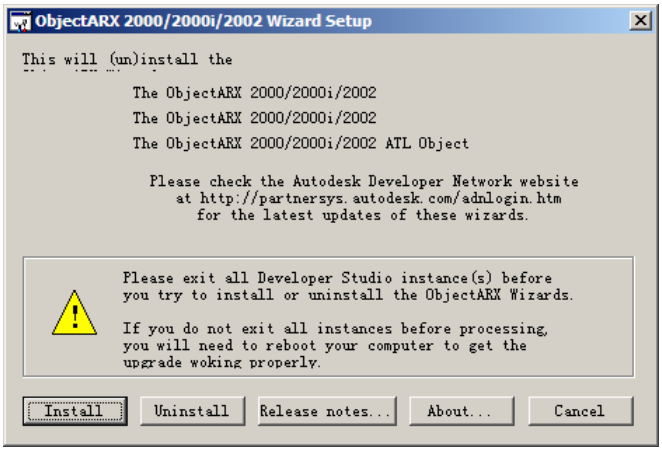

图1.1　安装 ObjectARX 开发向导

经过一段时间，系统弹出如图 1.2所示的对话框，单击【确定】按钮完成ObjectARX向导的安装。

图1.2 向导安装完成

此时启动VC++ 6.0，选择【File/New】菜单项，系统会弹出【New】对话框，其中的项目列表中已经包含了ObjectARX 2000/2000i/2002 AppWizard，如图 1.3所示。

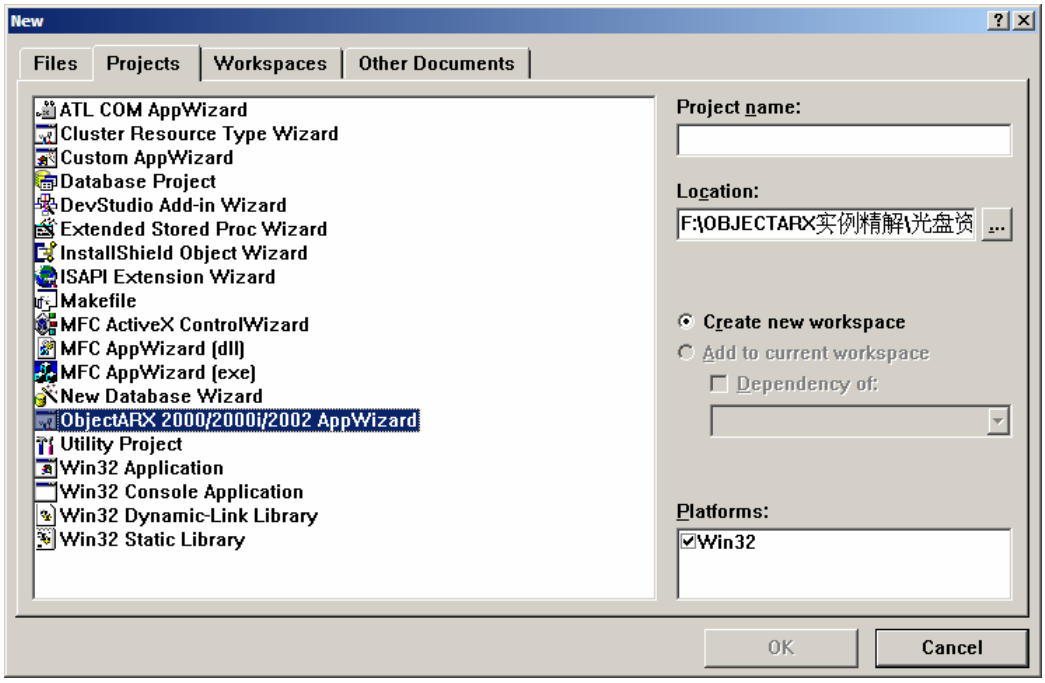

图1.3 ObjectARX 开发向导被添加到列表中

（4）配置ObjectARX的帮助信息。安装ObjectARX开发向导之后，除了【New】对话框的项目列表增加了对ObjectARX项目的支持，还增加了一个专门的嵌入工具栏，如图 1.4所示。

图1.4 ObjectARX 的嵌入工具栏

单击ObjectARX嵌入工具栏的“ObjectARX AddIn Configuration”按钮，系统会弹出如图1.5所示的对话框。如果你愿意，可以取消选择【AddIn configuration】选项组的【Enable Live Update for the ObjectARX 2000(i) Wizard】复选框，避免每次启动VC++时ObjectARX向导的自动更新。

在【Help configuration】选项组的第一个文本框中输入 ObjectARX 帮助文档的位置，也可以单击文本框右侧的按钮从计算机中查找该文件。最好选择 arxdoc.chm，这个文件包含了

其他的几个文件。

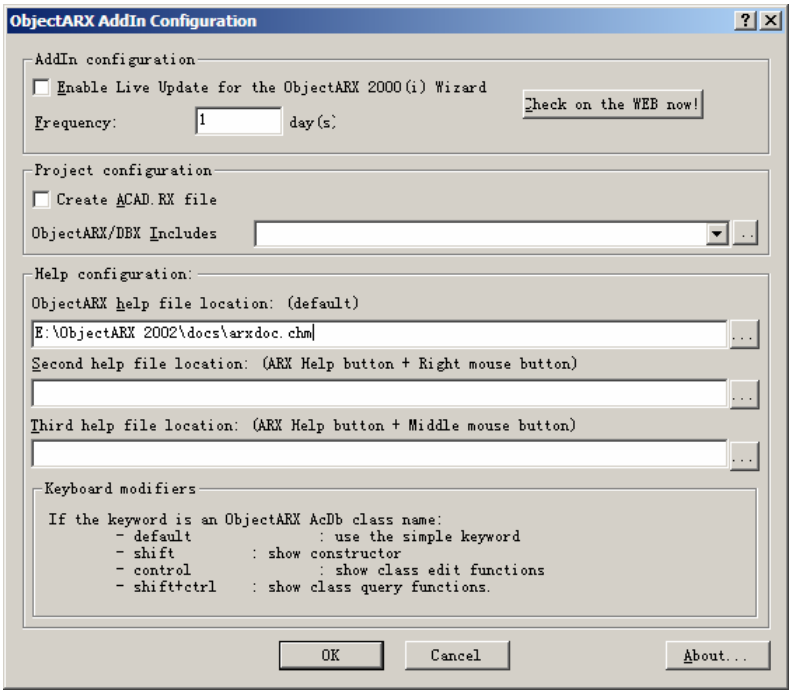

图1.5　添加帮助文件的位置

在VC++ 6.0 中，选择【Tools/Customize】菜单项，系统会弹出如图 1.6所示的对话框。切换到【Keyboard】选项卡中，从【Category】列表中选择【Add-ins】选项，从【Commands】列表中选择【ObjectARXAddInArdHelp】选项，也就是对应了ObjectARX嵌入工具栏的帮助按钮。在【Press new shortcut】文本框中单击左键，然后按下快捷键Alt+F1（为了避免和VC++本身的F1 快捷键冲突，你可以自己选择适当的快捷键），单击【Assign】按钮，然后单击【Close】按钮完成设置。

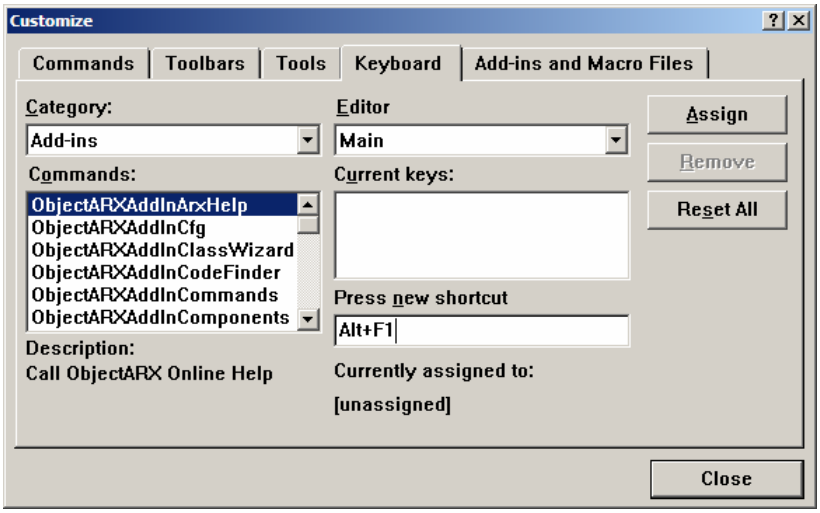

图1.6 为帮助文件指定快捷键

这时，在 VC++中编写 ObjectARX 代码时，就可以按下快捷键 Alt+F1，获得 ObjectARX 的相关帮助。

（5）高亮显示、自动提示 ObjectARX 关键字。答案很简单，安装工具软件 Visual Assist 6.0（也称 VC 助手）。安装该软件之后，不禁能够完成显示和自动提示 ObjectARX 关键字的功能，而且能够设置编辑器的环境，自动提示自定义类的成员变量和函数，非常方便。

如果没有 VC 助手的帮助，那么起码还要在枯燥的编程工作中摸索更长时间。

1.1.4 效果

由于本节内容的特殊性，这里的实例效果没有意义，故略去。

1.1.5 小结

虽然本书介绍的是 ObjectARX 2002，但是 ObjectARX 的开发包除若干全局函数的名称和类的数量有所变化之外，总体构架没有太大改变，因此本书的绝大部分实例稍加更改后即可运行于 AutoCAD 2000 以上的各个版本。

如果你使用的是 VC++.NET 2002，配置开发环境的方法与之类似。

1.2 ObjectARX 应用程序的加载和运行

1.2.1 说明

本节介绍在开发 ObjectARX 程序阶段加载和运行应用程序的方法，这是调试程序的基础，也是程序员所使用的方法。在创建打包程序的部分，还要介绍一些自动加载 ObjectARX 应用程序的方法，那些则是用户使用的方法。

1.2.2 思路

加载 ObjectARX 应用程序可以通过多种方法：

- 使用 APPLOAD 命令。
- 使用 ARX 命令。
- 直接拖放 ARX 文件。

执行 ObjectARX 应用程序仅可在命令行输入程序中注册的命令。

1.2.3 步骤

下面的步骤演示加载和运行 ObjectARX 应用程序的方法：

（1）使用APPLOAD命令加载程序。在AutoCAD 2004 中，选择【工具 / 加载应用程序】菜单项，系统会弹出如图 1.7所示的【加载 / 卸载应用程序】对话框。

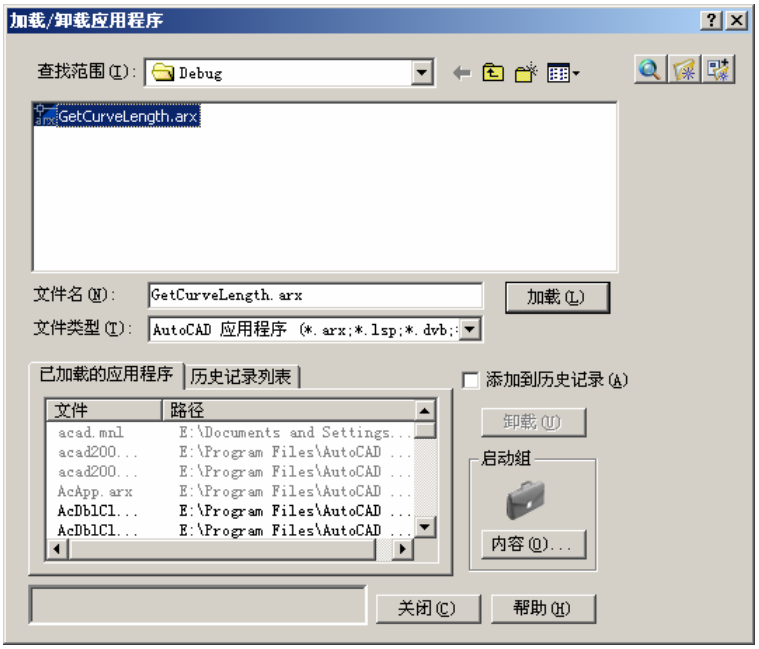

图1.7　加载应用程序

从文件列表中选择所要加载的程序，单击【加载】按钮，就可以将选择的程序加载到AutoCAD 2004 中。需要注意的是，所有二次开发的程序必须加载后才能使用。

如果要经常使用某个程序，就可以让AutoCAD 2004 在启动时自动加载该程序。在【加载 / 卸载应用程序】对话框中单击【内容】按钮，系统会弹出如图 1.8所示的【启动组】对话框。单击【添加】按钮，系统会弹出【将文件添加到启动组中】对话框，在此对话框中选择所要添加到启动组中的程序。回到【启动组】对话框后，单击【关闭】按钮，返回【加载 / 卸载应用程序】对话框，此时启动组中的程序已经被加载。

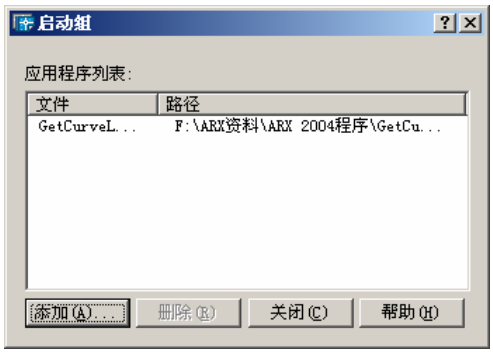

图1.8　将程序添加到启动组

提示：将程序添加到启动组，使其在 AutoCAD 启动时自动加载，对于 ObjectARX 程序的调试很有帮助。

（2）使用 ARX 命令加载程序。在命令行执行 ARX 命令，按照命令提示进行操作：

命令: arx

输入选项 [?/加载(L)/卸载(U)/命令(C)/选项(O)]: l 〖输入L并按下Enter键，系统弹出如图1.9所示的对话框〗

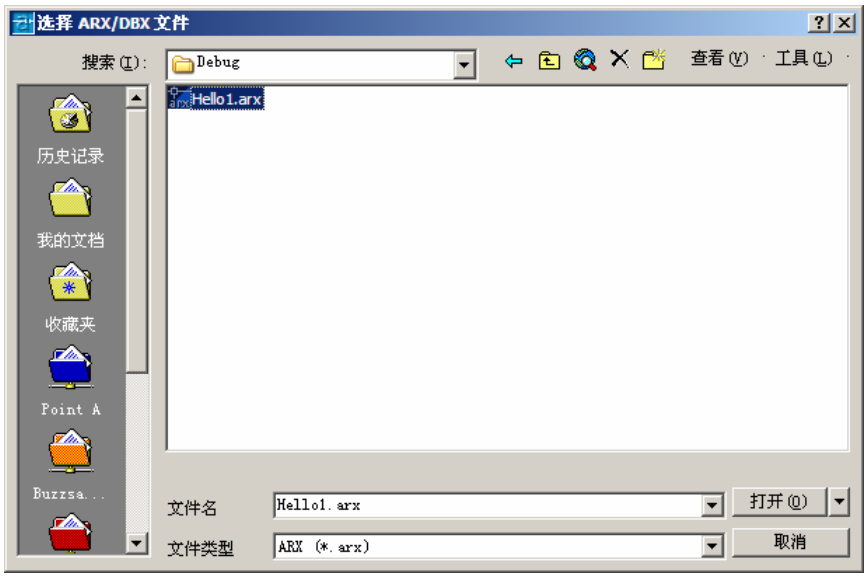

图1.9 选择所要加载的文件

ARX 命令还能够完成多种功能：

- 查看当前已经加载的 ARX 文件。
- 查看系统中已经定义的外部命令。
- 卸载应用程序。

（3）直接将ARX文件拖放到AutoCAD窗口中。如果指定的ARX成功加载，会在命令行得到如图 1.10所示的结果。

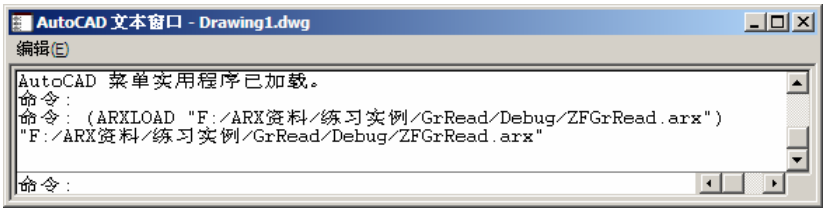

图1.10 成功加载 ARX 文件的提示

（4）要运行 ObjectARX 程序中注册的命令，可以直接在命令行键入命令名称并按下 Enter 键，前提是指定的程序已经加载到 AutoCAD 中。

使用ARX命令可以查看系统中已经注册的外部命令，假设已经加载了本章第 3 节创建的程序（参见 1.3 节程序的运行结果），那么可以使用ARX命令查看已经注册的命令，运行过程

和结果如图 1.11所示。

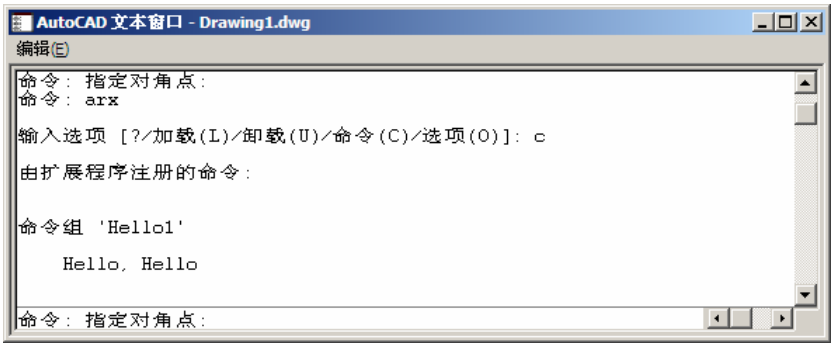

图1.11　查看已经注册的命令

要运行该程序，便可以直接在命令行键入Hello并按下Enter键，得到如图 1.12所示的结果。

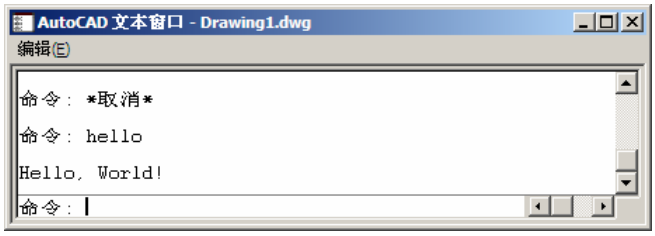

图1.12　执行 ARX 程序的结果

1.2.4　效果

参见上一节的内容。

1.2.5　小结

学习本节内容之后，读者应该掌握使用 APPLOAD 命令加载 ObjectARX 应用程序的方法，以及运行 ARX 程序的方法。

1.3　手工创建 Hello，World 程序

1.3.1　说明

几乎所有程序设计的书籍都从“Hello,World”程序开始，虽然它很简单，但是用来描述程序的基本结构却非常合适。本节将介绍一个完全手工创建的“Hello,World”程序，剖析 ObjectARX 程序的组成结构。

程序所要展示的效果非常简单：当用户加载该程序并在命令行执行相应的命令时，AutoCAD 会在命令行输出“Hello,World”。

1.3.2 思路

虽然 ObjectARX 程序输出文件的后缀是.arx，但是它实际上是一种动态链接库（DLL）。与常规的 Windows 动态链接库相同，ObjectARX 程序是在 AutoCAD 调用它时加载的，与 AutoCAD 应用程序自身的代码相互独立。

ObjectARX 是专门用于 AutoCAD 应用程序的动态链接库，因此创建 ARX 程序时需要进行一些设置工作：

- 设置编译器的参数：包括代码的生成方式、ObjectARX 头文件的路径。
- 设置链接器的参数：包括输出文件的名称、添加链接库文件、指定 ObjectARX 库文件的路径。

1.3.3 步骤

（1）在VC++ 6.0 集成开发环境中，选择【File/New】菜单项，系统会弹出【New】对话框。在【Projects】选项卡的工程类型列表中选择【Win32 Dynamic-Link Library】，在【Project name】文本框中输入工程名称Hello1，在【Location】文本框中输入适当的工程位置，如图1.13所示。完成设置后，单击【OK】按钮。

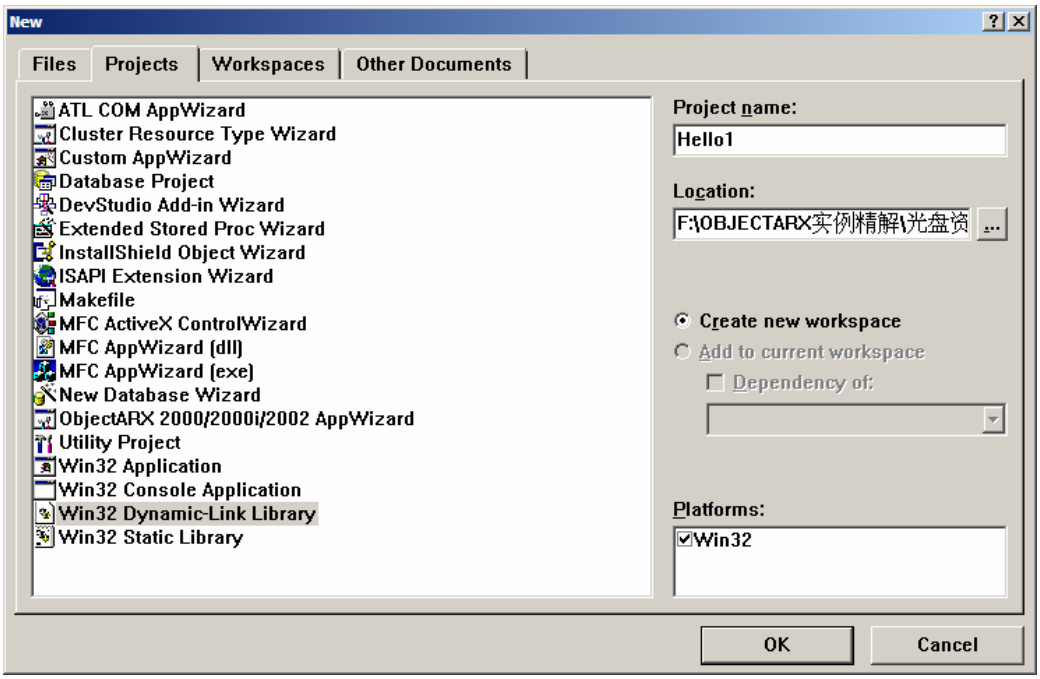

图1.13 新工程名称和位置的设置

（2）在【Win32 Dynamic-Link Library – Step 1 of 1】对话框中，选择An empty DLL project

选项，如图 1.14所示。设置完毕之后，单击【Finish】按钮。

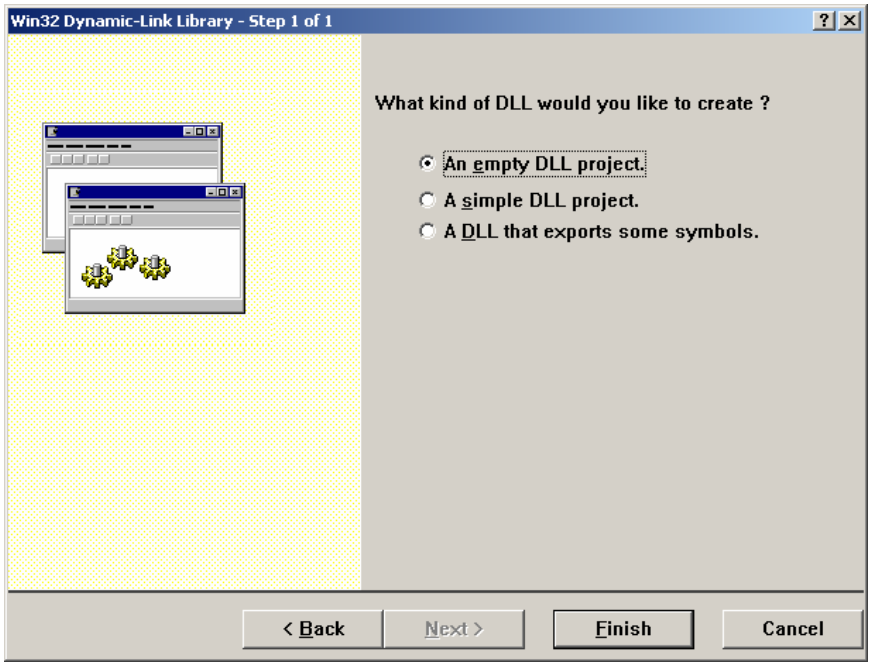

图1.14　创建空的动态链接库

（3）在【New Project Information】对话框中，显示了向导所创建的动态链接库的基本信息，如图 1.15所示。在对话框中单击【OK】按钮。

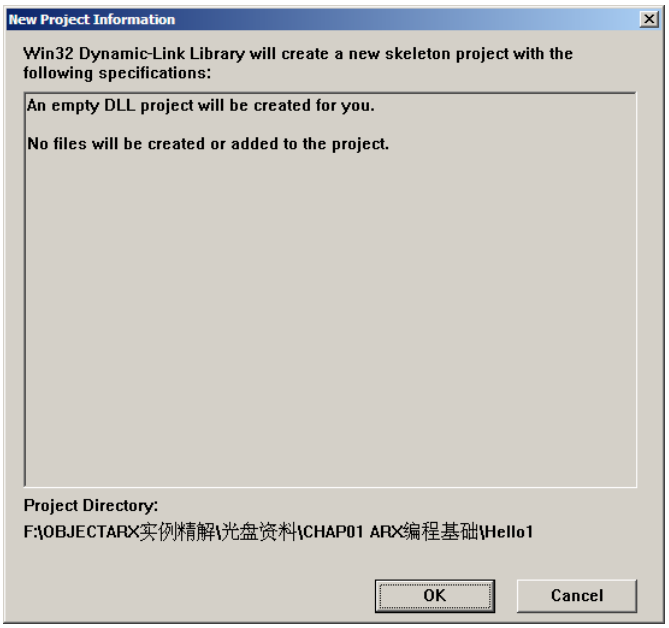

图1.15　显示向导生成的 DLL 工程信息

（4）选择【Project/Setting】菜单项，系统会弹出【Project Settings】对话框。在【General】

选项卡中，从对话框左侧的【Settings for:】组合框中选择【All configurations】选项，表明设置的参数对调试版本和发行版本同样有效。

注意：如不做特殊说明，本书所有程序的调试版本和发行版本编译和连接选项均为相同设置。

（5）设置编译器参数。进入【Project Settings】对话框的【C/C++】选项卡，从【Category:】组合框中选择【Code Generation】选项，从【Use run-time library】组合框中选择【Multithreaded DLL】选项。从【Category:】组合框中选择【Preprocessor】选项，在【Additional include directories:】文本框中输入ObjectARX头文件的路径（笔者使用的路径为“E:\ObjectARX 2002\inc”，请根据自己的开发包的安装位置设置路径），如图 1.16所示。

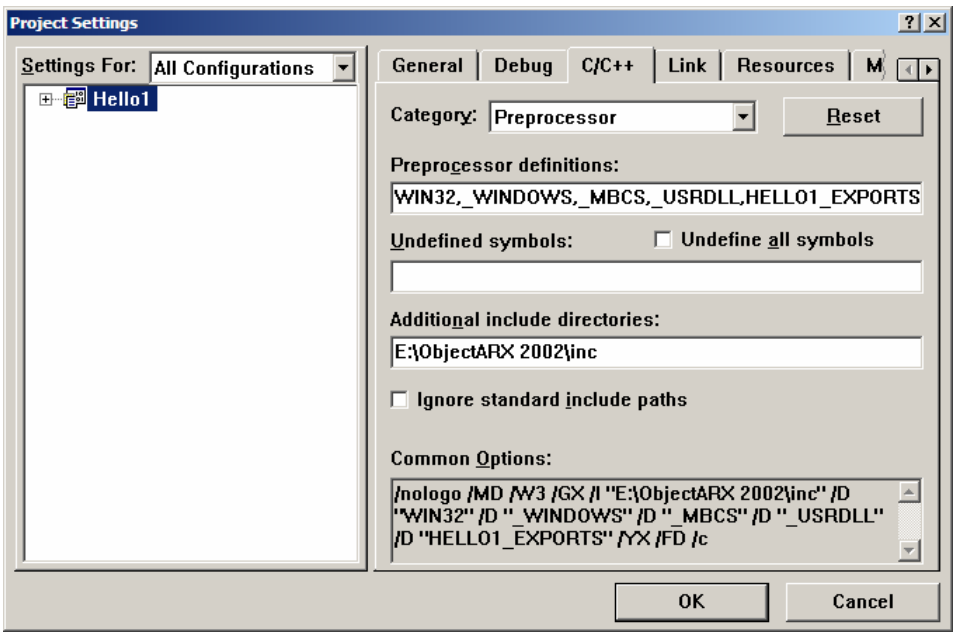

图1.16　设置编辑器参数

（6）设置输出文件。在【Project Settings】对话框中，进入【Link】选项卡。确保【Category:】组合框中选择【General】选项，从对话框左侧的【Settings for:】组合框中选择【Win32 debug】选项，在【Output File Name:】文本框中将文件的扩展名由dll修改为arx，如图 1.17所示。

注意：之所以将文件扩展名修改为 arx，是因为 AutoCAD 中默认的 ARX 程序扩展名是 arx。

从对话框左侧的【Settings for:】组合框中选择【Win32 release】选项，在【Output File Name:】文本框中同样将文件的扩展名由 dll 修改为 arx。之所以分成两步来修改调试版本和发行版本的文件扩展名，是因为两个文件不是放置在同一个路径下。后面你将会看到，扩展名为 arx 的文件是 ObjectARX 编程的目标文件，AutoCAD 调用该文件而执行 ARX 程序。

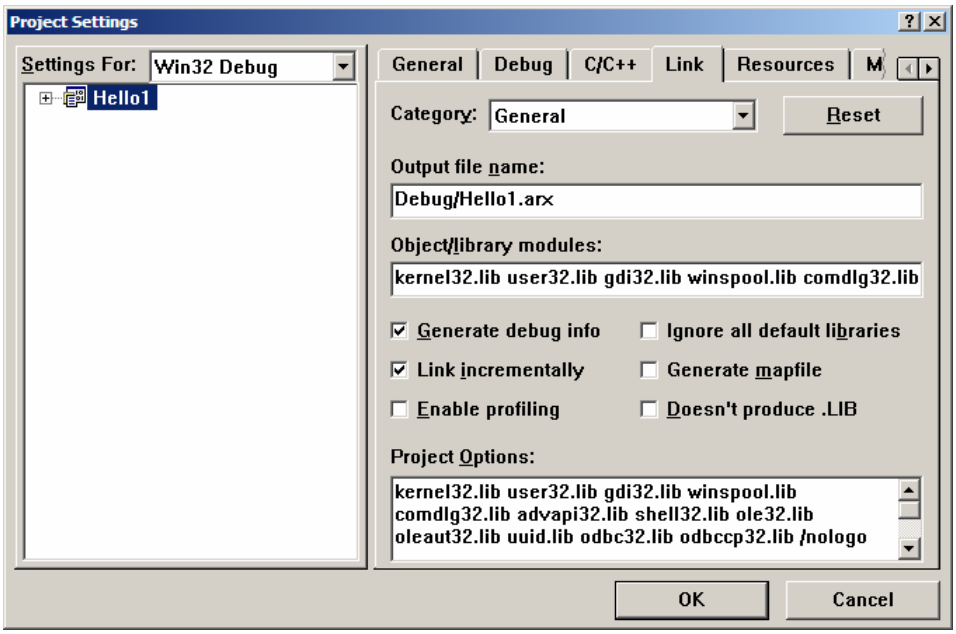

图1.17　修改输出文件的扩展名

（7）设置连接器连接的库文件。从对话框左侧的【Settings for:】组合框中选择【All configurations】选项，其它的参数设置对调试版本和发行版本是完全相同的。在【Object/library module】文本框中输入该程序所要使用的ARX库名称，这里输入了rxapi.lib、acrx15.lib、acutil15.lib和acedapi.lib四个文件，每个文件之间用空格隔开，如图 1.18所示。

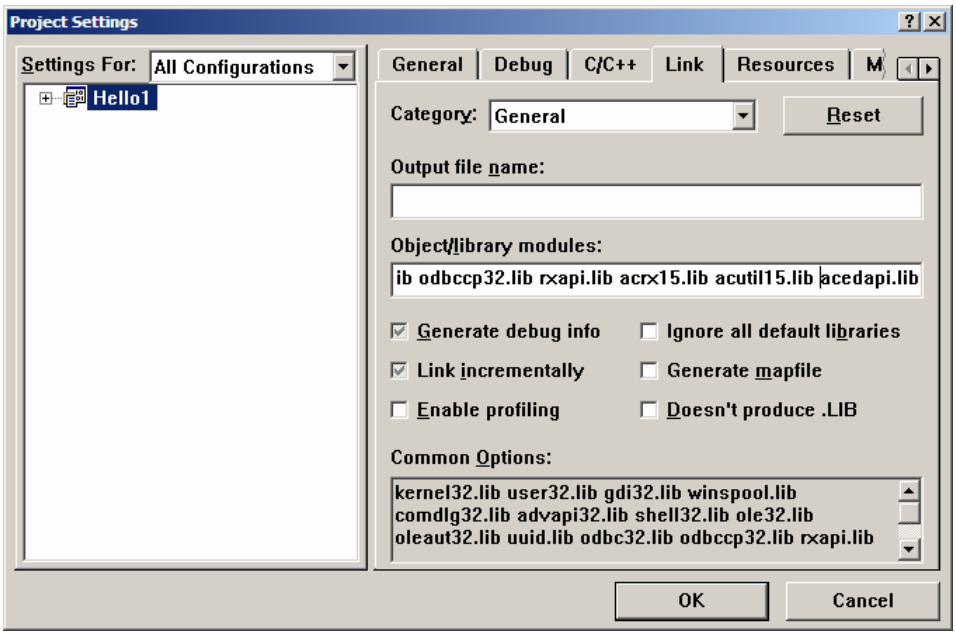

图1.18　添加 ARX 库文件

提示：在【Object/library module】文本框中本来就包含了一些 Windows 编程所需要的库文件，向其中添加 ARX 库文件时，可以保留或删除原有的库文件，在这里不会对程序产生什么影响。

（8）设置连接器的附加库文件路径。从【Category:】组合框中选择【Input】选项，在【Additional Library Path:】文本框中输入ObjectARX库文件的路径（同样需要根据你的ARX安装包路径来设置，笔者输入的是“E:\ObjectARX 2002\lib”），如图 1.19所示。

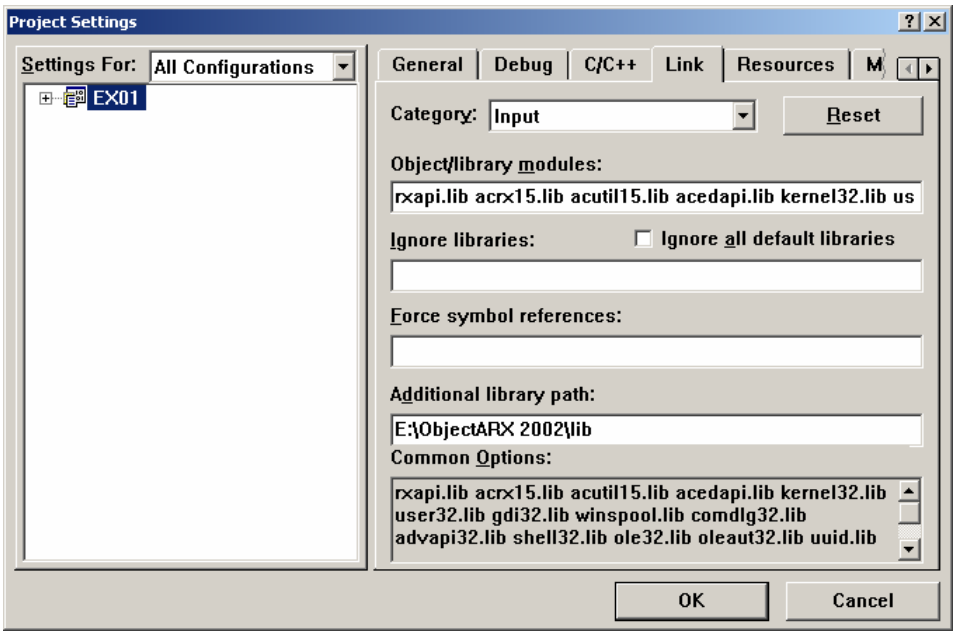

图1.19　设置 ObjectARX 库文件路径

（9）在【Project settings】对话框中，单击【OK】按钮，完成工程的参数设置。到这里，工程的设置工作已经完成，后面我们将要添加一些代码，实现在 AutoCAD 命令窗口中显示“Hello World!”的功能。

（10）选择【Project/ Add to Project/ New...】菜单项，系统会弹出【New】对话框。在【Files】选项卡的文件类型列表中，选择【C++ Source File】选项，在【File】文本框中输入Hello作为文件的名称，如图 1.20所示，单击【OK】按钮关闭该对话框。

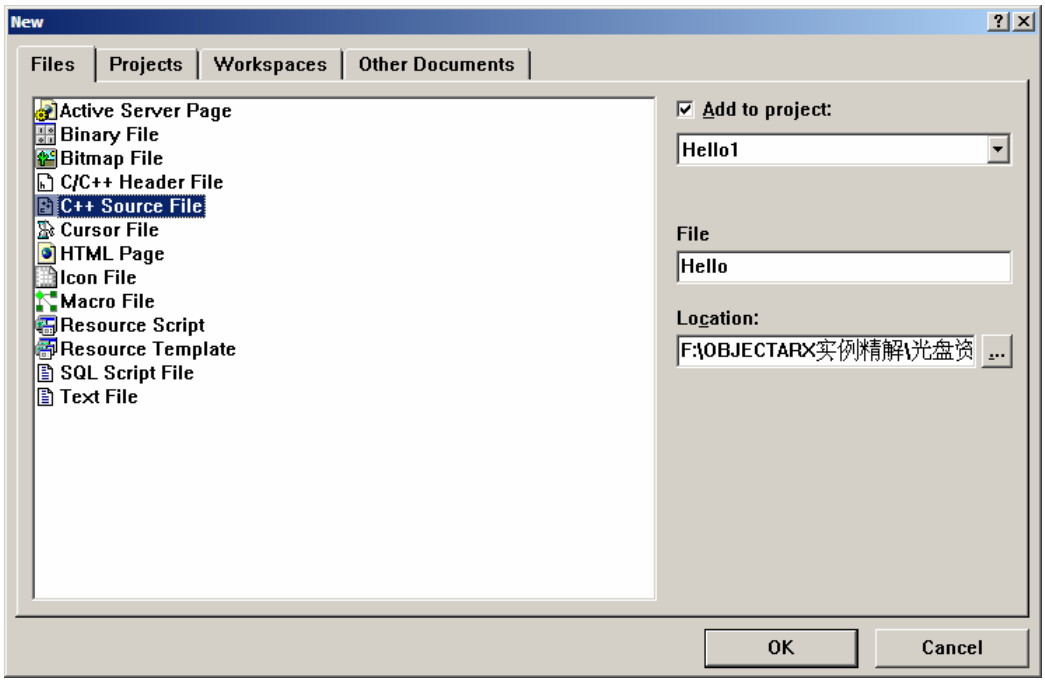

图1.20　添加 C++源文件

（11）在代码窗口中，首先要包含两个 ObjectARX 的头文件，具体代码为：

```
#include <aced.h>
#include <rxregsvc.h>
```

这两个头文件，分别是 ARX 应用程序定义和访问 AutoCAD 指定服务所需要的头文件（aced.h），以及使用 acrxXXX 工具函数所需的头文件（rxregsvc.h）。

（12）声明 initApp()和 unloadApp()函数。其中 initApp()函数在我们的应用程序被 AutoCAD 加载时调用，而 unloadApp()函数则在程序被卸载时所调用。这两个函数是在后面要介绍的 acrxEntryPoint 函数（相当于一般 C 程序中的 main 函数）中被调用，其格式为：

```
void initApp();
void unloadApp();
```

（13）添加自定义函数的声明。该函数能够在 AutoCAD 的命令窗口中显示“Hello，World!”语句，其格式为：

```
void HelloWorld();
```

（14）定义 initApp()函数。实际上这里它只做一件事情，就是使用 AutoCAD 的命令机制注册一个新命令。这个命令同 AutoCAD 的内部命令一样，可以直接在命令行中执行。实际上，这就是运行 ARX 程序的方法。initApp()函数的实现代码为：

```
void initApp()
{
        //使用AutoCAD命令机制注册一个新命令
        acedRegCmds->addCommand("Hello1",
            "Hello",    //输入这两个命令名称均可以在
```

```
                    "Hello",    //AutoCAD中运行该程序
                    ACRX_CMD_MODAL,
                    HelloWorld);
}
```

ObjectARX 应用程序使用一个名为 AcEdCommandStack 的类（命令堆栈）来添加和删除命令，而 acedRegCmds 宏提供了一个指向 AcEdCommandStack 类的指针。AcEdCommandStack 类的 addCommand 函数用来向 AutoCAD 注册一个外部命令，而 removeGroup 函数用来删除已经存在的一个外部命令组。

addCommand 函数的定义形式为：

```
virtual Acad::ErrorStatus addCommand(
                    const char* cmdGroupName,
                    const char* cmdGlobalName,
                    const char* cmdLocalName,
                    Adesk::Int32 commandFlags,
                    AcRxFunctionPtr FunctionAddr,
                    AcEdUIContext * UIContext = NULL,
                    int fcode = -1,
                    HINSTANCE hResourceHandle = NULL,
                    AcEdCommand** cmdPtrRet = NULL) = 0;
```

现在不需要深入研究这个函数，只需要关注前面的5个参数，分别用来指定命令组名称、命令的国际名称、命令的本国名称、命令的类型和指向实现函数的指针。

一个命令组可以包含多个不同的命令，同一个命令可以用国际名称和本国名称来执行，但是由于在通常使用的 AutoCAD 中文版本也使用英文来作为命令名称，因此命令的本国名称与国际名称保持一致。如果你愿意的话，用本国名称来实现命令的别名倒也是一个不错的主意。

何谓命令的别名？不熟悉 AutoCAD 使用的读者可能不太明白它的含义。AutoCAD 所有的功能都是通过命令的形式来实现，譬如要绘制一条直线，就会使用对应的 LINE 命令，可以在 AutoCAD 命令行输入 LINE 并且按下 Enter 键来执行该命令（即使你是单击了“绘图”工具栏的“直线”按钮，或者选择了【绘图 / 直线】菜单项，其本质也是调用 LINE 命令）。同时，也可以在命令行键入 L 并按下 Enter 键来执行该命令，这样输入 L 和 LINE 就达到了同样的效果，简化了用户的输入工作量。

addCommand 函数的第4个参数用来说明命令的类型：模态命令或者透明命令。模态命令在执行过程中无法运行其它的模态命令，但是可以运行透明命令。例如，在创建直线的过程中，可以执行 ZOOM 命令来缩放视图，但是却不能执行 CIRCLE 命令来创建一个圆。大部分的命令都被注册为模态命令，也就是用 ACRX_CMD_MODAL 来作为第4个参数。

第5个参数指定该命令调用时所要执行的函数，上面的代码中输入了 HelloWorld 函数响应该参数，那么当 Hello 命令被执行时，就会运行 HelloWorld 函数中的代码。

（15）定义 unloadApp()函数。该函数会从 AutoCAD 中删除我们所定义的命令组“EX01”，此后就不能再调用所定义的命令了。前面在 initApp()函数中注册了外部函数，在卸载程序时

自然要将其删除。unloadApp()函数的实现代码为：

```
void unloadApp()
{
        //删除命令组
        acedRegCmds->removeGroup("Hello1");
}
```

removeGroup 函数所要实现的功能很简单，就是删除指定的命令组，以及保存在其中的所有命令。因此，当应用程序被卸载之后，就不能执行该命令组中注册的命令了。

（16）定义 HelloWorld()函数。该函数的作用是在 AutoCAD 的命令行中输出语句“Hello, World!”，通过 ARX 的一个全局函数 acutPrintf()在 AutoCAD 命令行显示指定的字符串，其实现代码为：

```
void HelloWorld()
{
        acutPrintf("\nHello, World!");
}
```

acutPrintf()函数的使用类似于 C 语言中的 printf 函数，可以使用“\n”来实现打印中的换行。

（17）添加入口点函数。类似于 Windows 应用程序中的消息机制，入口点函数 acrxEntryPoint()用一个 switch 结构来处理各种消息。最基本的消息就是 AcRx::kInitAppMsg 和 AcRx::kUnloadAppMsg，前者在应用程序加载时发生，后者则是应用程序卸载时发生。入口点函数的实现代码为：

```
extern "C" AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt)
{
        switch (msg)
        {
        case AcRx::kInitAppMsg:
                acrxDynamicLinker->unlockApplication(pkt);
                acrxRegisterAppMDIAware(pkt);
                initApp();
                break;
        case AcRx::kUnloadAppMsg:
                unloadApp();
                break;
        default:
                break;
        }
        return AcRx::kRetOK;
}
```

这里还没必要详细去了解入口点函数各语句的含义，只要知道通过调用 initApp()和unloadApp()函数，实现了命令的注册和删除。记住，ObjectARX 应用程序是一个 DLL，因此它没有主函数 main()，AutoCAD 调用 ObjectARX 模块的 acrxEntryPoint() 函数来传送消息到应用程序中。

（18）添加模块定义文件。选择【File/New】菜单项，系统会弹出如图 1.21所示的【New】对话框，从文件类型列表中选择【Text File】选项，在【File】文本框中输入“Hello.def”，作为模块定义文件的名称。

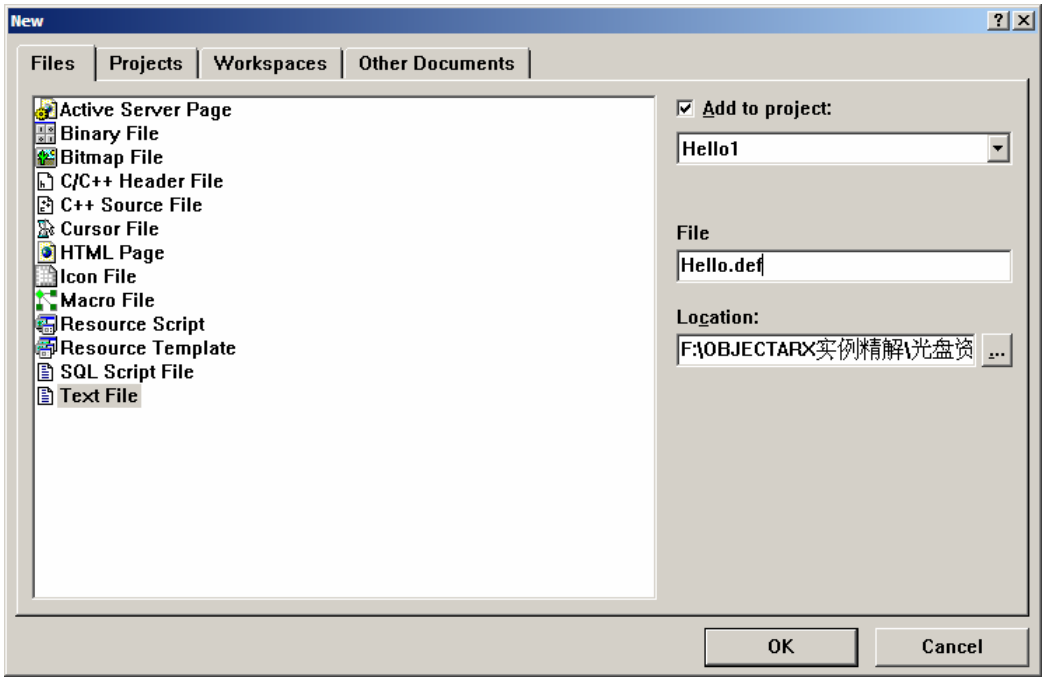

图1.21 添加模块定义文件

（19）在模块定义文件中添加下面的代码：

```
LIBRARY Hello1
DESCRIPTION "First ARX Application."

EXPORTS
acrxEntryPoint          PRIVATE
acrxGetApiVersion       PRIVATE
```

模块定义文件是 Windows 动态链接库创建输出函数的一种方式，也就是说，通过模块定义文件，AutoCAD 能够知道这个应用程序输出了哪些函数，以便执行入口点函数。

一个最小的模块定义文件也必须包含下面的部分：

- 第一句必须是 LIBRARY 语句，后面跟着项目的名称。
- EXPORTS 语句列出了动态链接库输出的函数，对于 ARX 应用程序，至少要输出 acrxEntryPoint 和 acrxGetApiVersion 两个函数。

- 虽然并非必不可少，但是一般来说最好使用 DESCRIPTION 语句来说明动态链接库的作用。

到此为止，已经完成了手工创建“Hello,World”应用程序的所有步骤。可能读者对各部分代码的放置会有疑问，因此给出完整的实现代码：

```
//////////////////////////////////////////////////////////////////////////
// Hello.cpp
// by 张帆
//////////////////////////////////////////////////////////////////////////
// 包含头文件
#include <aced.h>
#include <rxregsvc.h>

// 声明初始化函数和卸载函数
void initApp();
void unloadApp();

// 声明命令的执行函数
void HelloWorld();

// 加载应用程序时被调用的函数
void initApp()
{
        //使用AutoCAD命令机制注册一个新命令
        acedRegCmds->addCommand("Hello1",
            "Hello",    //输入这两个命令名称均可以在
            "Hello",    //AutoCAD中运行该程序
            ACRX_CMD_MODAL,
            HelloWorld);
}

// 卸载应用程序时被调用的函数
void unloadApp()
{
        //删除命令组
        acedRegCmds->removeGroup("Hello1");
}

// 实现Hello命令的函数
void HelloWorld()
```

```
{
        acutPrintf("\nHello, World!");
}

// 入口点函数
extern "C" AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt)
{
        switch (msg)
        {
        case AcRx::kInitAppMsg:
                acrxDynamicLinker->unlockApplication(pkt);
                acrxRegisterAppMDIAware(pkt);
                initApp();
                break;
        case AcRx::kUnloadAppMsg:
                unloadApp();
                break;
        default:
                break;
        }
        return AcRx::kRetOK;
}
//////////////////////////////////////////////////////////////////////////
// Hello.def
// by 张帆
//////////////////////////////////////////////////////////////////////////

LIBRARY Hello1
DESCRIPTION "First ARX Application."

EXPORTS
acrxEntryPoint          PRIVATE
acrxGetApiVersion       PRIVATE
```

1.3.4 效果

下面的步骤要加载并且运行已经创建的 ARX 应用程序：

（1）打开前一小节中创建的Hello1 工程，选择【Build/Start Debug/Go】菜单项，或者直接按下快捷键F5，系统会弹出如图 1.22所示的信息提示框，单击【是】按钮创建ARX文件。

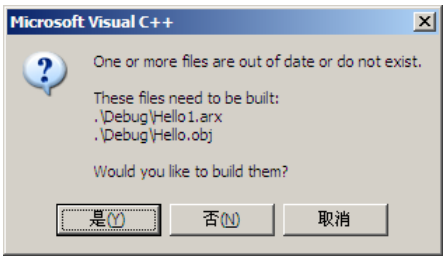

图1.22　编译提示

（2）编译无错之后，系统会弹出如图 1.23所示的对话框，提示用户输入可执行文件的路径（也就是调用DLL的应用程序，在这里就是AutoCAD 2002 的位置）。单击文本框右侧的按钮，从弹出的快捷菜单中选择【Browser】菜单项，并从弹出的对话框中选择acad.exe文件。关闭对话框之后，AutoCAD 2002 应用程序的路径就出现在文本框中，单击【OK】按钮。

图1.23　选择 AutoCAD 2002 的路径

（3）系统弹出如图 1.24所示的对话框，提示用户AutoCAD 2002 未包含任何的调试信息。选择【Do not prompt in the future】复选框，单击【OK】按钮继续进行调试。

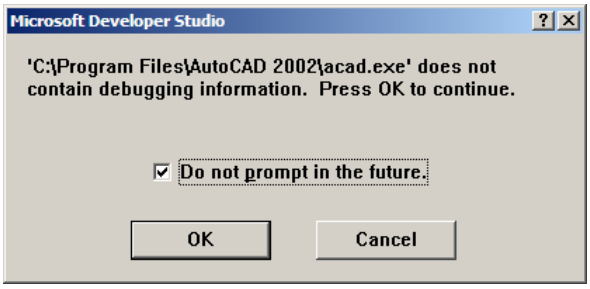

图1.24　关闭下一次的提示

（4）AutoCAD 2002 被自动启动，在其中选择【工具 / 加载应用程序】菜单项，系统会弹出如图 1.25所示的【加载 / 卸载应用程序】对话框。选择在指定的输出路径中生成的Hello.arx文件，单击【加载】按钮，将其加载到AutoCAD 2002 中，单击【关闭】按钮。

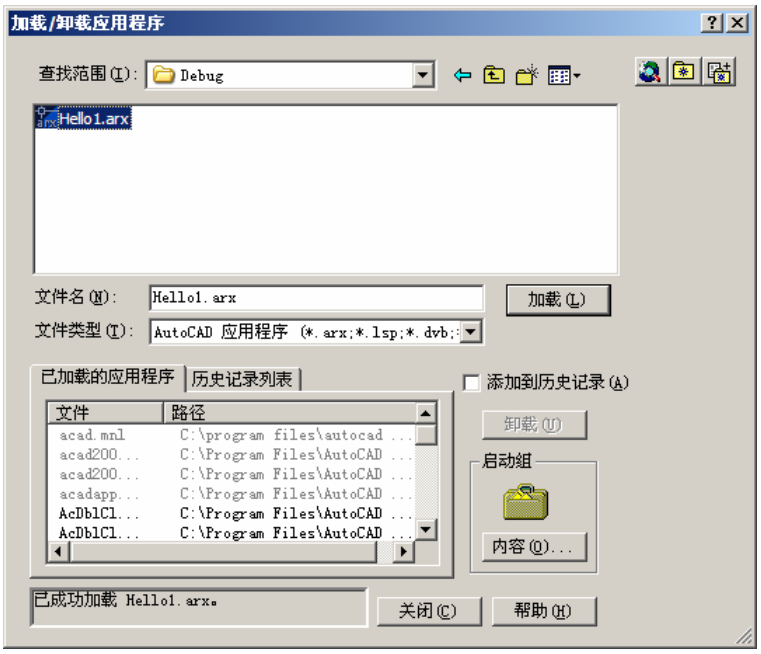

图1.25 加载生成的 ARX 文件

（5）在AutoCAD 2002 命令行中键入hello并按下Enter键，命令行会显示“Hello,World!”文字，如图 1.26所示。

图1.26 命令执行结果

1.3.5 小结

本节用纯手工的方式创建了一个基本的 ARX 程序，其主要目的在于让读者了解编译器的设置和 ARX 程序的基本结构。这两个问题之所以关键，是因为下一节开始，我们就要使用 ObjectARX 提供的向导来生成应用程序，向导会生成大量的代码，不利于理解基本的程序框架。

在本节结束时，提示一个小的问题。在创建应用程序的步骤（5）和（8）中，分别指定了 ObjectARX 程序所需要的头文件和库文件路径。除了这种方法之外，还可以按照下面的方法来进行设置：

（1）在VC++ 6.0 中，选择【Tools/Options】菜单项，系统会弹出如图 1.27所示的【Options】对话框。进入【Directories】选项卡，向路径列表中添加ObjectARX头文件的路径。

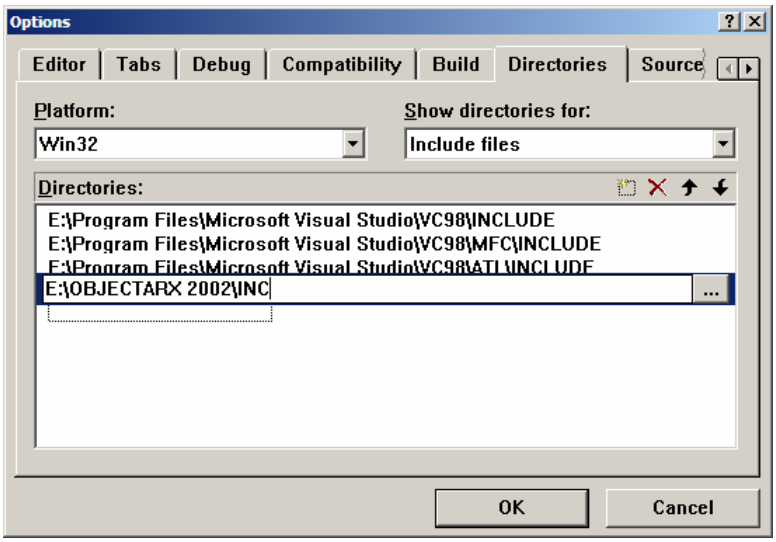

图1.27　添加 ObjectARX 的头文件路径

（2）从【Show directories for】列表框中选择【Library files】选项，然后在路径列表中添加ObjectARX的库文件路径，如图 1.28所示。

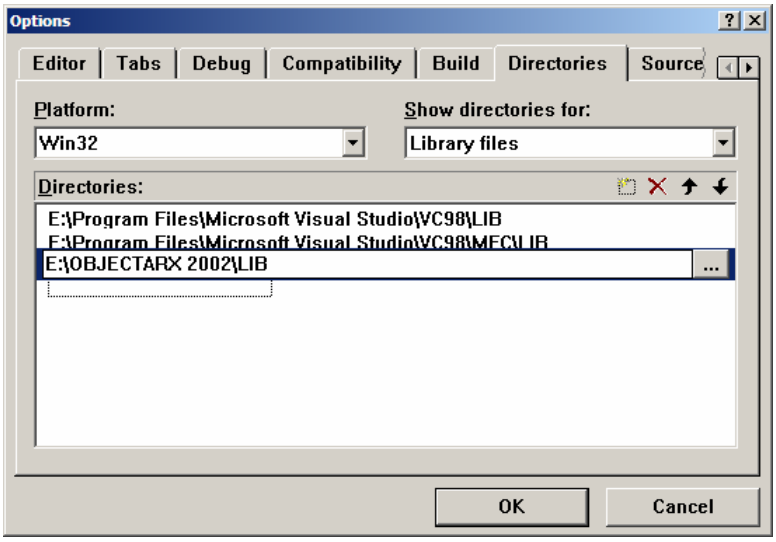

图1.28　添加 ObjectARX 的库文件路径

如果在【Options】对话框中设置了 ObjectARX 的库文件和头文件路径，不必在步骤（5）和（8）中设置附加文件路径，也可以顺利编译 ARX 文件，这是一种替代的方法。

虽然两种方法都能达到同样的效果，甚至直接在【Options】对话框中设置路径能达到一劳永逸的效果，但是当用户同时安装了多个 ObjectARX 的开发包时，就不能直接在【Options】对话框中添加多个版本 ObjectARX 的路径了，这样可能会引起编译时出现一些莫名其妙的错误。

由于本书中所有的程序都在 AutoCAD 2002平台下开发，因此直接在 VC++ 6.0的【Options】对话框中设置了 ObjectARX 的库文件和头文件路径，而未给所有的工程添加附加

文件路径。

1.4 用向导创建 Hello,World 程序

1.4.1 说明

本节使用 ObjectARX 向导创建一个 Hello,World 程序，了解向导创建的工程的基本结构，并且学习使用 ObjectARX 内嵌工具栏注册命令的方法。

程序执行的结果与上一节的程序一样，在 AutoCAD 命令行显示“Hello，World!”字符串。

1.4.2 思路

使用向导创建 ObjectARX 应用程序，能够大大简化操作的步骤，向导自动生成了大量的代码。同学习 MFC 一样，如果了解了向导工作的内容，那么使用向导能够大大节省工作量；如果不了解，则可能越用越糊涂。

1.4.3 步骤

（1）启动VC++ 6.0，选择【File/New】菜单项，系统会弹出如图1.29所示的对话框。从项目列表中选择【ObjectARX 2000/2000i/2002 AppWizard】选项，输入Hello2作为项目名称，指定适当的保存位置，单击【OK】按钮。

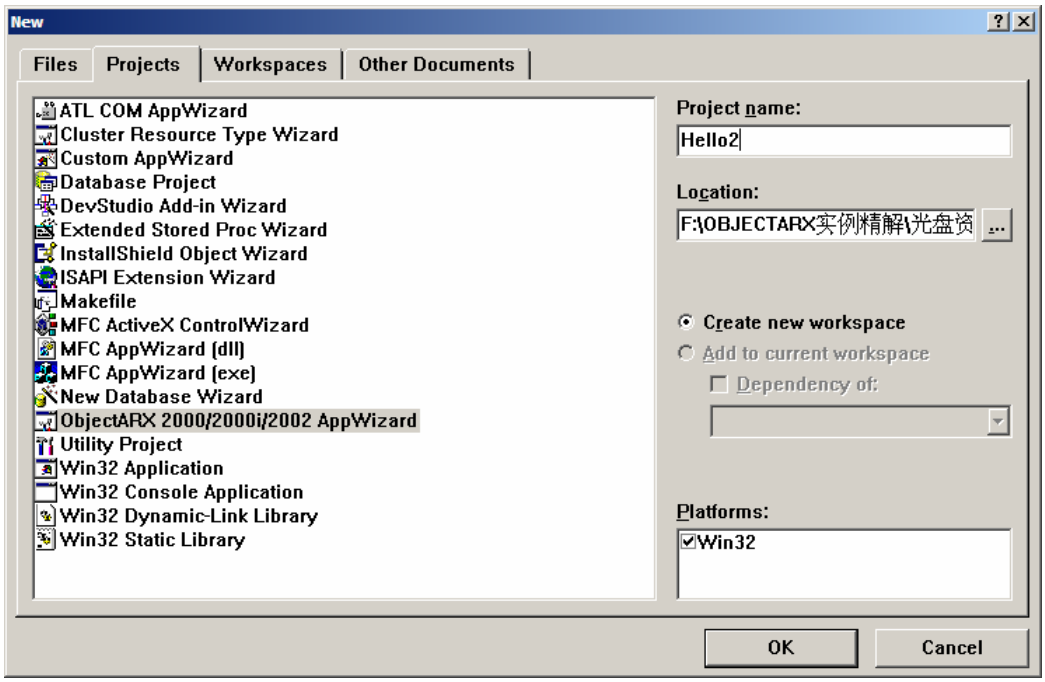

图1.29　输入项目名称和保存位置

（2）系统会弹出如图1.30所示的对话框。输入你的注册名称（可用公司名称或你的个人名称作为前缀，避免和其他工程在命名上的重复），其他选项使用默认值，单击【Next】按钮。

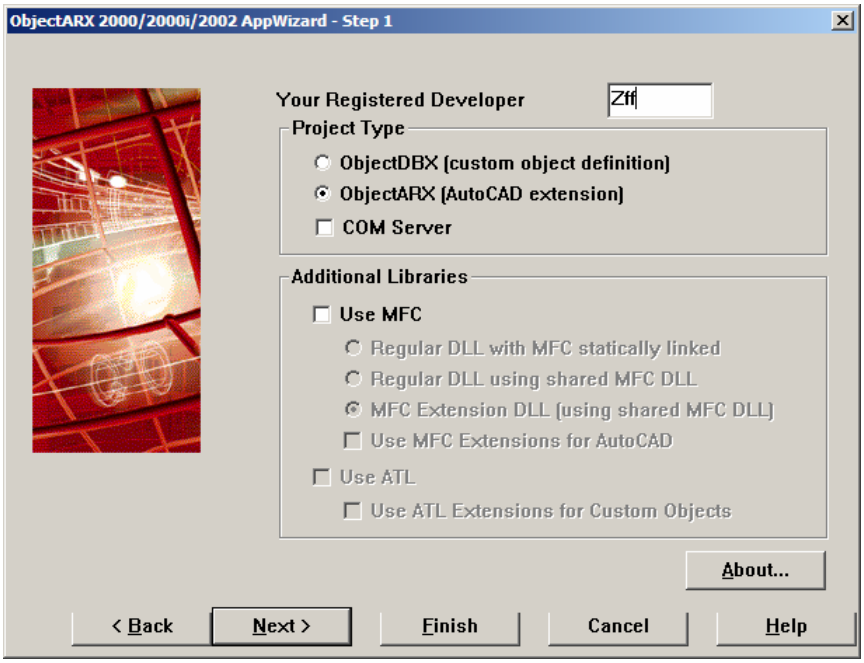

图1.30　输入工程选项

（3）系统会弹出如图1.31所示的对话框，显示了已经创建的项目信息，包含了向导创建

的各个文件及其作用：

- StdAfx.cpp 和 StdAfx.h：预编译头文件的创建。
- Hello2.cpp：应用程序入口点。
- Hello2.def：模块输出文件。
- StdArx.h：项目特定函数所需要的通用头文件声明。
- Resource.h：资源标记的声明。
- res\Hello2.rc2：工程的第二个资源文件。
- ObjectARX.prj：ObjectARX 2000 插件所需要的项目设置。
- RxDebug.cpp 和 RxDebug.h：有用的调试工具。
- DocData.h 和 DocData.cpp：文档数据的封装类。
- AdskDMgr.h：定义 Autodesk 数据管理模板类。

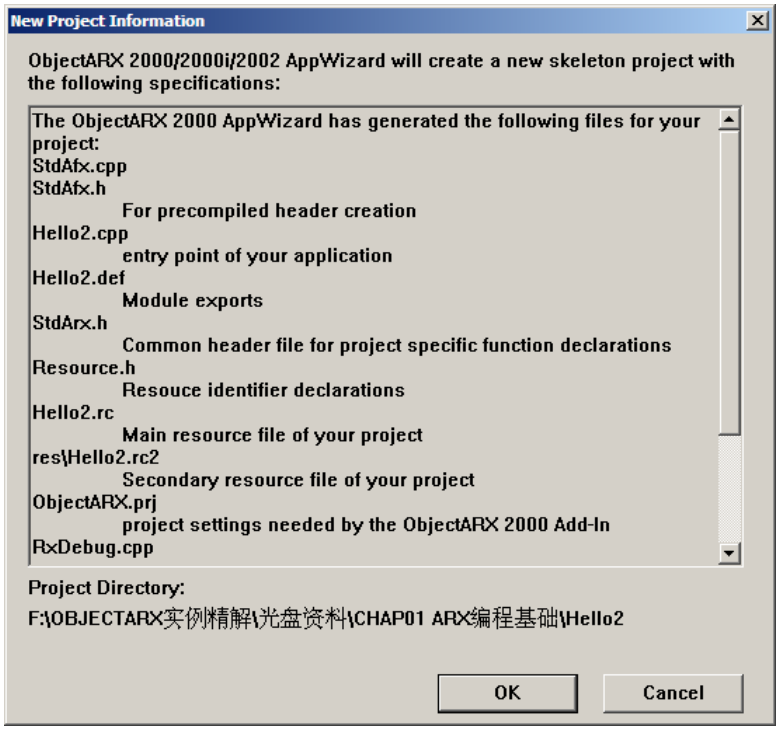

图1.31 项目信息的汇总

（4）单击【OK】按钮关闭对话框，完成项目的创建。如果你愿意，可以选择【Project/Settings】菜单项，切换到【Debug】选项卡，在【Executable for debug session】文本框中指定AutoCAD 2002的位置，如图1.32所示。这样，在第一次调试程序时，系统就不会提示用户指定可执行程序的位置，也就是图1.23的提示。

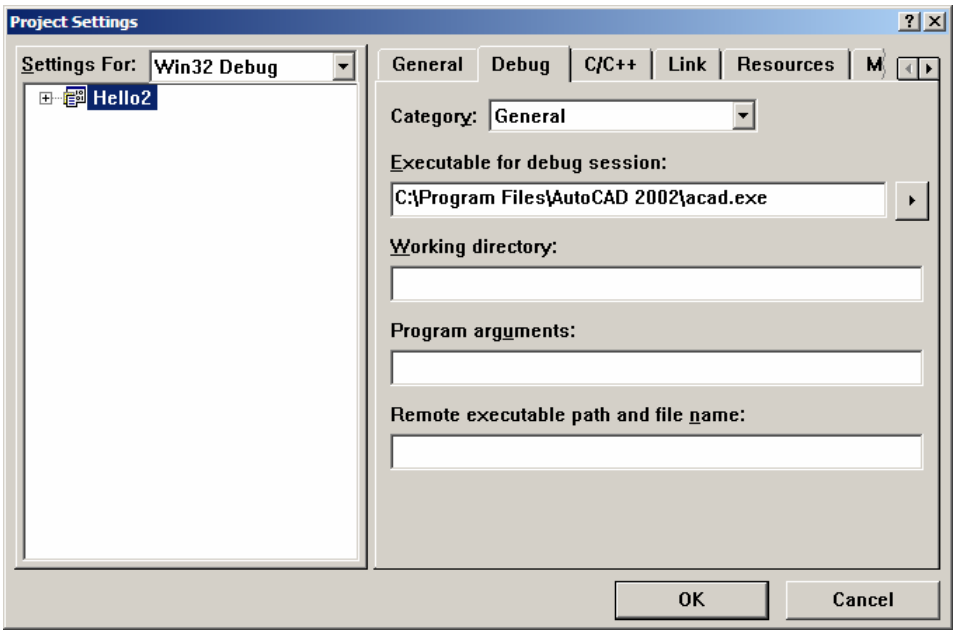

图1.32　输入 AutoCAD 2002 的位置

（5）分析向导生成的代码。这里还不准备探讨向导生成的 AsdkDataManager 和 CDocData 类的作用，现在看起来还比较难以理解，何况它们对现在的程序没有什么影响，在后面的篇幅中会慢慢介绍。在 Hello2.cpp 文件中，向导创建了如下几个函数：

- AcrxEntryPoint 函数：ObjectARX 入口点函数，在上节的程序中已经介绍，这里也并无特殊之处。
- InitApplication 函数：同样在上节已经介绍，用于应用程序的初始化。
- UnloadApplication 函数：已介绍过，用于清理应用程序的相关命令。
- _hdllInstance 变量：在 DllMain 函数中使用，一般不用注意。
- DllMain 函数：动态链接库（DLL）的入口点函数，一般来说不用注意。
- AddCommand 函数：封装了 addCommand 函数，用于向 AutoCAD 注册命令。

（6）注册一个新的Hello命令。单击ObjectARX嵌入工具栏的“ObjectARX commands”按钮，系统会弹出如图1.33所示的注册命令对话框。在【Command flags】选项组中，从【Document】列表中选择【Shared write】选项，取消选择【Use pickset】复选框。在【Group】文本框中输入BASIC，【International】文本框中输入Hello，左键在【Local】文本框内单击，系统自动添加Hello文本，使用系统自动给出的名称ZffBASICHello作为Hello命令执行的函数名称。

设置命令名称和标记之后，单击【Add】命令就能注册该命令。单击【OK】按钮关闭命令注册的对话框。

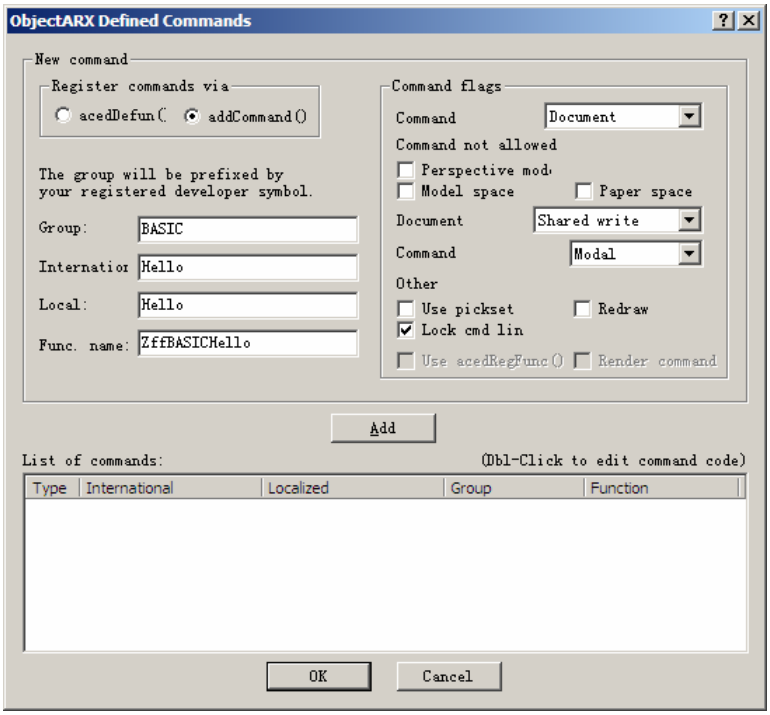

图1.33 注册新的 Hello 命令

（7）注册命令之后，系统自动生成了一个 Hello2Commands.cpp 文件，其中包含了 Hello 命令的实现函数 ZffBASICHello 的定义，当然还没有任何的内容。在该函数中添加代码：

```
void ZffBASICHello()
{
        acutPrintf("Hello,World!");
}
```

到此为止，项目全部完成！你应该能看出使用向导节省了多少工作量。

1.4.4 效果

在VC++ 6.0中，按下快捷键F5对程序进行调试，系统会自动启动AutoCAD 2002。使用APPLOAD命令加载生成的ARX文件，在命令行执行Hello命令，能够得到如图1.34所示的结果。

图1.34 程序运行的结果

1.4.5 小结

后面的程序全部使用向导来创建，规模也会更大，这里用向导生成简单的 Hello,World 程序，目的就是让读者经过分析掌握向导的辅助作用，而不至于在大量的辅助代码中迷失。

从现在开始，我们要专注于 ObjectARX 的精髓内容。冒险，才刚刚开始。

第2章 创建和编辑基本图形对象

在我学习 ObjectARX 的时候，曾经试图直接去学习数据库的操作、几何类等概念，结果并不理想。经过一段时间的摸索，我将创建和编辑基本图形对象作为突破口，逐步深入，在学习过程中成就感很强。

因此，我将这种方法引入本书，希望能以更直观的方式引导你进入 ObjectARX 编程的大观园。

2.1 创建直线

2.1.1 说明

本实例运行的结果是在 AutoCAD 2002中，创建一条直线，该直线的起点是（0，0，0），终点是（100，100，0）。除此之外，不准备再做更多的事情。

麻雀虽小，五脏俱全。通过这个程序，你将要开始了解 AutoCAD 数据库的基本结构，这比学习 VBA 和 AutoLISP 都要困难，但是更有意思。

2.1.2 思路

首先来看看，在 AutoCAD 中，使用 LINE 命令创建一条直线，需要哪些东西：

命令: _line
指定第一点: 0,0
指定下一点或 [放弃(U)]: 100,100
指定下一点或 [放弃(U)]:

从上面的命令提示可以看出，创建一条直线，需要用户指定起点和终点。

在继续之前，必须给大家介绍一点数据库最基础的几个名词：

- 表：表是数据库的组成单位，一个数据库至少包含一个表。
- 记录：记录是表的组成单位，一个表可能包含多条记录，也可能不包含任何记录。

图2.1用来描述AutoCAD数据库的基本结构再好不过了。从图中来看，实体包含在块表记录中，因此要创建一个图形对象，需要遵循下面的基本步骤：

（1）确定要创建对象的图形数据库；

（2）获得图形数据库的块表；

（3）获得一个存储实体的块表记录，所有模型空间的实体都存储在模型空间的特定记录中。

（4）创建实体类的一个对象，将该对象附加到特定的块表记录中。

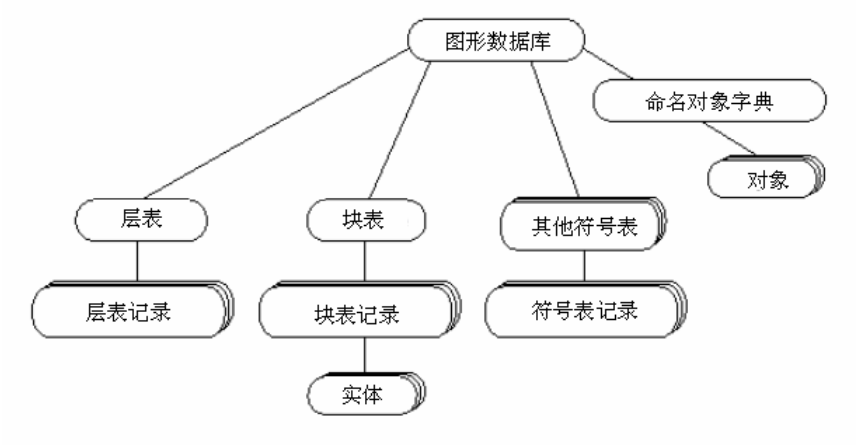

图2.1　图形数据库的结构

2.1.3 步骤

（1）在VC++ 6.0中，使用向导创建一个新的ObjectARX项目（名称为CreateLine），其方法与1.4节完全一致。使用ObjectARX嵌入工具注册一个新命令，命令的各项参数如图2.2所示。

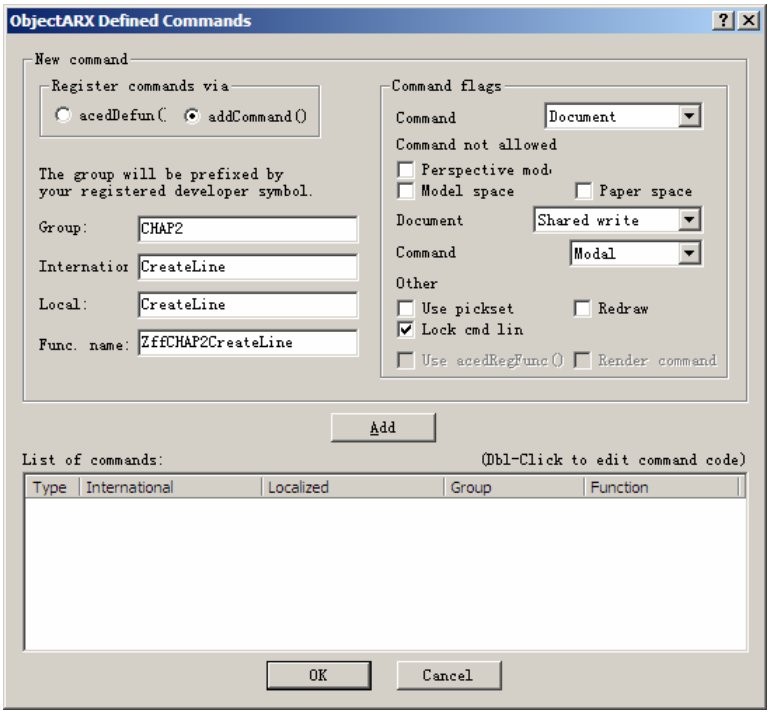

图2.2　设置命令的参数

（2）在 ZffCHAP2CreateLine 函数中，添加创建直线对象（在 ObjectARX 中，AcDbLine 类代表直线）的代码：

```
// 在内存上创建一个新的AcDbLine对象
AcGePoint3d ptStart(0, 0, 0);
AcGePoint3d ptEnd(100, 100, 0);
AcDbLine *pLine = new AcDbLine(ptStart, ptEnd);
```

注意，基于 AutoCAD 内部的实现机制，必须在堆上创建对象，而不能用下面的语句创建直线的对象：

```
AcDbLine line(ptStart, ptEnd);
```

此时，直线对象仅被在内存上创建，并没有添加到图形数据库中，因此不可能会显示在图形窗口中。

（2）在 ZffCHAP2CreateLine 函数中，添加获得指向块表的指针的相关代码：

```
// 获得指向块表的指针
AcDbBlockTable *pBlockTable;
acdbHostApplicationServices()->workingDatabase()
                    ->getBlockTable(pBlockTable, AcDb::kForRead);
```

acdbHostApplicationServices()->workingDatabase()能够获得一个指向当前活动的图形数据库的指针，这在后面还要经常遇到。getBlockTable 是 AcDbDatabase 类的一个成员函数，用于获得指向图形数据库的块表的指针，其定义为：

```
inline Acad::ErrorStatus getBlockTable(
                    AcDbBlockTable*& pTable,
                    AcDb::OpenMode mode);
```

该函数的返回值 Acad::ErrorStatus 是 ObjectARX 中定义的一个枚举类型，主要用于判断函数的返回状态，如果函数成功执行会返回 Acad::eOk。第一个参数 pTable 返回指向块表的指针；第二个参数同样是一个枚举类型的变量，其类型 AcDb::OpenMode 包含了 AcDb::kForRead、AcDb::kForWrite 和 AcDb::kForNotify 三个可取的值，创建直线的时候不需要更改块表，因此这里打开的模式为 AcDb::kForRead。

如果对 C++的概念理解不够深入，很可能会不了解 AcDbBlockTable*& pTable 的含义。从左向右来读，就可知道该形式参数的类型是 AcDbBlockTable*（指向块表的指针），由于引用运算符（&）的存在，那么形参 pTable 是一个**指针的引用**。

下面的函数相信在学习引用时大部分人都遇到过：

```
void swap(int &m, int &n);
```

没错，之所以在参数中使用引用运算符，是为了通过函数的参数实现返回值。在 getBlockTable 函数中是一样的情况，不过形参的类型变成了一个指针而已。

（3）在 ZffCHAP2CreateLine 函数中，添加获得指向特定块表记录的指针的相关代码：

```
// 获得指向特定的块表记录（模型空间）的指针
AcDbBlockTableRecord *pBlockTableRecord;
pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);
```

getAt 函数是 AcDbBlockTable 类的一个成员函数，用于获得块表中特定的记录，其定义为：

```
Acad::ErrorStatus getAt(
                const char* entryName,
                AcDbBlockTableRecord*& pRec,
                AcDb::OpenMode openMode,
                bool openErasedRec = false) const;
```

第一个参数用于指定块表记录的名称，ACDB_MODEL_SPACE 是 ObjectARX 中定义的一个常量，其内容是“*Model_Space”；第二个参数用于返回指向块表记录的指针；第三个参数指定了块表记录打开的模式，下一步要向块表记录中添加实体，所以就用写的模式（AcDb::kForWrite）打开；第四个参数指定是否查找已经被删除的记录，这里暂时不深入介绍，后面在合适的地方我会谈到它，一般使用默认的参数值。

（4）在 ZffCHAP2CreateLine 函数中，添加向块表记录中附加实体的代码：

```
// 将AcDbLine类的对象添加到块表记录中
AcDbObjectId lineId;
pBlockTableRecord->appendAcDbEntity(lineId, pLine);
```

appendAcDbEntity 是 AcDbBlockTableRecord 类的成员函数，用于将 pEntity 指向的实体添加到块表记录和图形数据库中，其定义为：

```
Acad::ErrorStatus appendAcDbEntity(
                AcDbObjectId& pOutputId,
                AcDbEntity* pEntity);
```

第一个参数返回图形数据库为添加的实体分配的 ID 号；第二个参数指定了所要添加的实体。

（笑谈 AcDbObjectId）多次在论坛上有人问什么是 ID（身份、标识），我给过这样的解释：看过周星驰的《唐伯虎点秋香》吧？由于下人太多，名字不好记，每个人都被编了一个号码（唐伯虎自然就是 9527），管理者通过这个编号来管理。图形数据库也一样，作为管理者，只能通过编号（AcDbObjectId）来管理每一个实体。

（5）在 ZffCHAP2CreateLine 函数中，添加关闭图形数据库各种对象的代码：

```
// 关闭图形数据库的各种对象
pBlockTable->close();
pBlockTableRecord->close();
pLine->close();
```

在操作图形数据库的各种对象时，必须遵守 AutoCAD 的打开和关闭对象的协议。该协议确保当对象被访问时在物理内存中，而未被访问时可以被分页存储在磁盘中。创建和打开数据库的对象之后，必须在不用的时候关闭它。

初学 ObjectARX 的人肯定会有两点疑问：

- 各种数据库对象的关闭：在打开或创建数据库对象之后，必须尽可能早的关闭它。在初学者所犯的错误中，未及时关闭对象的错误至少占一半！
- **不要使用 delete pLine 的语句。**对 C++比较熟悉的读者，习惯于配对使用 new 和 delete 运算符，这在 C++编程中是一个良好的编程习惯。但是在 ObjectARX 的编程中，当

编程者使用 appendAcDbEntity 函数将对象添加到图形数据库之后，就需要由图形数据库来操作该对象。

（6）最后，来看一下完整的代码：

```
void ZffCHAP2CreateLine()
{
        // 在内存上创建一个新的AcDbLine对象
        AcGePoint3d ptStart(0, 0, 0);
        AcGePoint3d ptEnd(100, 100, 0);
        AcDbLine *pLine = new AcDbLine(ptStart, ptEnd);

        // 获得指向块表的指针
        AcDbBlockTable *pBlockTable;
        acdbHostApplicationServices()->workingDatabase()
                ->getBlockTable(pBlockTable, AcDb::kForRead);

        // 获得指向特定的块表记录（模型空间）的指针
        AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);

        // 将AcDbLine类的对象添加到块表记录中
        AcDbObjectId lineId;
    pBlockTableRecord->appendAcDbEntity(lineId, pLine);

        // 关闭图形数据库的各种对象
        pBlockTable->close();
    pBlockTableRecord->close();
    pLine->close();
}
```

（7）按下键盘上的快捷键F7，对上面的代码进行编译，肯定会得到两个错误，如图2.3所示。这就引出另一个在ObjectARX编程中常见的问题：包含适当的头文件。

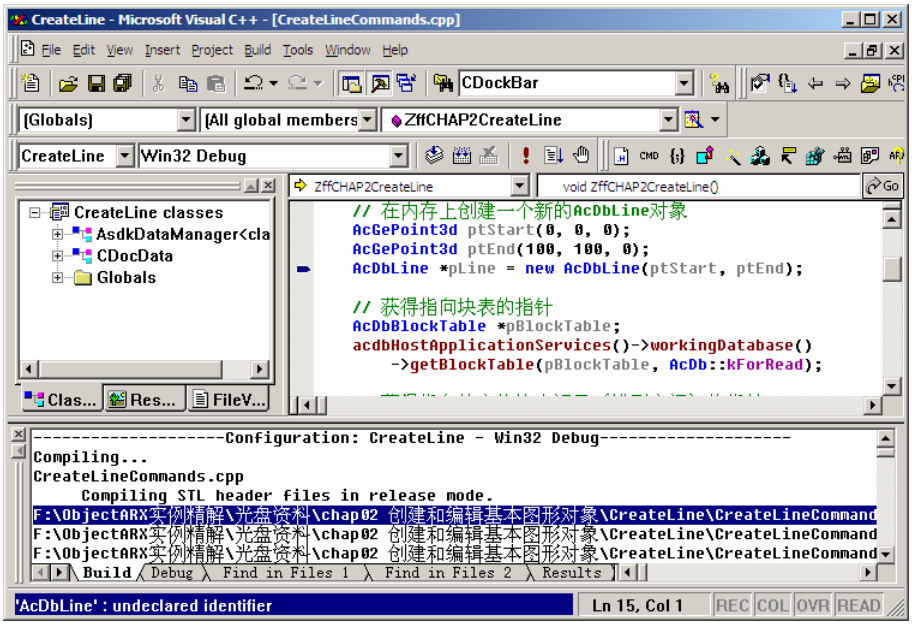

图2.3　编译出错的位置和错误提示

编译错误的提示为：AcDbLine 是一个未定义的标识符。已经知道了 AcDbLine 是 ObjectARX 中的一个类，那么就要找到其定义的位置，将其所在的头文件包含到当前文件中来。如何确定 AcDbLine 类需要包含哪个头文件呢？

打开ObjectARX帮助文档中的ObjectARX Reference，转到【索引】选项卡，在文本框中输入AcDbLine（如果你已经指定了ObjectARX帮助文件的位置，并且为其定义了快捷键，就可以像我这样，直接按下Alt+F1，自动完成上面的手工操作），在列表中双击AcDbLine Class选项，得到如图2.4所示的结果。

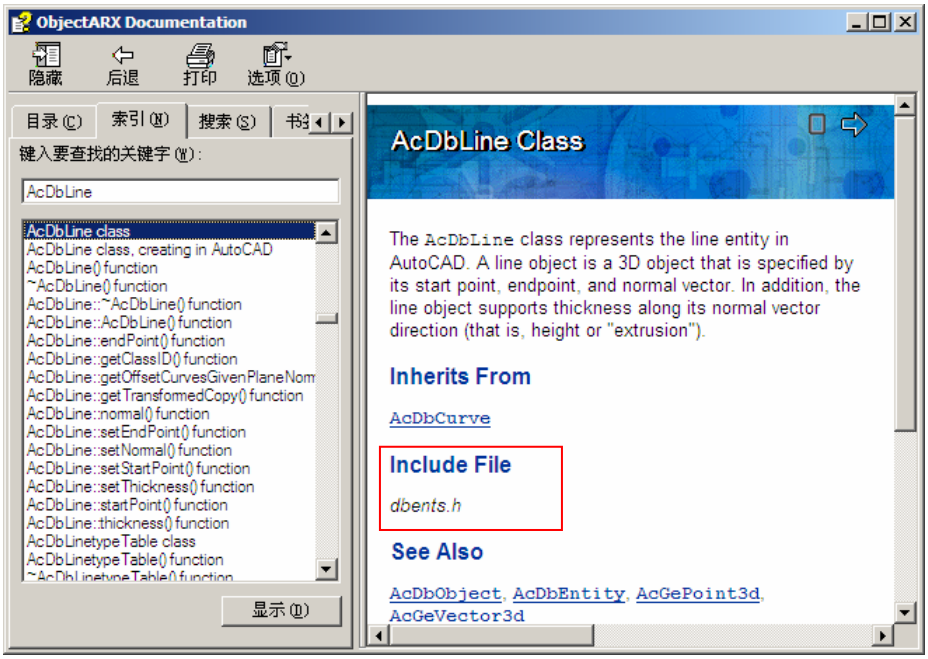

图2.4 确定所要包含的头文件

从图中的红框内你可以发现一个秘密，没错，要确定一个 ObjectARX 关键字、类、全局函数需要包含的头文件，都可以如法炮制。

（8）添加包含头文件的语句。已经知道 AcDbLine 类需要包含什么头文件，就可以在 CreateLineCommands.cpp 文件的开头添加下面的语句：

```
#include "dbents.h"
```

再次按下 F7 键编译程序，不会再出现错误和警告，并且生成了目标文件 ZffCreateLine.arx。

提示：关于包含头文件语句放置的位置，还值得讨论一下。如果使用向导生成 ObjectARX 项目，想到会自动创建一个名为 StdArx.h 的文件，由于该文件自动被所有的源文件包含，因此其中可以放置一些经常要被包含的头文件（放在注释语句“// TODO: Here you can add your own includes / declarations”之后即可）；如果某个头文件可能仅会在一个文件内使用，那么可以像本节实例那样，直接放在源文件的开头。

希望从上面的过程中，你可以了解到创建图形实体的一般过程。Ok，如果有足够的信心，可以试写一下创建圆的代码。

重要提示：在 ObjectARX 程序中会大量地涉及到包含头文件的语句，如果每次都在书中出现这些语句无疑会浪费大量的篇幅，因此后面的很多程序都不会贴出需要包含的头文件，读者可以自行根据上面的方法确定需要包含的头文件，或者从配套光盘中获得相应的包含语句。

2.1.4 效果

编译应用程序之后，在AutoCAD 2002中加载并运行程序中注册的CreateLine命令，能够得到如图2.5所示的结果。

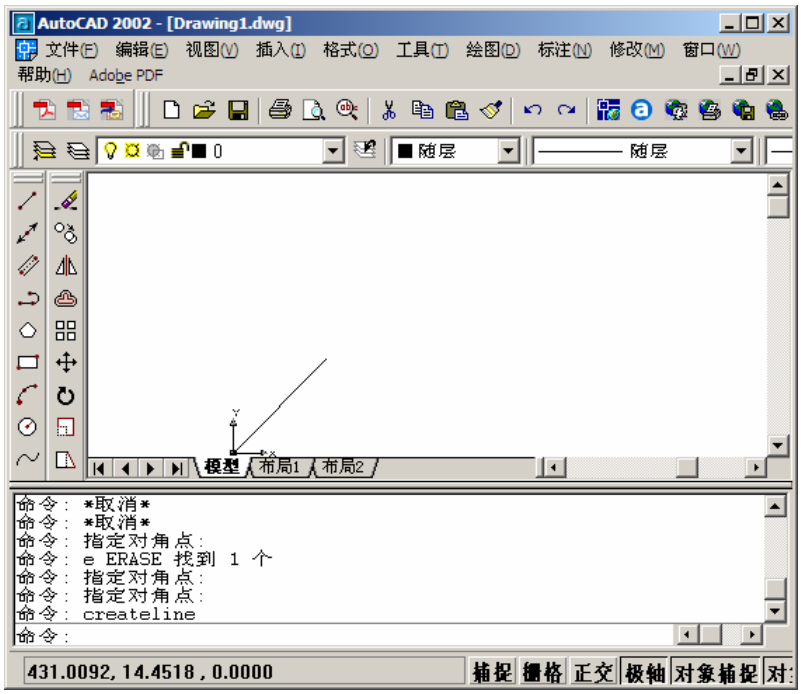

图2.5　创建直线的结果

2.1.5　小结

步骤（3）使用 getAt 函数获得了指向模型空间块表记录的指针，因此创建的直线是在模型空间，如果你想在图纸空间创建一条直线，那么你可以用 ACDB_PAPER_SPACE 作为 getAt 函数的第一个参数，其他的代码完全不用改变。

本节用了大量的篇幅来介绍在 ObjectARX 中创建一条直线的过程，目的是让你了解图形数据库中实体存储的结构，一个实例可能还太少了，于是，后面的内容跟着来了。

2.2 修改图形对象的属性

2.2.1　说明

上一节的学习，你已经能创建一条直线了，本节介绍的例子则会改变直线的颜色。所要实现的效果非常简单：创建一条直线之后，将它的颜色变为红色。

2.2.2　思路

如果是在创建时修改直线的颜色，就可以直接在上节的 ZffCHAP2CreateLine 函数中加入

下面的代码（放在关闭图形数据库各种对象之前）：

```
pLine->setColorIndex(1);
```

运行程序中注册的 CreateLine 命令，创建的直线颜色变为红色。

在实际编程中，并不是每一次都可以在创建对象时将其特性设置到合适的状态，相反，更多的时候可能在创建对象之后才修改其特性，本节正要解决这个问题。

1. 打开和关闭图形数据库的对象

访问图形数据库中对象的特性，必须在该对象被打开（对象创建时也会被打开）的状态下，用对象的指针进行访问，并且在访问结束后要及时关闭该对象，不然就会引起 AutoCAD 的错误终止。

创建一个对象，必须在创建之后关闭该对象，那么如何在某个时候再访问该对象？这就要用到 2.1 节介绍的 AcDbObjectId，也就是对象的 ID 号。在创建对象时，可以将图形数据库分配给该对象的 ID 保存起来，在需要访问该对象时，根据这个 ID 从数据库中获得指向该对象的指针，就可以修改或者查询该对象的特性。

AcDbBlockTableRecord 类的 appendAcDbEntity 函数能够将一个实体添加到图形数据库中，并且返回分配给该实体的 ID，这个函数上一节已经介绍过；全局函数 acdbOpenAcDbEntity 用于从实体的 ID 号获得指向图形数据库中实体的指针，其定义为：

```
Acad::ErrorStatus acdbOpenAcDbEntity(
                    AcDbEntity*& pEnt,
                    AcDbObjectId id,
                    AcDb::OpenMode mode,
                    bool openErasedEntity = false);
```

第一个参数返回指向图形数据库实体的指针；第二个参数输入了要获得的实体的 ID 号；第三个参数指定了打开该实体的方式，如果仅是查询该实体的特性用“读”模式打开即可，要修改实体的特性就必须用“写”模式打开；第四个参数指定是否允许访问一个已经被删除的实体。

ObjectARX 提供了另外两个全局函数 acdbOpenAcDbObject 和 acdbOpenObject 来实现类似的功能，这三个函数的区别在与适用范围：

- acdbOpenAcDbEntity：适用于打开继承于 AcDbEntity 的数据库常驻对象，这类对象一般都能在图形窗口中显示，如直线、圆等。
- acdbOpenAcDbObject：适用于打开未继承于 AcDbEntity 的数据库常驻对象，这类对象不能在图形窗口中显示，如层表、线型表等。
- acdbOpenObject：如何不知道要打开的对象是否继承于 AcDbEntity 类，可以使用这个函数。

打开某个对象之后，使用 close 函数就可以将其关闭。

2. ID（AcDbObjectId）、指针、句柄（Handle）和ads_name

访问实体的特性必须通过对象指针，但是一旦你获得了实体的ID、句柄或者ads_name，都能通过ID作中介而获得对象的指针。ID、指针、句柄和ads_name的关系如图 2.6所示，其

中ID是一个桥梁。

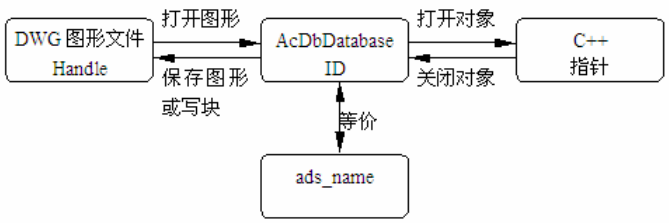

图2.6　ID、指针、句柄和 ads_name 的关系

想来 ID 和指针大家都比较了解，有必要解释一下句柄和 ads_name。句柄是 Windows 编程一个常用的概念，在 ObjectARX 编程中一般指 AcDbHandle 类（如果是指 Windows 编程的界面元素，可以从上下文环境中区分），该类封装了一个 64 位整形标识符，随 DWG 文件一同保存。ads_name 则是在 ADS 编程（ObjectARX 编程的前身）中出现的一个概念，其实际上是一个二维数组，数组元素类型为长整型，在与用户交互的函数中还会经常用到。

ID、句柄和 ads_name 具有各自的特点：

- ID：在一个 AutoCAD 任务中，可能会加载多个图形数据库，但是所有对象的 ID 在本次任务中都是独一无二的。在不同的 AutoCAD 任务中，同一个图形对象的 ID 可能不同。
- 句柄：在一个 AutoCAD 任务中，不能保证每个对象的句柄都唯一，但是在一个图形数据库中所有对象的句柄都是唯一的。句柄随 DWG 图形一起保存，在两次任务期间同一对象的句柄是相同的。
- ads_name：是不稳定的，仅当你在 AutoCAD 的一个特定图形中工作时可以使用，一旦退出 AutoCAD 或者切换到另一个图形，ads_name 就会丢失。

具体来说，ID、指针、句柄和 ads_name 之间具有下面的转换关系（不完全归纳，不常用的转换并未提及）：

- 从 ID 到对象指针：通过打开数据库对象的三个函数 acdbOpenAcDbEntity、acdbOpenAcDbObject 和 acdbOpenObject 中的任何一个。
- 从对象指针到 ID：所有的数据库常驻对象都继承自 AcDbObject，而 AcDbObject 类包含的 objectId 函数能获得所指向对象的 ID。
- 从句柄到 ID：使用 AcDbDatabase::getAcDbObjectId 函数。
- 从 ID 到句柄：使用 AcDbObjectId::handle 函数。
- 从指针到句柄：使用 AcDbObject::getAcDbHandle 函数。
- 从 ads_name 到 ID：使用全局函数 acdbGetObjectId。
- 从 ID 到 ads_name：使用全局函数 acdbGetAdsName。

2.2.3 步骤

（1）VC++ 6.0 中，使用 ObjectARX 向导创建一个新项目（名称为 ModifyEnt），使用 ObjectARX 嵌入工具注册一个名为 ChangeColor 的新命令。

（2）为了充分利用上一节的成果，将 CreateLine 函数复制到本节工程的命令实现文件

ModifyEntCommands.cpp 中，然后进行一点改动，具体代码为：

```
AcDbObjectId CreateLine()
{
        AcGePoint3d ptStart(0, 0, 0);
        AcGePoint3d ptEnd(100, 100, 0);
        AcDbLine *pLine = new AcDbLine(ptStart, ptEnd);

        AcDbBlockTable *pBlockTable;
        acdbHostApplicationServices()->workingDatabase()
                ->getBlockTable(pBlockTable, AcDb::kForRead);

        AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);

        AcDbObjectId lineId;
    pBlockTableRecord->appendAcDbEntity(lineId, pLine);

        pBlockTable->close();
    pBlockTableRecord->close();
    pLine->close();

        return lineId;
}
```

与上一节的函数相比，函数的返回值类型为 AcDbObjectId，并且在函数实现部分增加了返回的语句。return 语句返回了图形数据库为新添加的直线分配的 ID。

此外，AcDbLine 类所需要的头文件包含语句也要添加到该文件中：

```
#include "dbents.h"
```

（3）创建 ChangeColor 函数，用于修改指定实体的颜色。该函数包含了两个参数，第一个参数指定了要修改颜色的实体的 ID，第二个参数指定了要使用的颜色索引值，具体代码为：

```
Acad::ErrorStatus ChangeColor(AcDbObjectId entId, Adesk::UInt16 colorIndex)
{
        AcDbEntity *pEntity;
        // 打开图形数据库中的对象
    acdbOpenObject(pEntity, entId, AcDb::kForWrite);

        // 修改实体的颜色
    pEntity->setColorIndex(colorIndex);
```

```
                    // 用完之后，及时关闭
        pEntity->close();

        return Acad::eOk;
    }
```

正如前面介绍，使用 acdbOpenObject 函数实现了从 ID 到对象指针的转换，进而使用 AcDbEntity::setColorIndex 函数设置对象的颜色。参数中的 colorIndex 实际上就是 AutoCAD 中所使用的颜色索引，可以指定 0～256 的值，其中 0 代表随块，256 代表随层。在设置特性之后，记得要将打开的实体关闭。

Acad::ErrorStatus 的含义前面已经介绍过，它的目的就相当于一般使用 BOOL 类型作为函数的返回值，用来标识函数是否执行成功。Acad::ErrorStatus 是一个枚举类型，定义了多个可取的值，用来作返回值能够显示各种不同类型的错误，比使用 BOOL 类型含义要丰富。

（4）在 ChangeColor 命令的实现函数中，添加下面的代码：

```
void ZffCHAP2ChangeColor()
{
                    // 创建直线
                    AcDbObjectId lineId;
                    lineId = CreateLine();

                    // 修改直线的颜色
                    ChangeColor(lineId, 1);
}
```

代码很简单，首先调用自定义函数 CreateLine 创建一条直线，然后调用函数 ChangeColor 函数修改直线的颜色为红色。在这个过程中，**AcDbObjectId 类型的局部变量 lineId 作为了传递对象的中介，这种方法在 ObjectARX 中非常普通。**

2.2.4 效果

在 VC++中，按下快捷键 F5，编译并运行程序。启动 AutoCAD 2002 中，加载生成的 ARX 文件，在命令行执行 ChangeColor 命令，就能在图形窗口的左下角得到一条红色的直线。

2.2.5 小结

这一节是结束的时候了，但是还有点意犹未尽，总觉得还有两个重要的问题没有解决，于是在小结里一并提出。

1. 用类来组织函数

如果你对 C 语言还算熟悉，你应该会主动将两个自定义函数 CreateLine 和 ChangeColor 放在 ZffCHAP2ChangeColor 函数前面，于是本节的程序成功编译；如果你对 C 语言的概念还有些含糊，没有这样做，你肯定在怪罪作者给出了一个无法成功编译的程序。真的出现后者

的情况，你最好还是再温习一下 C 和 C++的知识。

即使能成功编译，还是有一个问题：如果这个项目注册了十几个命令，每个命令都要调用若干个自定义函数，那 ModifyEntCommands.cpp 文件的体积会有多大？这些东西看起来是多么难以维护？这才几十个函数，如果是上百个呢？一定要想办法解决这个问题才行。

如果你熟悉面向对象的程序设计，肯定会向导用类来组织这些函数，例如创建一个包含了创建实体的函数的新类 CCreateEnt，一个包含修改实体函数的类 CMofifyEnt。CreateLine 和 ChangeColor 分别作为两个类的成员函数。

由于在本章后面都会使用这种形式来组织函数，因此这里把本节的函数转换一下，用类来组织。下面的步骤创建一个新项目，作为后面部分的基础：

（1）VC++ 6.0 中，使用ObjectARX向导创建一个新项目（名称为CreateEnts），在向导中注意要选中【Use MFC】复选框，如图 2.7所示。

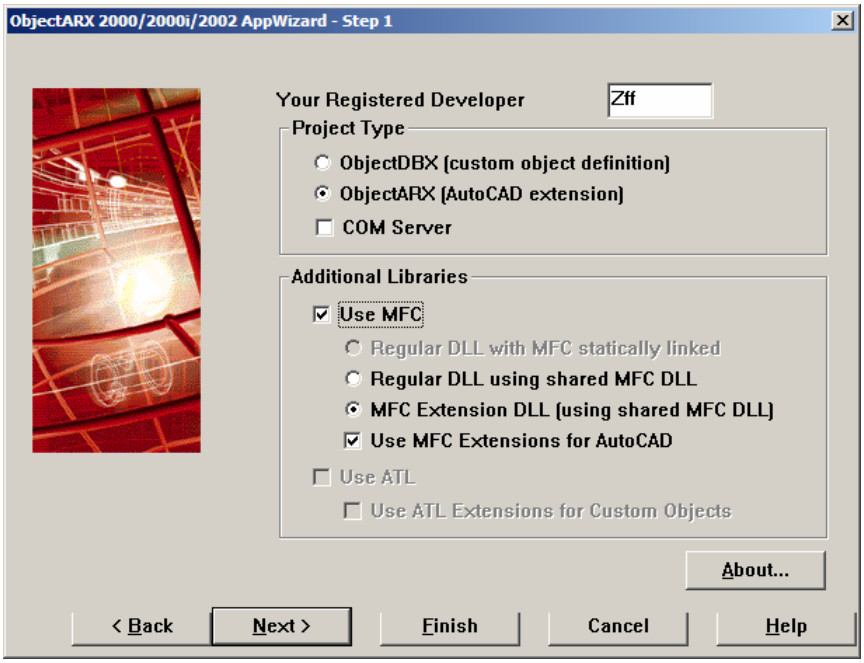

图2.7　设置项目的特性

（2）选择【Insert/New Class】菜单项，系统会弹出如图 2.8所示的对话框。输入CCreateEnt作为新类的名称，单击【OK】按钮创建新类。使用同样的方法，创建一个名为CmodifyEnt的类。

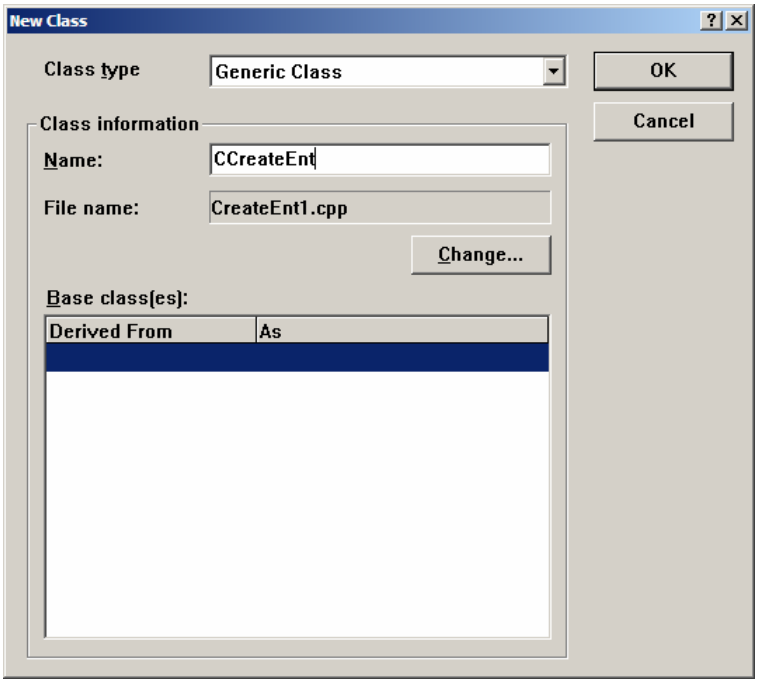

图2.8　创建新类

（3）在 CCreateEnt 和 CModifyEnt 类的头文件中，分别添加包含头文件的语句：

```
#include "StdArx.h"
```

然后在 StdArx.h 文件中，添加下面的包含语句：

```
#include "dbents.h"
```

（4）在 CCreateEnt 类中添加一个 CreateLine 成员函数，其声明和实现形式分别为：

```
static AcDbObjectId CreateLine();       // 函数声明
AcDbObjectId CCreateEnt::CreateLine()                    // 函数实现
{
                ……
}
```

由于函数的内容并未改变，因此这里不再给出具体的实现代码。

（4）在 CModifyEnt 类中添加一个 ChangeColor 成员函数，其声明和实现形式分别为：

```
static Acad::ErrorStatus ChangeColor(AcDbObjectId entId, Adesk::UInt16
colorIndex);
Acad::ErrorStatus CModifyEnt::ChangeColor(AcDbObjectId entId, Adesk::UInt16
colorIndex)
{
                ……
}
```

（5）此时，如果要注册一个新的命令，实现 ZffCHAP2ChangeColor 函数的功能，就要将代码写成这样：

```
void ZffCHAP2ChangeColor()
{
        // 创建直线
        AcDbObjectId lineId;
        lineId = CCreateEnt::CreateLine();

        // 修改直线的颜色
        CModifyEnt::ChangeColor(lineId, 1);
}
```

2. 修改实体的其他特性

已经介绍了修改实体颜色的方法，你可以试着写几个其他的函数，例如修改实体的图层和线型，写完之后，可以和下面的样例函数对照一下：

```
Acad::ErrorStatus CModifyEnt::ChangeLayer(AcDbObjectId entId,
                                          CString strLayerName)
{
        AcDbEntity *pEntity;
        // 打开图形数据库中的对象
    acdbOpenObject(pEntity, entId, AcDb::kForWrite);

        // 修改实体的图层
    pEntity->setLayer(strLayerName);

        // 用完之后，及时关闭
    pEntity->close();

    return Acad::eOk;
}

Acad::ErrorStatus CModifyEnt::ChangeLinetype(AcDbObjectId entId,
                                             CString strLinetype)
{
        AcDbEntity *pEntity;
        // 打开图形数据库中的对象
    acdbOpenObject(pEntity, entId, AcDb::kForWrite);

        // 修改实体的线型
    pEntity->setLayer(strLinetype);
```

```
                    // 用完之后，及时关闭
        pEntity->close();

        return Acad::eOk;
    }
```

可以看出，与修改实体颜色的函数相比，修改实体图层和线型的函数仅仅是使用了 AcDbEntity 类的不同的编辑函数，在打开和关闭数据库对象方面没有任何的区别。

3. 提高CreateLine函数的可重用性

CCreateEnt::CreateLine 函数仅能创建一条起点为（0，0，0）、终点为（100，100，0）的直线，局限性很大，因此要将它改造一下，便于在程序中调用。另外，考虑到所有的图形对象创建过程都要进行获得图形数据库的块表、获得模型空间的块表记录、将实体添加到模型空间的块表记录等操作，将这些步骤分离出来，封装成一个独立的函数。

于是，CCreateEnt 类现在包含了两个静态成员函数：

```
    // 创建直线
    AcDbObjectId CCreateEnt::CreateLine(AcGePoint3d ptStart,
                                                    AcGePoint3d ptEnd)
    {
                    AcDbLine *pLine = new AcDbLine(ptStart, ptEnd);

                    // 将实体添加到图形数据库
                    AcDbObjectId lineId;
                    lineId = CCreateEnt::PostToModelSpace(pLine);

                    return lineId;
    }

    // 将实体添加到图形数据库的模型空间
    AcDbObjectId CCreateEnt::PostToModelSpace(AcDbEntity* pEnt)
    {
                    AcDbBlockTable *pBlockTable;
                    acdbHostApplicationServices()->workingDatabase()
                            ->getBlockTable(pBlockTable, AcDb::kForRead);

                    AcDbBlockTableRecord *pBlockTableRecord;
        pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
            AcDb::kForWrite);

                    AcDbObjectId entId;
```

```
        pBlockTableRecord->appendAcDbEntity(entId, pEnt);

                pBlockTable->close();
        pBlockTableRecord->close();
        pEnt->close();

                return entId;
    }
```

如果要测试上面创建的这些函数，可以在当前项目中注册一个命令 AddLine，其实现函数的代码为：

```
    void ZffCHAP2AddLine()
    {
                AcGePoint3d ptStart(0, 0, 0);
                AcGePoint3d ptEnd(100, 100, 0);
                AcDbObjectId lineId;

                lineId = CCreateEnt::CreateLine(ptStart, ptEnd);
                CModifyEnt::ChangeColor(lineId, 1);
                CModifyEnt::ChangeLayer(lineId, _T("虚线"));
                CModifyEnt::ChangeLinetype(lineId, _T("中心线"));
    }
```

这段代码同样在模型空间创建一条直线，并将其颜色设置为红色，图层设置为“虚线”层，线型为“中心线”。当然，在程序运行之前，要确保“虚线”层和“中心线”线型的存在。

2.3 创建圆

2.3.1 说明

前面已经分析了创建一个图形对象的基本过程，本节开始就要将着眼点放在创建实体的参数上。本节的实例分别用“圆心、半径”、“直径的两个端点”和“三点法”创建圆。

2.3.2 思路

在 ObjectARX 中，AcDbCircle 类用来表示圆。该类有两个构造函数，其形式分别为：

```
AcDbCircle();
AcDbCircle(const AcGePoint3d& cntr, const AcGeVector3d& nrm, double radius);
```

两个构造函数的名称相同，接受不同的参数，这是 C++中函数的重载。重载是 C++提供

的一个很有用的特性，相同功能的函数采用同样的名称，大大减少了程序员的记忆量。

创建一个圆需要三个参数：圆心、半径和圆所在的平面（一般用平面的法向量来表示）。第一个构造函数不接受任何参数，创建一个圆心为（0，0，0）、半径为0的圆，其所在平面法向量为（0，0，1）；第二个构造函数则接受了圆心、圆所在平面法向量和半径三个参数。一般来说，我习惯于在创建实体时直接将其初始化，很少用第一个构造函数。

2.3.3 步骤

（1）在 VC++ 6.0中，选择【File/Open Workspace】菜单项，从弹出的对话框中选择上一节创建的 CreateEnts 项目，打开该项目。注意，直接选择【File/Open】菜单项仅会打开一个文件，而不会打开整个项目的所有文件。工作区（Workspace）中保存了项目所有文件的位置和名称，在打开工作区时就自动加载了项目中所有的文件。

（2）在 CCreateEnt 类中，添加一个新的函数 CreateCircle，该函数直接封装 AcDbCircle 类的构造函数，其声明和实现分别为：

```
// 声明部分（在CreateEnt.h文件中）
static AcDbObjectId CreateCircle(AcGePoint3d ptCenter,
                                 AcGeVector3d vec, double radius);
// 实现部分（在CreateEnt.cpp文件中）
AcDbObjectId CCreateEnt::CreateCircle(AcGePoint3d ptCenter,
                                                    AcGeVector3d vec, double
radius)
{
        AcDbCircle *pCircle = new AcDbCircle(ptCenter, vec, radius);

        // 将实体添加到图形数据库
        AcDbObjectId circleId;
        circleId = CCreateEnt::PostToModelSpace(pCircle);

        return circleId;
}
```

（3）在 CCreateEnt 类中，再添加一个新的函数 CreateCircle（这就使用了函数重载的概念），用于创建位于 XOY 平面上的圆（一般创建的二维图形都是在 XOY 平面上），其声明和实现分别为：

```
// 声明部分
static AcDbObjectId CreateCircle(AcGePoint3d ptCenter, double radius);
// 实现部分
AcDbObjectId CCreateEnt::CreateCircle(AcGePoint3d ptCenter, double radius)
{
        AcGeVector3d vec(0, 0, 1);
```

```
        return CCreateEnt::CreateCircle(ptCenter, vec, radius);
}
```

代码相当简单，区别就在于输入了一个代表 XOY 平面的法向矢量。

（4）在项目中添加一个新类 CCalculation，用于封装计算的相关函数。添加两个重载静态函数 MiddlePoint，用于计算两点连线的中点：

```
AcGePoint2d CCalculation::MiddlePoint(AcGePoint2d pt1, AcGePoint2d pt2)
{
        AcGePoint2d pt;
        pt[X] = (pt1[X] + pt2[X]) / 2;
        pt[Y] = (pt1[Y] + pt2[Y]) / 2;

        return pt;
}

AcGePoint3d CCalculation::MiddlePoint(AcGePoint3d pt1, AcGePoint3d pt2)
{
        AcGePoint3d pt;
        pt[X] = (pt1[X] + pt2[X]) / 2;
        pt[Y] = (pt1[Y] + pt2[Y]) / 2;
        pt[Z] = (pt1[Z] + pt2[Z]) / 2;

        return pt;
}
```

（5）在 CCreateEnt 类中，添加一个两点法创建圆的函数：

```
AcDbObjectId CCreateEnt::CreateCircle(AcGePoint2d pt1, AcGePoint2d pt2)
{
        // 计算圆心和半径
        AcGePoint2d pt = CCalculation::MiddlePoint(pt1, pt2);
        AcGePoint3d ptCenter(pt[X], pt[Y], 0);      // 圆心
        double radius = pt1.distanceTo(pt2) / 2;

        // 创建圆
        return CCreateEnt::CreateCircle(ptCenter, radius);
}
```

上面的代码中，调用 CCalculation::MiddlePoint 函数来获得两点连线的中点，也就是圆心；AcGePoint2d::distanceTo 函数用于计算两点之间的距离。

（6）添加三点法创建圆的函数。以前在用 VBA 编程时，我曾经用纯数学的方法写过一个三点法画圆的函数，转化到 ObjectARX 程序中来就是下面的形式：

```
AcDbObjectId CCreateEnt::CreateCircle(AcGePoint2d pt1, AcGePoint2d pt2,
```

```
                                                    AcGePoint2d pt3)
    {
        // 使用数学方法
        double xysm, xyse, xy;
        AcGePoint3d ptCenter;
        double radius;

        xy = pow(pt1[X], 2) + pow(pt1[Y], 2);
        xyse = xy - pow(pt3[X], 2) - pow(pt3[Y], 2);
        xysm = xy - pow(pt2[X], 2) - pow(pt2[Y], 2);
        xy = (pt1[X] - pt2[X]) * (pt1[Y] - pt3[Y]) - (pt1[X] - pt3[X]) * (pt1[Y] -
pt2[Y]);

        // 判断参数有效性
        if (fabs(xy) < 0.000001)
        {
            AfxMessageBox("所输入的参数无法创建圆形！");
            return 0;
        }

        // 获得圆心和半径
        ptCenter[X] = (xysm * (pt1[Y] - pt3[Y]) - xyse * (pt1[Y] - pt2[Y])) / (2 *
xy);
        ptCenter[Y] = (xyse * (pt1[X] - pt2[X]) - xysm * (pt1[X] - pt3[X])) / (2 *
xy);
        ptCenter[Z] = 0;
        radius = sqrt((pt1[X] - ptCenter[X]) * (pt1[X] - ptCenter[X]) +
                      (pt1[Y] - ptCenter[Y]) * (pt1[Y] - ptCenter[Y]));

        if (radius < 0.000001)
        {
            AfxMessageBox("半径过小！");
            return 0;
        }

        return CCreateEnt::CreateCircle(ptCenter, radius);
    }
```

代码中出现了几个 C 语言的标准库函数：pow、fabs 和 sqrt，分别用于求幂函数、获得数的绝对值和开平方。要使用这几个函数，必须在当前文件的开头添加下面的语句：

```
#include <math.h>
```

该函数的算法原理为：

假设圆心坐标为（x，y，z），已知的三点坐标分别为（x_1，y_1，z_1）、（x_2，y_2，z_2）和（x_3，y_3，z_3），可以直接建立的方程是：

$$(x-x_1)^2+(y-y_1)^2=(x-x_2)^2+(y-y_2)^2=(x-x_3)^2+(y-y_3)^2$$

打开括号，消去相同的部分，得到下面的方程：

$$-2x_1x+x_1^{\ 2}-2y_1y+y_1^{\ 2}=-2x_2x+x_2^{\ 2}-2y_2y+y_2^{\ 2}=-2x_3x+x_3^{\ 2}-2y_3y+y_3^{\ 2}$$

整理成二元一次方程组，表示为：

$$\begin{cases}(x_2-x_1)x+(y_2-y_1)y=\left[(x_2^{\ 2}+y_2^{\ 2})-(x_1^{\ 2}+y_1^{\ 2})\right]/2\\(x_3-x_1)x+(y_3-y_1)y=\left[(x_3^{\ 2}+y_3^{\ 2})-(x_1^{\ 2}+y_1^{\ 2})\right]/2\end{cases}$$

根据线性代数中求解齐次方程组的方法，只需求解下面三个行列式的值，然后进行相除的运算即可：

$$\begin{vmatrix}(x_2-x_1)&...&(y_2-y_1)\\(x_3-x_1)&...&(y_3-y_1)\end{vmatrix}\qu、\quad\begin{vmatrix}\left[(x_2^{\ 2}+y_2^{\ 2})-(x_1^{\ 2}+y_1^{\ 2})\right]/2&...&(y_2-y_1)\\\left[(x_3^{\ 2}+y_3^{\ 2})-(x_1^{\ 2}+y_1^{\ 2})\right]/2&...&(y_3-y_1)\end{vmatrix}$$ 和

$$\begin{vmatrix}(x_2-x_1)&...&\left[(x_2^{\ 2}+y_2^{\ 2})-(x_1^{\ 2}+y_1^{\ 2})\right]/2\\(x_3-x_1)&...&\left[(x_3^{\ 2}+y_3^{\ 2})-(x_1^{\ 2}+y_1^{\ 2})\right]/2\end{vmatrix}$$

计算出圆心位置之后，就可以使用计算两点距离的函数，计算出圆形的半径。

（7）使用几何类来实现三点法画圆的函数。在 ObjectARX 中提供了一个以 AcGe 开头的类库（一般称为几何类），用来完成一些计算工作，关于几何类会在详细介绍，这里使用了 AcGeCircArc3d 类来完整需要的工作。与数学方法的函数相比，代码相当简洁：

```
AcDbObjectId CCreateEnt::CreateCircle(AcGePoint2d pt1, AcGePoint2d pt2,
                                      AcGePoint2d pt3)
{
        // 使用几何类
        AcGeCircArc2d geArc(pt1, pt2, pt3);
        AcGePoint3d ptCenter(geArc.center().x, geArc.center().y, 0);
        return CCreateEnt::CreateCircle(ptCenter, geArc.radius());
}
```

AcGeCircArc2d 类能够创建一个几何类的圆弧对象，该对象仅用来计算，不能在图形窗口中显示。AcGeCircArc2d 类有四个构造函数，其中一个构造函数可以根据三个点创建圆弧，这里正是使用该函数创建了几何类的圆弧。要使用该类，必须添加下面的语句：

```
#include "gearc3d.h"
```

构建几何类的圆弧之后，就可以查询该对象的圆心、所在平面、半径等特性，将这些参

数传递到创建圆的函数中，就可以实现三点法创建圆。

2.3.4 效果

（1）在项目中注册一个新命令 AddCircle，对几个创建圆的函数进行测试，该命令的实现函数为：

```
void ZffCHAP2AddCircle()
{
        // “圆心、半径”法创建一个圆
        AcGePoint3d ptCenter(100, 100, 0);
        CCreateEnt::CreateCircle(ptCenter, 20);

        // 两点法创建一个圆
        AcGePoint2d pt1(70, 100);
        AcGePoint2d pt2(130, 100);
        CCreateEnt::CreateCircle(pt1, pt2);

        // 三点法创建一个圆
        pt1.set(60, 100);
        pt2.set(140, 100);
        AcGePoint2d pt3(100, 60);
        CCreateEnt::CreateCircle(pt1, pt2, pt3);
}
```

（2）编译并运行该程序，在AutoCAD 2002 中运行AddCircle命令，就能得到如图 2.9所示的结果。

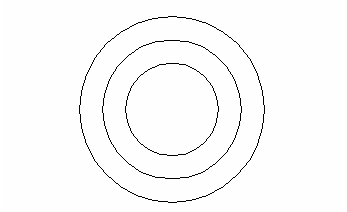

图2.9　命令执行结果

2.3.5 小结

本节开始，我们已经将注意力集中在对象的创建上，本节有两个重点：

- 在封装代码时，注意函数重载的使用。
- 使用几何类，实现三点法创建圆。

2.4 创建圆弧

2.4.1 说明

与上一节创建圆的函数相对应，本节将要实现用“圆心、半径、圆弧所在的平面、起点角度和终点角度”、三点法、“起点、圆心、终点”和“起点、圆心、圆弧角度”几种方法来创建圆弧。

2.4.2 思路

在 ObjectARX 中，AcDbArc 类被用来表示圆弧，该类有三个构造函数：

```
AcDbArc(
                const AcGePoint3d& center,
                double radius,
                double startAngle,
                double endAngle);
AcDbArc(
                const AcGePoint3d& center,
                const AcGeVector3d& normal,
                double radius,
                double startAngle,
                double endAngle);
AcDbArc();
```

第二个构造函数接受最多的参数，因此首先对该函数进行封装，其他几个函数均以封装后的函数为基础。

2.4.3 步骤

（1）打开 CreateEnts 项目，在 CCalculation 类中增加一个新的函数 Pt2dTo3d，其实现代码为：

```
AcGePoint3d CCalculation::Pt2dTo3d(AcGePoint2d pt)
{
                AcGePoint3d ptTemp(pt.x, pt.y, 0);
                return ptTemp;
}
```

（2）在 CCreateEnt 类中添加一个函数 CreateArc，用于向模型空间添加一个圆弧，其实现代码为：

```
AcDbObjectId CCreateEnt::CreateArc(AcGePoint3d ptCenter, AcGeVector3d vec,
                double radius, double startAngle, double endAngle)
```

```
{
        AcDbArc *pArc = new AcDbArc(ptCenter, vec, radius, startAngle,
endAngle);

        AcDbObjectId arcId;
        arcId = CCreateEnt::PostToModelSpace(pArc);

        return arcId;
}
```

（3）添加一个创建位于 XOY 平面上的圆弧的函数，其实现代码为：

```
AcDbObjectId CCreateEnt::CreateArc(AcGePoint2d ptCenter, double radius,
                                   double startAngle, double
endAngle)
{
        AcGeVector3d vec(0, 0, 1);
        return CCreateEnt::CreateArc(CCalculation::Pt2dTo3d(ptCenter),
                vec, radius, startAngle, endAngle);
}
```

（4）三点法创建圆弧，可以使用下面的函数：

```
AcDbObjectId CCreateEnt::CreateArc(AcGePoint2d ptStart, AcGePoint2d ptOnArc,
                                   AcGePoint2d ptEnd)
{
        // 使用几何类获得圆心、半径
        AcGeCircArc2d geArc(ptStart, ptOnArc, ptEnd);
        AcGePoint2d ptCenter = geArc.center();
        double radius = geArc.radius();

        // 计算起始和终止角度
        AcGeVector2d vecStart(ptStart.x - ptCenter.x, ptStart.y - ptCenter.y);
        AcGeVector2d vecEnd(ptEnd.x - ptCenter.x, ptEnd.y - ptCenter.y);
        double startAngle = vecStart.angle();
        double endAngle = vecEnd.angle();

        return CCreateEnt::CreateArc(ptCenter, radius, startAngle, endAngle);
}
```

AcGeVector2d 类用来表示一个二维空间中的矢量，其成员函数 angle 返回该矢量和 X 轴正半轴的角度（用弧度来表示）。

（5）使用“起点、圆心、终点”方法来创建圆弧，可以使用下面的函数：

```
AcDbObjectId CCreateEnt::CreateArcSCE(AcGePoint2d ptStart, AcGePoint2d
```

```
ptCenter,
                                                AcGePoint2d ptEnd)
{
        // 计算半径
        double radius = ptCenter.distanceTo(ptStart);

        // 计算起、终点角度
        AcGeVector2d vecStart(ptStart.x - ptCenter.x, ptStart.y - ptCenter.y);
        AcGeVector2d vecEnd(ptEnd.x - ptCenter.x, ptEnd.y - ptCenter.y);
        double startAngle = vecStart.angle();
        double endAngle = vecEnd.angle();

        // 创建圆弧
        return CCreateEnt::CreateArc(ptCenter, radius, startAngle, endAngle);
}
```

这个函数的名称不再是 CreateArc，而是 CreateArcSCE，这是因为该函数的参数列表、返回值都与三点法的函数相同，无法实现函数的重载，就只能重新定义一个新的函数名称。

（6）使用“起点、圆心、圆弧角度”方法来创建圆弧，可以使用下面的函数：

```
AcDbObjectId CCreateEnt::CreateArc(AcGePoint2d ptStart, AcGePoint2d ptCenter,
                                                double angle)
{
        // 计算半径
        double radius = ptCenter.distanceTo(ptStart);

        // 计算起、终点角度
        AcGeVector2d vecStart(ptStart.x - ptCenter.x, ptStart.y - ptCenter.y);
        double startAngle = vecStart.angle();
        double endAngle = startAngle + angle;

        // 创建圆弧
        return CCreateEnt::CreateArc(ptCenter, radius, startAngle, endAngle);
}
```

2.4.4 效果

（1）在 CCalculation 类中添加一个新的函数 PI，用来计算常量 π 的值，其实现代码为：

```
double CCalculation::PI()
{
        return 4 * atan(1.0);
}
```

atan 是一个 C 语言的库函数，用来计算反正切函数的值，在使用该函数时需要添加下面的语句：

```
#include <math.h>
```

（2）在项目中注册一个新命令 AddArc，该命令的实现函数为：

```
void ZffCHAP2AddArc()
{
        // 创建位于XOY平面上的圆弧
        AcGePoint2d ptCenter(50, 50);
        CCreateEnt::CreateArc(ptCenter, 100 * sqrt(2) / 2,
                5 * CCalculation::PI() / 4, 7 * CCalculation::PI() / 4);

        // 三点法创建圆弧
        AcGePoint2d ptStart(100, 0);
        AcGePoint2d ptOnArc(120, 50);
        AcGePoint2d ptEnd(100, 100);
        CCreateEnt::CreateArc(ptStart, ptOnArc, ptEnd);

        // “起点、圆心、终点”创建圆弧
        ptStart.set(100, 100);
        ptCenter.set(50, 50);
        ptEnd.set(0, 100);
        CCreateEnt::CreateArcSCE(ptStart, ptCenter, ptEnd);

        // “起点、圆心、圆弧角度”创建圆弧
        ptStart.set(0, 100);
        ptCenter.set(50, 50);
        CCreateEnt::CreateArc(ptStart, ptCenter, CCalculation::PI() / 2);
}
```

要成功编译该程序，必须在 CreateEntsCommand.cpp 文件中添加下面的语句：

```
#include "Calculation.h"
```

（3）编译运行程序，在AutoCAD 2002 中运行AddArc命令，能够得到如图 2.10所示的结果。实际上，这个图形由四个圆弧组成。

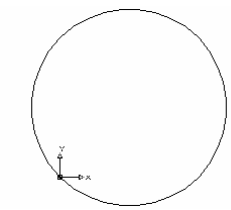

图2.10　命令执行结果

2.4.5 小结

学习本节的实例之后，需要注意下面的问题：

- 使用 AcGeVector2d 类。
- 获得常量 π 的精确值。

2.5 创建多段线

2.5.1 说明

本节介绍若干个函数，分别用于创建轻量多段线、三维多段线、正多边形、矩形、圆（圆环）和圆弧。这里所说的圆形和2.3 节创建的圆不一样，因为本节创建的圆实际上是一条闭合的多段线，可以设置线宽。

2.5.2 思路

ObjectARX 中提供了三种多段线的相关类：AcDbPolyline、AcDb2dPolyline 和 AcDb3dPolyline。其中，利用 AutoCAD 的内部命令可以创建 AcDbPolyline 和 AcDb3dPolyline 类的对象，用 PLINE 命令创建的对象是轻量多段线（AcDbPolyline），用 3DPOLY 命令创建的对象是三维多段线（AcDb3dPolyline）。鉴于 AcDb2dPolyline 目前并不常用，本节不准备详细讨论它。

创建轻量多段线和三维多段线的函数，直接封装 AcDbPolyline 和 AcDb3dPolyline 类的构造函数即可；创建正多边形、矩形、圆形和圆环，实际上都是创建了特殊形状的轻量多段线，创建这些对象的关键都在于顶点和凸度的确定。

2.5.3 步骤

（1）打开 CreateEnts 项目，添加一个函数 CreatePolyline，用于创建轻量多段线：

```
// 声明
static AcDbObjectId CreatePolyline(AcGePoint2dArray points, double width = 0);
// 实现
AcDbObjectId CCreateEnt::CreatePolyline(AcGePoint2dArray points, double width)
{
        int numVertices = points.length();
        AcDbPolyline *pPoly = new AcDbPolyline(numVertices);

        for (int i = 0; i < numVertices; i++)
        {
```

```
            pPoly->addVertexAt(i, points.at(i), 0, width, width);
        }

        AcDbObjectId polyId;
        polyId = CCreateEnt::PostToModelSpace(pPoly);

        return polyId;
    }
```

在 CreatePolyline 函数声明中，为 width 参数设置了默认的实参值 0，这样可以减少该函数被调用时的输入量，如果对此还有疑问，请参阅 C++语法中对默认实参值的说明。注意，不能同时在函数的声明和定义中指定默认参数值，一般可在函数的声明中指定。

创建 AcDbPolyline 对象可以分成两个步骤：创建类的实例和添加顶点。创建类的实例，以多段线的顶点个数作为参数调用 AcDbPolyline::AcDbPolyline 函数；添加顶点时，则使用了 AcDbPolyline::addVertexAt 函数，将每一个顶点添加到多段线中。

addVertexAt 函数定义为：

```
    Acad::ErrorStatus addVertexAt(
        unsigned int index,
        const AcGePoint2d& pt,
        double bulge = 0.,
        double startWidth = -1.,
        double endWidth = -1);
```

index 用来指定插入顶点的索引号（从 0 开始）；pt 指定顶点的位置；bulge 指出要创建的顶点的凸度；startWidth 和 endWidth 指定了从该顶点到下一顶点之间连线的起始和终止线宽，利用该特性可以使用多段线创建一个实心箭头。

凸度是多段线中一个比较难理解的概念，用于指定当前顶点的平滑性，其被定义为：在多段线顶点显示中，选取顶点与下一个顶点形成的弧之间角度的四分之一的正切值。凸度可以用来设置多段线某一段的凸出参数，0 表示直线，1 表示半圆，介于 0～1 之间为劣弧，大于 1 为优弧。在创建圆的函数中，你会进一步学习。

（2）添加一个函数，创建仅包含一条直线的多段线，该函数的实现代码为：

```
    AcDbObjectId CCreateEnt::CreatePolyline(AcGePoint2d ptStart,
        AcGePoint2d ptEnd, double width)
    {
        AcGePoint2dArray points;
        points.append(ptStart);
        points.append(ptEnd);

        return CCreateEnt::CreatePolyline(points, width);
    }
```

AcGePoint2dArray 类的使用非常简单，append 函数用于向数组中添加一个二维点，

removeAt 函数用于从数组中删除指定的元素，length 函数返回数组的长度。

（3）添加一个创建三维多段线的函数：

```
AcDbObjectId CCreateEnt::Create3dPolyline(AcGePoint3dArray points)
{
        AcDb3dPolyline *pPoly3d = new
AcDb3dPolyline(AcDb::k3dSimplePoly, points);

        return CCreateEnt::PostToModelSpace(pPoly3d);
}
```

AcDb3dPolyline 类的构造函数接受了三个参数：第一个参数值 AcDb::k3dSimplePoly 表示创建的多段线是一个未经拟合的标准多段线；第二个参数值指定了创建三维多段线的顶点数组；第三个参数指定是否闭合多段线，这里使用了默认参数值，不在创建多段线时将其闭合。

（4）在 CModifyEnt 类中，添加一个函数 Rotate，按照指定的角度（用弧度值表示）旋转指定的实体，其实现代码为：

```
Acad::ErrorStatus CModifyEnt::Rotate(AcDbObjectId entId,
                                     AcGePoint2d ptBase, double
rotation)
{
        AcGeMatrix3d xform;
        AcGeVector3d vec(0, 0, 1);
        xform.setToRotation(rotation, vec, CCalculation::Pt2dTo3d(ptBase));


        AcDbEntity *pEnt;
       Acad::ErrorStatus es = acdbOpenObject(pEnt, entId, AcDb::kForWrite,
false);
        pEnt->transformBy(xform);
        pEnt->close();

        return es;
}
```

上面的函数使用 AcDbEntity::transformBy 函数对实体进行旋转，该函数能对实体进行比例变换、移动和旋转操作，其输入参数是一个几何变换矩阵（AcGeMatrix3d 类）。AcGeMatrix3d 类用于实体的几何变换非常简单，只要使用 setToScaling、setToRotation 和 setToTranslation 三个函数设置所要进行的变换，然后对所要变换的实体执行 transformBy 函数即可。

有了这个基础，不妨一鼓作气，写出另外两个函数 Move 和 Scale，以备以后使用：

```
Acad::ErrorStatus CModifyEnt::Move(AcDbObjectId entId, AcGePoint3d ptBase,
                AcGePoint3d ptDest)
```

```
{
        // 设置变换矩阵的参数
        AcGeMatrix3d xform;
        AcGeVector3d vec(ptDest.x - ptBase.x, ptDest.y - ptBase.y,
                ptDest.z - ptBase.z);
        xform.setToTranslation(vec);

        AcDbEntity *pEnt;
        Acad::ErrorStatus es = acdbOpenObject(pEnt, entId, AcDb::kForWrite,
false);
        pEnt->transformBy(xform);
        pEnt->close();

        return es;
}
Acad::ErrorStatus CModifyEnt::Scale(AcDbObjectId entId,
                AcGePoint3d ptBase, double scaleFactor)
{
        // 设置变换矩阵的参数
        AcGeMatrix3d xform;
        xform.setToScaling(scaleFactor, ptBase);

        AcDbEntity *pEnt;
        Acad::ErrorStatus es = acdbOpenObject(pEnt, entId, AcDb::kForWrite,
false);
        pEnt->transformBy(xform);
        pEnt->close();

        return es;
}
```

（5）添加创建正多边形的函数。创建正多边形的输入参数为中心、边数、外接圆半径、旋转角度（弧度值）和线宽，其实现代码为：

```
AcDbObjectId CCreateEnt::CreatePolygon(AcGePoint2d ptCenter, int number,
        double radius, double rotation, double width)
{
        AcGePoint2dArray points;
        double angle = 2 * CCalculation::PI() / (double)number;

        for (int i = 0; i < number; i++)
```

```
        {
            AcGePoint2d pt;
            pt.x = ptCenter.x + radius * cos(i * angle);
            pt.y = ptCenter.y + radius * sin(i * angle);

            points.append(pt);
        }

        AcDbObjectId polyId = CCreateEnt::CreatePolyline(points, width);

        // 将其闭合
        AcDbEntity *pEnt;
        acdbOpenAcDbEntity(pEnt, polyId, AcDb::kForWrite);
        AcDbPolyline *pPoly = AcDbPolyline::cast(pEnt);
        if (pPoly != NULL)
        {
            pPoly->setClosed(Adesk::kTrue);
        }
        pEnt->close();

        CModifyEnt::Rotate(polyId, ptCenter, rotation);

        return polyId;
    }
```

创建正多边形，实际上有四个步骤：（a）计算顶点位置；（b）根据顶点位置创建多段线；（c）闭合多段线；（d）旋转多段线。其中，步骤（a）、（b）和（d）不用再多说，步骤（c）是新接触到的内容。前面已经介绍过，使用 acdbOpenAcDbEntity 函数能够得到一个指向 AcDbEntity 对象的指针，但是如何将其转化为指向 AcDbPolyline 的指针，从而利用 AcDbPolylien 类的函数修改其特性呢？注意下面的语句：

```
AcDbPolyline *pPoly = AcDbPolyline::cast(pEnt);
```

cast 是 ObjectARX 所有类的基类——AcRxObject 中实现的一个函数，该函数提供了一种安全的类型转换机制，这里的作用就是从基类指针 pEnt 获得派生类的指针 pPoly。如果你想为这条语句加上一个错误处理，还可以写成这样的形式：

```
if (pEnt->isKindOf(AcDbPolyline::desc()) == Adesk::kTrue)
{
        AcDbPolyline *pPoly = AcDbPolyline::cast(pEnt);

        if (pPoly != NULL)
        {
```

```
                pPoly->setClosed(Adesk::kTrue);
            }
        }
```

isKindOf 和 desc 函数提供了 ObjectARX 中一种常用的动态类型检查机制，这种方法在 ObjectARX 编程中极为常见。

（6）添加用于创建矩形的函数。该函数根据两个角点和线宽来创建矩形，其实现代码为：

```
AcDbObjectId CCreateEnt::CreateRectangle(AcGePoint2d pt1, AcGePoint2d pt2,
                                         double width)
{
    // 提取两个角点的坐标值
    double x1 = pt1.x, x2 = pt2.x;
    double y1 = pt1.y, y2 = pt2.y;

    // 计算矩形的角点
    AcGePoint2d ptLeftBottom(CCalculation::Min(x1, x2),
        CCalculation::Min(y1, y2));
    AcGePoint2d ptRightBottom(CCalculation::Max(x1, x2),
        CCalculation::Min(y1, y2));
    AcGePoint2d ptRightTop(CCalculation::Max(x1, x2),
        CCalculation::Max(y1, y2));
    AcGePoint2d ptLeftTop(CCalculation::Min(x1, x2),
        CCalculation::Max(y1, y2));

    // 创建对应的多段线
    AcDbPolyline *pPoly = new AcDbPolyline(4);
    pPoly->addVertexAt(0, ptLeftBottom, 0, width, width);
    pPoly->addVertexAt(1, ptRightBottom, 0, width, width);
    pPoly->addVertexAt(2, ptRightTop, 0, width, width);
    pPoly->addVertexAt(3, ptLeftTop, 0, width, width);
    pPoly->setClosed(Adesk::kTrue);

    // 将多段线添加到模型空间
    AcDbObjectId polyId;
    polyId = CCreateEnt::PostToModelSpace(pPoly);

    return polyId;
}
```

CCalculation::Min 和 CCalculation::Max 是两个自定义函数，分别用于获得两个数的最大

值额最小值，其实现代码为：

```
double CCalculation::Max(double a, double b)
{
        if (a > b)
        {
                return a;
        }
        else
        {
                return b;
        }
}

double CCalculation::Min(double a, double b)
{
        if (a < b)
        {
                return a;
        }
        else
        {
                return b;
        }
}
```

（7）添加创建圆的函数，输入参数包括圆心、半径和线宽，其实现代码为：

```
AcDbObjectId CCreateEnt::CreatePolyCircle(AcGePoint2d ptCenter,
                                                double radius, double width)
{
        // 计算顶点的位置
        AcGePoint2d pt1, pt2, pt3;
        pt1.x = ptCenter.x + radius;
        pt1.y = ptCenter.y;
        pt2.x = ptCenter.x - radius;
        pt2.y = ptCenter.y;
        pt3.x = ptCenter.x + radius;
        pt3.y = ptCenter.y;

        // 创建多段线
        AcDbPolyline *pPoly = new AcDbPolyline(3);
```

```
        pPoly->addVertexAt(0, pt1, 1, width, width);
        pPoly->addVertexAt(1, pt2, 1, width, width);
        pPoly->addVertexAt(2, pt3, 1, width, width);
        pPoly->setClosed(Adesk::kTrue);

        // 将多段线添加到模型空间
        AcDbObjectId polyId;
        polyId = CCreateEnt::PostToModelSpace(pPoly);

        return polyId;
}
```

其他的东西无须多谈，继续来讨论一下凸度的问题。前面在使用 addVertexAt 函数时，总是指定 0 作为凸度值，上面的函数中则指定了 1 作为凸度。

图图 2.11用来帮助你更好地认识凸度。左侧的半圆起点（顶点索引号为 0）凸度为 1，右侧的半圆起点凸度则为－1，凸度的不同导致圆弧的方向不同。

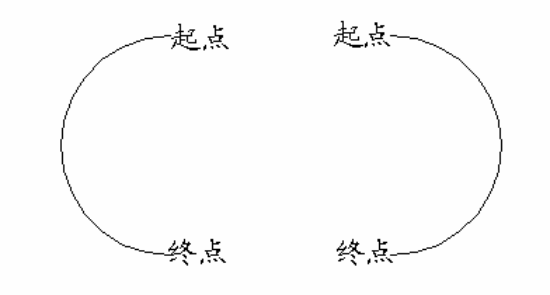

图2.11　凸度的含义

（8）添加用于创建圆弧的函数，输入参数包括圆心、半径、起始角度、终止角度和线宽，其实现代码为：

```
AcDbObjectId CCreateEnt::CreatePolyArc(AcGePoint2d ptCenter, double radius,
        double angleStart, double angleEnd, double width)
{
        // 计算顶点的位置
        AcGePoint2d pt1, pt2;
        pt1.x = ptCenter.x + radius * cos(angleStart);
        pt1.y = ptCenter.y + radius * sin(angleStart);
        pt2.x = ptCenter.x + radius * cos(angleEnd);
        pt2.y = ptCenter.y + radius * sin(angleEnd);

        // 创建多段线
        AcDbPolyline *pPoly = new AcDbPolyline(3);
```

```
                pPoly->addVertexAt(0, pt1, tan((angleEnd - angleStart) / 4), width,
width);
                pPoly->addVertexAt(1, pt2, 0, width, width);

                // 将多段线添加到模型空间
                AcDbObjectId polyId;
                polyId = CCreateEnt::PostToModelSpace(pPoly);

                return polyId;
        }
```

如果能够独立编写这个函数，相信你已经领悟了凸度的含义。

2.5.4 效果

（1）在 CCalculation 类中添加两个函数，用于在角度和弧度之间转换，其实现代码为：

```
// 弧度转化为角度
double CCalculation::RtoG(double angle)
{
        return angle * 180 / CCalculation::PI();
}
// 角度转化为弧度
double CCalculation::GtoR(double angle)
{
        return angle * CCalculation::PI() / 180;
}
```

（2）注册一个新命令 AddPolyline，该命令的实现函数中，添加对本节所有函数进行测试的代码：

```
void ZffCHAP2AddPolyline()
{
        // 创建仅包含一段直线的多段线
        AcGePoint2d ptStart(0, 0), ptEnd(100, 100);
        CCreateEnt::CreatePolyline(ptStart, ptEnd, 1);

        // 创建三维多段线
        AcGePoint3d pt1(0, 0, 0), pt2(100, 0, 0), pt3(100, 100, 0);
        AcGePoint3dArray points;
        points.append(pt1);
        points.append(pt2);
        points.append(pt3);
        CCreateEnt::Create3dPolyline(points);
```

```
        // 创建正多边形
        CCreateEnt::CreatePolygon(AcGePoint2d::kOrigin, 6, 30, 0, 1);

        // 创建矩形
        AcGePoint2d pt(60, 70);
        CCreateEnt::CreateRectangle(pt, ptEnd, 1);

        // 创建圆
        pt.set(50, 50);
        CCreateEnt::CreatePolyCircle(pt, 30, 1);

        // 创建圆弧
        CCreateEnt::CreatePolyArc(pt, 50, CCalculation::GtoR(45),
            CCalculation::GtoR(225), 1);
    }
```

（3）编译运行程序，在AutoCAD 2002 中执行AddPolyline命令，得到如图 2.12所示的结果。

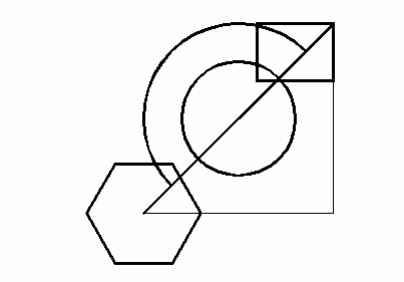

图2.12　测试多段线的创建

2.5.5 小结

学习本节内容之后，下面的几个知识点需要牢固掌握：

- 自定义函数中使用默认参数值；
- 多段线中凸度的含义；
- 使用变换矩阵对实体进行变换；
- ObjectARX 的动态类型检查机制。

2.6 创建椭圆和样条曲线

2.6.1 说明

本节的实例介绍了创建椭圆和样条曲线的方法，创建椭圆的方法包括对 AcDbEllipse 类构造函数的直接封装和根据外接矩形创建椭圆，创建样条曲线的方法仅提供了对 AcDbSpline 类构造函数的封装。

2.6.2 思路

对于创建椭圆对象，可以直接使用AcDbEllipse类的构造函数，给定中心点、所在平面、长轴的一个端点和半径比例来创建椭圆。半径比例是一个用来定义椭圆短轴相对于长轴的比例的参数，半径比例为 1 时椭圆变成圆，帮助系统中给出了一定半径比例时的椭圆形状，如图2.13所示。

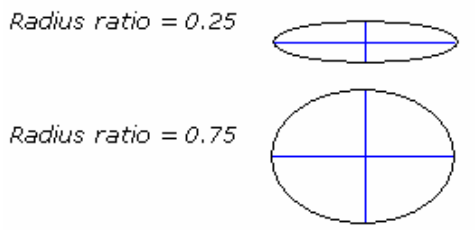

图2.13　半径比例的意义

根据外接矩形创建椭圆，椭圆长、短轴的端点是外接矩形四条边的中点，因此只要外接矩形的角点确定，椭圆的大小和形状就能计算出来。

样条曲线在工程中应用不是很多，因此本节仅对 AcDbSpline 类的构造函数进行封装。

2.6.3 步骤

（1）打开 CreateEnts 项目，添加一个函数 CreateEllipse，用于创建椭圆：

```
AcDbObjectId CCreateEnt::CreateEllipse(AcGePoint3d ptCenter,
                    AcGeVector3d vecNormal, AcGeVector3d majorAxis, double
ratio)
{
        AcDbEllipse *pEllipse = new AcDbEllipse(ptCenter, vecNormal,
            majorAxis, ratio);

        return CCreateEnt::PostToModelSpace(pEllipse);
}
```

AcDbEllipse 类的构造函数定义为：

```
AcDbEllipse( const AcGePoint3d& center,
                const AcGeVector3d& unitNormal,
                const AcGeVector3d& majorAxis,
                double radiusRatio,
                double startAngle = 0.0,
                double endAngle = 6.28318530717958647692);
```

center 指定了椭圆的中心；unitNormal 输入了椭圆所在的平面；majorAxis 输入代表 1/2 长轴的矢量，也就是说该矢量的起点是椭圆的中心，终点是椭圆长轴的一个端点；radiusRatio 给出了椭圆短轴与长轴的长度比例；startAngle 为椭圆的起始角度（弧度）；endAngle 为椭圆的终止角度（弧度）。

通过指定椭圆的起始角度和终止角度，能够方便地创建一个椭圆弧，如果忽略这两个参数，将会创建一个完整的椭圆。

此外，要使用 AcDbEllipse 类，必须在对应的文件中添加对 dbelipse.h 文件的包含。

（2）添加根据外接矩形创建椭圆的函数，输入参数为外接矩形的两个角点，其实现代码为：

```
AcDbObjectId CCreateEnt::CreateEllipse(AcGePoint2d pt1, AcGePoint2d pt2)
{
        // 计算椭圆的中心点
        AcGePoint3d ptCenter;
        ptCenter = CCalculation::MiddlePoint(CCalculation::Pt2dTo3d(pt1),
                CCalculation::Pt2dTo3d(pt2));

        AcGeVector3d vecNormal(0, 0, 1);
        AcGeVector3d majorAxis(fabs(pt1.x - pt2.x) / 2, 0, 0);
        double ratio = fabs((pt1.y - pt2.y) / (pt1.x - pt2.x));

        return CCreateEnt::CreateEllipse(ptCenter, vecNormal, majorAxis,
ratio);
}
```

根据外接矩形的两个角点首先可以计算出椭圆的中心点，然后计算椭圆的长轴端点和短、长轴长度比例，最后调用创建椭圆的函数。其中，fabs 是一个 C 语言的标准库函数，用于获得指定双精度浮点类型变量的绝对值，使用该函数必须添加对 math.h 的包含。

（3）添加创建样条曲线的函数，输入参数包括样条曲线的拟合点、拟合曲线的阶数和允许的拟合误差，其实现代码为：

```
// 声明部分
static AcDbObjectId CreateSpline(const AcGePoint3dArray& points,
                int order = 4, double fitTolerance = 0.0);
// 实现部分
AcDbObjectId CCreateEnt::CreateSpline(const AcGePoint3dArray& points,
```

```
                    int order, double fitTolerance)
{
                    assert (order >= 2 && order <= 26);
                    AcDbSpline *pSpline = new AcDbSpline(points, order, fitTolerance);

                    AcDbObjectId splineId;
                    splineId = CCreateEnt::PostToModelSpace(pSpline);

                    return splineId;
}
```

上面的代码中，需要注意 assert 函数的使用。assert 函数用于判断一个变量或表达式的值是否为 true，如果为 false 则弹出一个错误对话框，并且终止程序的运行。assert 函数适用于判断函数输入参数的有效性，AcDbSpline 类的构造函数要求 order 参数的值在 2～26 之间，因此可以使用 assert 函数来检查其有效性。

如果调用该函数时，order的值不满足要求，系统会弹出图 2.14所示的对话框。在错误提示的对话框中，显示了发生断言错误的具体位置。

图2.14　assert 函数对参数的错误检查

要使用 AcDbSpline 类，必须在文件中包含 dbspline.h 头文件。

（4）添加用于创建样条曲线的函数。与前面的函数相比，多了两个参数，分别用于指定样条曲线起点和终点的切线方向（前面的函数会自动根据各个拟合点的位置计算出起点和终点的切线方向）。其实现代码为：

```
// 声明部分
static AcDbObjectId CreateSpline(const AcGePoint3dArray& points,
                    const AcGeVector3d& startTangent, const AcGeVector3d&
endTangent,
                    int order = 4, double fitTolerance = 0.0);
// 实现部分
AcDbObjectId CCreateEnt::CreateSpline(const AcGePoint3dArray& points,
                     const AcGeVector3d& startTangent, const AcGeVector3d&
```

```
endTangent,
                                int order, double fitTolerance)
        {
                        assert(order >= 2 && order <= 26);
                        AcDbSpline *pSpline = new AcDbSpline(points, startTangent,
endTangent,
                                order, fitTolerance);

                        return CCreateEnt::PostToModelSpace(pSpline);
        }
```

该函数同样适用断言函数 assert 对 order 参数的有效性进行判断，并且封装了 AcDbSpline 类的一个构造函数。

如果你足够留心，在 ObjectARX 的函数中能够发现许多形如“const AcGeVector3d& startTangent”的参数，该参数的含义与“AcGeVector3d startTangent”相同，两者都保证传递的实参不会被修改。那为什么还要采用这种形式？在形参中使用取地址运算符（&），实参在传递给形参时不会被复制，而仅仅被作为引用，但是单独使用该运算符，实参就有可能会在函数中被修改。加上了 const 关键字，则能保证实参不会被修改。如果同时使用取地址运算符和 const 关键字，就能在确保实参不会被修改的情况下减少系统的开销，因此 ObjectARX 库函数中绝大部分的函数都采用了这种形式。

2.6.4 效果

（1）在项目中注册一个 AddEllipse 命令，用于测试创建椭圆的两个函数，其实现代码为：

```
        void ZffCHAP2AddEllipse()
        {
                        // 使用中心点、所在平面、长轴矢量和短长轴比例来创建椭圆
                        AcGeVector3d vecNormal(0, 0, 1);
                        AcGeVector3d majorAxis(40, 0, 0);
                        AcDbObjectId entId;
                        entId = CCreateEnt::CreateEllipse(AcGePoint3d::kOrigin, vecNormal,
                                majorAxis, 0.5);

                        // 使用外接矩形来创建椭圆
                        AcGePoint2d pt1(60, 80), pt2(140, 120);
                        CCreateEnt::CreateEllipse(pt1, pt2);
        }
```

AcGePoint3d::kOrigin 返回点（0，0，0）。

（2）注册一个 AddSpline 命令，用于测试创建样条曲线的两个函数，其实现代码为：

```
        void ZffCHAP2AddSpline()
```

```
{
        // 使用样本点直接创建样条曲线
        AcGePoint3d pt1(0, 0, 0), pt2(10, 30, 0), pt3(60, 80, 0), pt4(100, 100,
0);
        AcGePoint3dArray points;
        points.append(pt1);
        points.append(pt2);
        points.append(pt3);
        points.append(pt4);
        CCreateEnt::CreateSpline(points);

        // 指定起始点和终止点的切线方向，创建样条曲线
        pt2.set(30, 10, 0);
        pt3.set(80, 60, 0);

        points.removeSubArray(0, 3);
        points.append(pt1);
        points.append(pt2);
        points.append(pt3);
        points.append(pt4);

        AcGeVector3d startTangent(5, 1, 0);
        AcGeVector3d endTangent(5, 1, 0);
        CCreateEnt::CreateSpline(points, startTangent, endTangent);
}
```

（3）编译运行程序，执行AddEllipse和AddSpline命令，能够得到如图 2.15所示的结果。

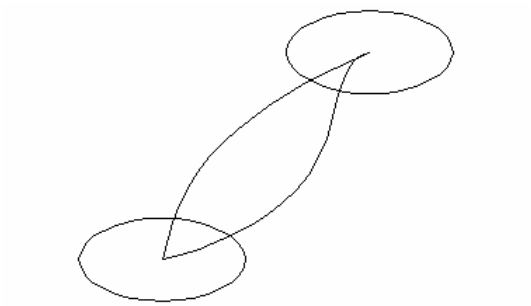

图2.15 创建椭圆和样条曲线

2.6.5 小结

学习本节内容之后，下面的几个知识点需要牢度掌握：

- assert 函数的使用。
- 根据外接矩形创建椭圆的方法。
- 创建椭圆弧的方法。

2.7 创建面域

2.7.1 说明

本节的实例能够提示用户选择所要组成面域的对象，然后在命令行显示创建面域的结果。这与 AutoCAD 中 REGION 命令的操作基本一致。

2.7.2 思路

AcDbRegion 类代表 AutoCAD 中的面域。在 ObjectARX 中创建面域对象非常特别，它一般不利用构造函数来完成对象的创建，而是使用 AcDbRegion 类的一个静态成员函数 createFromCurves 来完成。

createFromCurves 函数的定义为：

```
static Acad::ErrorStatus createFromCurves(
                const AcDbVoidPtrArray& curveSegments,
                AcDbVoidPtrArray& regions);
```

CurveSegments 是一个指向曲线实体的指针数组，用来定义面域的边界，作为面域边界的曲线必须首尾相连；regions 是一个指针数组，返回指向新创建的面域的指针。

在创建面域时，需要注意，作为面域边界的对象必须是 AcDbLine、AcDbArc、AcDbEllipse、AcDbCircle、AcDbSpline、AcDb3dPolyline 或 AcDbPolyline 类的对象。

2.7.3 步骤

打开 CreateEnts 项目，添加一个函数 CreateRegion，用于创建面域：

```
AcDbObjectIdArray CCreateEnt::CreateRegion(const AcDbObjectIdArray& curveIds)
{
        AcDbObjectIdArray regionIds;    // 生成的面域的ID数组
        AcDbVoidPtrArray curves;        // 指向作为面域边界的曲线的指针的
数组
        AcDbVoidPtrArray regions;       // 指向创建的面域对象的指针的数组
        AcDbEntity *pEnt;               // 临时指针，用来关闭边界曲线
        AcDbRegion *pRegion;       // 临时对象，用来将面域添加到模型空间

        // 用curveIds初始化curves
```

```
for (int i = 0; i < curveIds.length(); i++)
{
    acdbOpenAcDbEntity(pEnt, curveIds.at(i), AcDb::kForRead);
    if (pEnt->isKindOf(AcDbCurve::desc()))
    {
        curves.append(static_cast<void*>(pEnt));
    }
}

Acad::ErrorStatus es = AcDbRegion::createFromCurves(curves,
regions);
if (es == Acad::eOk)
{
    // 将生成的面域添加到模型空间
    for (i = 0; i < regions.length(); i++)
    {
        // 将空指针（可指向任何类型）转化为指向面域的指针
        pRegion = static_cast<AcDbRegion*>(regions[i]);
        pRegion->setDatabaseDefaults();
        AcDbObjectId regionId;
        regionId = CCreateEnt::PostToModelSpace(pRegion);
        regionIds.append(regionId);
    }
}
else  // 如果创建不成功，也要删除已经生成的面域
{
    for (i = 0; i < regions.length(); i++)
    {
        delete (AcRxObject*)regions[i];
    }
}

// 关闭作为边界的对象
for (i = 0; i < curves.length(); i++)
{
    pEnt = static_cast<AcDbEntity*>(curves[i]);
    pEnt->close();
}
```

```
        return regionIds;
    }
```

上面的代码中，最容易让人糊涂的就是诸如“static_cast<AcDbEntity*>”这样的东西。static_cast 是 C++的一个运算符，其使用格式为：

static_cast < *type-id* **> (** *expression* **)**

该运算符将 expression 转化成 type-id 类型，但是没有实时的类型检查来保证这种转换的安全性。因此，在 ObjectARX 的类之间相互转化时，你可以用前面介绍的 cast 函数来代替。

使用 createFromCurves 函数创建面域之后，需要自行将面域其添加到模型空间或者删除，并且需要关闭作为边界的曲线对象。

要使用 AcDbRegion 类，必须在文件中包含 dbregion.h 头文件。

2.7.4 效果

（1）注册一个 AddRegion 命令，用于测试创建面域的函数，其实现代码为：

```
    void ZffCHAP2AddRegion()
    {
        // 使用选择集，提示用户选择作为面域边界的对象
        ads_name ss;
        int rt = acedSSGet(NULL, NULL, NULL, NULL, ss);        // 提示用户选择对象
        AcDbObjectIdArray objIds;

        // 根据选择集中的对象构建边界曲线的ID数组
        if (rt == RTNORM)
        {
            long length;
            acedSSLength(ss, &length);          // 获得选择集中的对象个数
            for (int i = 0; i < length; i++)
            {
                ads_name ent;
                acedSSName(ss, i, ent);
                AcDbObjectId objId;
                acdbGetObjectId(objId, ent);

                objIds.append(objId);
            }
        }
        acedSSFree(ss);              // 及时释放选择集

        AcDbObjectIdArray regionIds;
```

```
        regionIds = CCreateEnt::CreateRegion(objIds);

        int number = regionIds.length();
        if (number > 0)
        {
            acutPrintf("\n已经创建%d个面域！", number);
        }
        else
        {
            acutPrintf("\n创建0个面域！");
        }
    }
```

为了实现用户自由在图形窗口中选择多个对象，必须使用选择集。由于选择集的操作在后面还要详细介绍，这里就不想再多费笔墨，让我们把着眼点放在 acutPrintf 函数上。acutPrintf 函数在第一个 Hello,World 程序中就曾介绍，并指出该函数与 C 语言中的 printf 函数非常类似，能够实现格式化的输出。

（2）编译运行程序，在AutoCAD 2002 中，使用LINE命令创建两个三角形，确保这两个三角形的每条直线都首尾相连，如图 2.16所示。

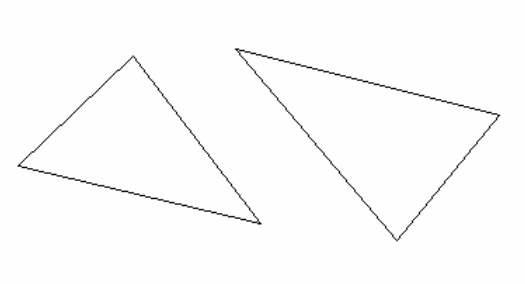

图2.16 创建两个三角形

（3）执行 AddRegion 命令，按照命令行的提示进行操作：

```
命令: addregion
选择对象: 指定对角点: 找到 6 个                    〖选择组成两个三角形的6条直线〗
选择对象:                                          〖按下Enter键完成选择〗
已经创建2个面域！
```

Ok，两个面域已经被创建。

2.7.5 小结

学习本节内容之后，读者要掌握下面的几个知识点：

- 创建面域的整个过程和错误处理。
- 使用 static_cast 运算符进行类型转换。

2.8 创建文字

2.8.1 说明

本节实现了创建文字和多行文字的函数，分别对 AcDbText 类和 AcDbMText 类的相关函数进行封装。文字在 CAD 软件开发中涉及到较多的操作，在后面的章节中还会介绍。

2.8.2 思路

创建文字对象可以使用 AcDbText 类的构造函数，其构造函数定义为：

```
AcDbText( const AcGePoint3d& position,
                const char* text,
                AcDbObjectId style = AcDbObjectId::kNull,
                double height = 0,
                double rotation = 0);
```

position 指定文字的插入点；text 是将要创建的文字对象的内容；style 指定要使用的文字样式的 ID，默认情况下使用 AutoCAD 中缺省的文字样式；height 为文字的高度；rotation 为文字的旋转角度。

AcDbMText 类对应 AutoCAD 中的多行文字，该类的构造函数不接受任何参数，仅能创建一个空的多行文字对象。要创建多行文字，在调用构造函数之后，必须在将其添加到模型空间之前调用 setTextStyle 和 setContents 函数，也可同时调用其它函数来设置多行文字的特性。

2.8.3 步骤

（1）打开 CreateEnts 项目，添加一个函数 CreateText，用于创建文字对象：

```
// 声明部分
static AcDbObjectId CreateText(const AcGePoint3d& ptInsert,
                        const char* text, AcDbObjectId style = AcDbObjectId::kNull,
                        double height = 2.5, double rotation = 0);
// 实现部分
AcDbObjectId CCreateEnt::CreateText(const AcGePoint3d& ptInsert,
                        const char* text, AcDbObjectId style,
                        double height, double rotation)
{
                AcDbText *pText = new AcDbText(ptInsert, text, style, height,
rotation);

                return CCreateEnt::PostToModelSpace(pText);
```

```
}
```

（2）添加一个 CreateMText 函数，用于添加多行文字：

```
// 声明部分
static AcDbObjectId CreateMText(const AcGePoint3d& ptInsert,
                        const char* text, AcDbObjectId style = AcDbObjectId::kNull,
                        double height = 2.5, double width = 10);
// 实现部分
AcDbObjectId CCreateEnt::CreateMText(const AcGePoint3d& ptInsert,
                        const char* text, AcDbObjectId style,
                        double height, double width)
{
        AcDbMText *pMText = new AcDbMText();

        // 设置多行文字的特性
        pMText->setTextStyle(style);
        pMText->setContents(text);
        pMText->setLocation(ptInsert);
        pMText->setTextHeight(height);
        pMText->setWidth(width);
        pMText->setAttachment(AcDbMText::kBottomLeft);

        return CCreateEnt::PostToModelSpace(pMText);
}
```

上面的函数中，在创建多行文字对象之后，修改了它的文字样式、字符串内容、插入点、文字高度、宽度和文字的对齐方式。

要使用 AcDbMText 类，必须在文件中包含 dbmtext.h 头文件。

2.8.4 效果

（1）注册一个 AddText 命令，用于测试本节创建的函数，其实现函数为：

```
void ZffCHAP2AddText()
{
        // 创建单行文字
        AcGePoint3d ptInsert(0, 4, 0);
        CCreateEnt::CreateText(ptInsert, "CAD大观园");

        // 创建多行文字
        ptInsert.set(0, 0, 0);
        CCreateEnt::CreateMText(ptInsert, "http://www.cadhelp.net");
}
```

由于使用文字样式涉及到符号表的知识，本节不再介绍，且等第 4 章详细分解。

（2）编译运行程序，在AutoCAD 2002 中执行AddText命令，能够得到如图 2.17所示的结果。

图2.17　创建文字的结果

（3）设置字体样式。对于不熟悉AutoCAD操作的编程者，到这里可能会感到迷惑，为什么汉字都被“？”代替了？因为当前使用的是西文字体，中文当然没法显示了。在AutoCAD 2002 中，选择【格式 / 文字样式】菜单项，系统会弹出如图 2.18所示的【文字样式】对话框。修改【字体】选项区的设置，选择一种中文字体，单击【应用】按钮，然后单击【关闭】按钮关闭该对话框。

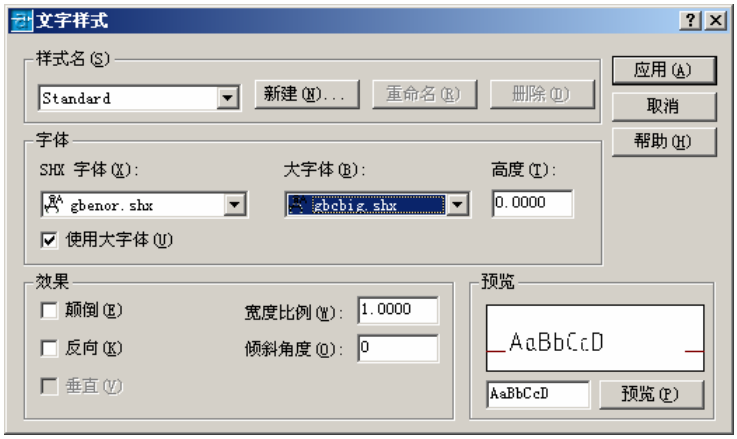

图2.18　设置文字样式

（4）查看生成的文字。设置字体样式之后，得到如图 2.19所示的结果。

图2.19　文字正确显示

2.8.5　小结

学习本节内容之后，读者需要牢固掌握下面的知识点：

- 创建单行文字的方法。
- 创建多行文字的方法。

2.9 创建填充

2.9.1 说明

本节的实例能够提示用户选择所要填充的对象，然后根据用户选择的结果为该区域创建图案填充。

2.9.2 思路

AcDbHatch 类代表 AutoCAD 中的填充对象，该类的构造函数仅创建一个空的填充对象，要想创建填充对象，必须按照下面的步骤进行：

（1）创建一个空的填充对象。
（2）指定填充对象所在的平面。
（3）设置填充对象的关联性；
（4）指定填充图案；
（5）添加填充边界；
（6）显示填充对象；
（7）将其添加到模型空间；
（8）如果是关联性的填充，将填充对象与边界绑定。

2.9.3 步骤

打开 CreateEnts 项目，添加一个 CreateHatch 函数，用于根据一组首尾连接的对象创建填充。该函数的输入参数为组成填充边界的对象 ID 数组、填充图案的名称和填充的关联性，其实现代码为：

```
AcDbObjectId CCreateEnt::CreateHatch(AcDbObjectIdArray objIds,
                                     const char* patName, bool
bAssociative)
{
        Acad::ErrorStatus es;
        AcDbHatch *pHatch = new AcDbHatch();

        // 设置填充平面
        AcGeVector3d normal(0, 0, 1);
        pHatch->setNormal(normal);
```

```
                pHatch->setElevation(0);

                // 设置关联性
                pHatch->setAssociative(bAssociative);

                // 设置填充图案
                pHatch->setPattern(AcDbHatch::kPreDefined, patName);

                // 添加填充边界
                es = pHatch->appendLoop(AcDbHatch::kExternal, objIds);

                // 显示填充对象
                es = pHatch->evaluateHatch();

                // 添加到模型空间
                AcDbObjectId hatchId;
                hatchId = CCreateEnt::PostToModelSpace(pHatch);

                // 如果是关联性的填充，将填充对象与边界绑定，以便使其能获得边界
对象修改的通知
                if (bAssociative)
                {
                    AcDbEntity *pEnt;
                    for (int i = 0; i < objIds.length(); i++)
                    {
                        es = acdbOpenAcDbEntity(pEnt, objIds[i],
AcDb::kForWrite);
                        if (es == Acad::eOk)
                        {
                            // 添加一个永久反应器
                            pEnt->addPersistentReactor(hatchId);
                            pEnt->close();
                        }
                    }
                }

                return hatchId;
            }
```

创建图案填充时最关键的就是定义填充边界，在 ObjectARX 使用 appendLoop 函数来向

填充对象添加边界，该函数定义为：

```
Acad::ErrorStatus appendLoop(
        Adesk::Int32 loopType,
        const AcDbObjectIdArray& dbObjIds);
```

第一个参数指定了边界类型；第二个参数输入一组实体的 ID，用来定义填充边界。此外，还有两种重载形式，但是不常用。

初学者容易忽略在创建填充之后调用 evaluateHatch 函数来显示填充。对于关联性填充，必须使用反应器来绑定填充和边界，这样边界发生变化的时候填充对象才能随之变化。关于反应器的问题，将在**错误！未找到引用源。**详细讨论，这里暂时不作介绍。

要使用 AcDbHatch 类，必须在文件中包含 dbhatch.h 头文件。

2.9.4 效果

（1）在项目中注册一个 AddHatch 命令，其实现函数为：

```
void ZffCHAP2AddHatch()
{
        // 提示用户选择填充边界
        ads_name ss;
        int rt = acedSSGet(NULL, NULL, NULL, NULL, ss);
        AcDbObjectIdArray objIds;

        // 初始化填充边界的ID数组
        if (rt == RTNORM)
        {
            long length;
            acedSSLength(ss, &length);
            for (int i = 0; i < length; i++)
            {
                ads_name ent;
                acedSSName(ss, i, ent);
                AcDbObjectId objId;
                acdbGetObjectId(objId, ent);

                objIds.append(objId);
            }
        }
        acedSSFree(ss);             // 释放选择集

        CCreateEnt::CreateHatch(objIds, "SOLID", true);
}
```

该函数中再次使用了选择集的操作，但是目前还不是最佳时机，暂且不需要了解它的详细使用。

（2）编译运行程序，在AutoCAD 2002 中，调用LINE命令创建一个三角形，如图 2.20所示。

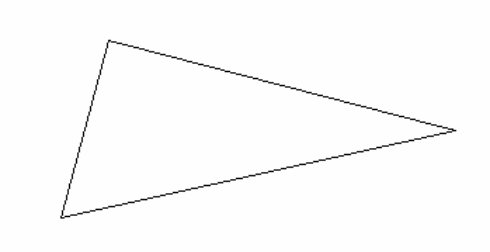

图2.20　创建三角形

（3）执行 AddHatch 命令，按照命令行提示进行操作：

命令: addhatch

选择对象: 指定对角点: 找到 3 个　　〖选择组成三角形的三条直线〗

选择对象:　　〖按下Enter完成选择〗

完成操作后，得到如图 2.21所示的结果。

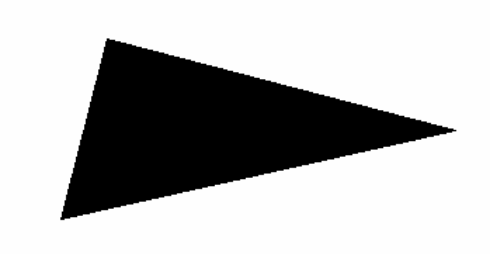

图2.21　创建填充的结果

2.9.5 小结

完成本节的学习，读者需要注意下面的知识点：

- 创建填充的一般步骤。
- 创建关联性和非关联性填充的区别。

2.10 创建尺寸标注

2.10.1 说明

本实例介绍了 12 个用于尺寸标注的函数，包括了转角标注、对齐标注、角度标注、半径标注、直径标注和坐标标注的创建函数。这些函数分别封装了系统提供的一些基本方法，并在实用性上进行了扩充。

2.10.2 思路

AcDbAlignedDimension 类对应的是对齐标注，该类的构造函数接受 5 个参数：第一条尺寸边界线的起点、第二条尺寸边界线的起点、通过尺寸线的一点、标注文字和样式。本节创建一个函数，对该类的构造函数直接进行封装，另外创建一个函数，可以在创建标注时修改标注文字的位置。

设置尺寸文字的替代和位置修改，在实际工程中是很有必要的，对于错综复杂的图形，很多情况下，长度可能会用字母或者其他的汉字替代，而多个按照标准方式放置的标注文字很可能发生重叠，这时候这两个预留的参数就能给你带来极大的方便。

AcDbRotatedDimension 类对应转角标注，该类的构造函数接受 6 个参数：标注的旋转角度、第一条尺寸边界线的起点、第二条尺寸边界线的起点、通过尺寸线的一点、标注文字和样式。本节创建的函数直接对该函数进行封装。

AcDbRadialDimension 类对应半径标注，该类的构造函数需要输入标注曲线的中心点、引线附着的坐标、引线长度、标注文字和样式。本节除了对构造函数进行封装之外，提供了根据圆心、半径、标注尺寸线旋转角度和引线长度来创建半径标注的函数，其关键点就在于根据已知的参数计算出构造函数需要的参数。

AcDbDiametricDimension 类对应直径标注，其构造函数需要输入标注直径的两个端点、引线长度、标注文字和样式。本节除对其构造函数直接封装之外，还提供了根据圆弧的圆心、半径、引线放置角度和引线长度创建标注的方法。

ObjectARX 提供了两个 AcDb2LineAngularDimension 类 AcDb3PointAngularDimension 和来对应角度标注，本节分别对这两个类的构造函数进行了封装。

AcDbOrdinateDimension 类对应坐标标注，其构造函数需要输入是否是 X 轴标注（布尔类型变量）、标注箭头的起始位置、标注箭头的终止位置、标注文字和样式。本节对该函数封装之后，又创建了两个新函数，能够同时创建 X、Y 两个坐标值，并且根据相对坐标修改引线端点的位置。

2.10.3 步骤

（1）打开 CreateEnts 项目，在 CreateEnt 类中添加一个新函数 CreateDimAligned，用于创建对齐标注：

```
// 声明部分
```

```
static AcDbObjectId CreateDimAligned(const AcGePoint3d& pt1,
                        const AcGePoint3d& pt2, const AcGePoint3d& ptLine,
                        const char* dimText = NULL,
                        AcDbObjectId dimStyle = AcDbObjectId::kNull);
// 实现部分
AcDbObjectId CCreateEnt::CreateDimAligned(const AcGePoint3d& pt1,
                const AcGePoint3d& pt2, const AcGePoint3d& ptLine,
                const char* dimText,
                AcDbObjectId dimStyle)
{
                AcDbAlignedDimension *pDim = new AcDbAlignedDimension(pt1, pt2,
                        ptLine, dimText, dimStyle);

                return CCreateEnt::PostToModelSpace(pDim);
}
```

要使用标注的相关类，必须在文件中包含 dbdim.h 头文件。

（2）创建一个重载的函数，允许用户输入 vecOffset 作为标注文字位置的偏移量，其实现代码为：

```
// 声明部分
static AcDbObjectId CreateDimAligned(const AcGePoint3d& pt1,
                        const AcGePoint3d& pt2, const AcGePoint3d& ptLine,
                        const AcGeVector3d& vecOffset = AcGeVector3d::kIdentity,
                        const char* dimText = NULL);
// 实现部分
AcDbObjectId CCreateEnt::CreateDimAligned(const AcGePoint3d& pt1,
                const AcGePoint3d& pt2, const AcGePoint3d& ptLine,
                const AcGeVector3d& vecOffset,
                const char* dimText)
{
                AcDbAlignedDimension *pDim = new AcDbAlignedDimension(pt1, pt2,
                        ptLine, dimText, AcDbObjectId::kNull);

                AcDbObjectId dimensionId;
                dimensionId = CCreateEnt::PostToModelSpace(pDim);

                // 打开已经创建的标注，对文字的位置进行修改
                AcDbEntity *pEnt;
                Acad::ErrorStatus es;
                es = acdbOpenAcDbEntity(pEnt, dimensionId, AcDb::kForWrite);
```

```
                AcDbAlignedDimension *pDimension =
AcDbAlignedDimension::cast(pEnt);

                if (pDimension != NULL)
                {
                    // 移动文字位置前，需先指定尺寸线的变化情况（这里指定为：
                    // 尺寸线不动，在文字和尺寸线之间加箭头）
                    pDimension->setDimtmove(1);
                    // 根据偏移向量修正文字插入点的位置
                    AcGePoint3d ptText = pDimension->textPosition();
                    ptText = ptText + vecOffset;
                    pDimension->setTextPosition(ptText);
                }

                pEnt->close();

                return dimensionId;
        }
```

注意，移动标注文字必须在将其添加到模型空间之后进行。

（3）创建一个新函数 CreateDimRotated，用于添加转角标注：

```
        // 声明部分
        static AcDbObjectId CreateDimRotated(const AcGePoint3d& pt1,
                        const AcGePoint3d& pt2, const AcGePoint3d& ptLine,
                        double rotation, const char* dimText = NULL,
                        AcDbObjectId dimStyle = AcDbObjectId::kNull);
        // 实现部分
        AcDbObjectId CCreateEnt::CreateDimRotated(const AcGePoint3d& pt1,
                        const AcGePoint3d& pt2, const AcGePoint3d& ptLine,
                        double rotation, const char* dimText,
                        AcDbObjectId dimStyle)
        {
                        AcDbRotatedDimension *pDim = new
AcDbRotatedDimension(rotation,
                            pt1, pt2, ptLine, dimText, dimStyle);

                        return CCreateEnt::PostToModelSpace(pDim);
        }
```

（4）创建一个新函数 CreateDimRadial，用于创建半径标注：

```
        // 声明部分
```

```
static AcDbObjectId CreateDimRadial(const AcGePoint3d& ptCenter,
                        const AcGePoint3d& ptChord, double leaderLength,
                        const char* dimText = NULL,
                        AcDbObjectId dimStyle = AcDbObjectId::kNull);
// 实现部分
AcDbObjectId CCreateEnt::CreateDimRadial(const AcGePoint3d& ptCenter,
                const AcGePoint3d& ptChord, double leaderLength,
                const char* dimText,
                AcDbObjectId dimStyle)
{
                AcDbRadialDimension *pDim = new AcDbRadialDimension(ptCenter,
                        ptChord, leaderLength, dimText, dimStyle);

                return CCreateEnt::PostToModelSpace(pDim);
}
```

（5）创建一个重载的函数，用于根据圆心、半径、标注尺寸线的旋转角度和引线长度来创建半径标注：

```
// 声明部分
static AcDbObjectId CreateDimRadial(const AcGePoint3d& ptCenter,
                        double radius, double angle, double leaderLength = 5);
// 实现部分
AcDbObjectId CCreateEnt::CreateDimRadial(const AcGePoint3d& ptCenter,
                double radius, double angle, double leaderLength)
{
                AcGePoint3d ptChord = CCalculation::PolarPoint(ptCenter, angle,
radius);

                return CCreateEnt::CreateDimRadial(ptCenter, ptChord,
leaderLength);
}
```

其中，CCalculation::PolarPoint 是一个自定义函数，能够根据相对极坐标来确定一个点的位置：

```
AcGePoint3d CCalculation::PolarPoint(const AcGePoint3d& pt, double angle,
                        double distance)
{
                ads_point ptForm, ptTo;
                ptForm[X] = pt.x;
                ptForm[Y] = pt.y;
                ptForm[Z] = pt.z;
```

```
                acutPolar(ptForm, angle, distance, ptTo);
                return asPnt3d(ptTo);
        }
```

（6）创建一个函数 CreateDimDiametric，用于创建直径标注：

```
        // 声明部分
        static AcDbObjectId CreateDimDiametric(const AcGePoint3d& ptChord1,
                        const AcGePoint3d& ptChord2, double leaderLength,
                        const char* dimText = NULL,
                        AcDbObjectId dimStyle = AcDbObjectId::kNull);
        // 实现部分
        AcDbObjectId CCreateEnt::CreateDimDiametric(const AcGePoint3d& ptChord1,
                const AcGePoint3d& ptChord2, double leaderLength,
                const char* dimText, AcDbObjectId dimStyle)
        {
                AcDbDiametricDimension *pDim = new
AcDbDiametricDimension(ptChord1,
                        ptChord2, leaderLength, dimText, dimStyle);

                return CCreateEnt::PostToModelSpace(pDim);
        }
```

（7）创建一个重载的函数，根据圆心、半径、标注尺寸线的旋转角度和引线长度来创建直径标注：

```
        // 声明部分
        static AcDbObjectId CreateDimDiametric(const AcGePoint3d& ptCenter,
                        double radius, double angle, double leaderLength = 5);
        // 实现部分
        AcDbObjectId CCreateEnt::CreateDimDiametric(const AcGePoint3d& ptCenter,
                double radius, double angle, double leaderLength)
        {
                // 计算标注通过点的位置
                AcGePoint3d ptChord1, ptChord2;
                ptChord1 = CCalculation::PolarPoint(ptCenter, angle, radius);
                ptChord2 = CCalculation::PolarPoint(ptCenter,
                        angle + CCalculation::PI(), radius);

                return CCreateEnt::CreateDimDiametric(ptChord1, ptChord2,
leaderLength);
        }
```

（8）创建一个函数，用于根据两条直线的关系来创建角度标注：

```
// 声明部分
static AcDbObjectId CreateDim2LineAngular(const AcGePoint3d& ptStart1,
                        const AcGePoint3d& ptEnd1, const AcGePoint3d& ptStart2,
                        const AcGePoint3d& ptEnd2, const AcGePoint3d& ptArc,
                        const char* dimText = NULL,
                        AcDbObjectId dimStyle = AcDbObjectId::kNull);
// 实现部分
AcDbObjectId CCreateEnt::CreateDim2LineAngular(const AcGePoint3d& ptStart1,
                    const AcGePoint3d& ptEnd1, const AcGePoint3d& ptStart2,
                    const AcGePoint3d& ptEnd2, const AcGePoint3d& ptArc,
                    const char* dimText, AcDbObjectId dimStyle)
{
                    AcDb2LineAngularDimension *pDim = new
AcDb2LineAngularDimension(
                        ptStart1, ptEnd1, ptStart2, ptEnd2, ptArc, dimText, dimStyle);

                    return CCreateEnt::PostToModelSpace(pDim);
}
```

（9）创建一个重载的函数，可以根据顶点、起始点、终止点和标注尺寸线通过点来创建角度标注：

```
// 声明部分
static AcDbObjectId CreateDim3PtAngular(const AcGePoint3d& ptCenter,
                        const AcGePoint3d& ptEnd1, const AcGePoint3d& ptEnd2,
                        const AcGePoint3d& ptArc,
                        const char* dimText = NULL,
                        AcDbObjectId dimStyle = AcDbObjectId::kNull);
// 实现部分
AcDbObjectId CCreateEnt::CreateDim3PtAngular(const AcGePoint3d& ptCenter,
                    const AcGePoint3d& ptEnd1, const AcGePoint3d& ptEnd2,
                    const AcGePoint3d& ptArc, const char* dimText,
                    AcDbObjectId dimStyle)
{
                    AcDb3PointAngularDimension *pDim = new
AcDb3PointAngularDimension(
                        ptCenter, ptEnd1, ptEnd2, ptArc, dimText, dimStyle);

                    return CCreateEnt::PostToModelSpace(pDim);
}
```

（10）创建一个函数 CreateDimOrdinate，用于创建坐标标注，直接封装了类的构造函数，其实现代码为：

```
// 声明部分
static AcDbObjectId CreateDimOrdinate(Adesk::Boolean xAxis,
                        const AcGePoint3d& ptStart, const AcGePoint3d& ptEnd,
                        const char* dimText = NULL,
                        AcDbObjectId dimStyle = AcDbObjectId::kNull);
// 实现部分
AcDbObjectId CCreateEnt::CreateDimOrdinate(Adesk::Boolean xAxis,
                const AcGePoint3d& ptStart, const AcGePoint3d& ptEnd,
                const char* dimText,
                AcDbObjectId dimStyle)
{
                AcDbOrdinateDimension *pDim = new
AcDbOrdinateDimension(xAxis,
                        ptStart, ptEnd, dimText, dimStyle);

                return CCreateEnt::PostToModelSpace(pDim);
}
```

（11）创建一个重载的函数，能够同时创建一个点的 X、Y 坐标标注：

```
// 声明部分
static AcDbObjectIdArray CreateDimOrdinate(const AcGePoint3d& ptDef,
                        const AcGePoint3d& ptTextX, const AcGePoint3d& ptTextY);
// 实现部分
AcDbObjectIdArray CCreateEnt::CreateDimOrdinate(const AcGePoint3d& ptDef,
                        const AcGePoint3d& ptTextX, const AcGePoint3d& ptTextY)
{
                AcDbObjectId dimId;
                AcDbObjectIdArray dimIds;

                dimId = CCreateEnt::CreateDimOrdinate(Adesk::kTrue, ptDef,
ptTextX);
                dimIds.append(dimId);
                dimId = CCreateEnt::CreateDimOrdinate(Adesk::kFalse, ptDef,
ptTextY);
                dimIds.append(dimId);

                return dimIds;
}
```

（12）创建一个重载的函数，能够根据点的偏移位置来创建坐标标注：

```
// 声明部分
static AcDbObjectIdArray CreateDimOrdinate(const AcGePoint3d& ptDef,
                    const AcGeVector3d& vecOffsetX, const AcGeVector3d&
vecOffsetY);
// 实现部分
AcDbObjectIdArray CCreateEnt::CreateDimOrdinate(const AcGePoint3d& ptDef,
                    const AcGeVector3d& vecOffsetX, const AcGeVector3d&
vecOffsetY)
{
        AcGePoint3d ptTextX = ptDef + vecOffsetX;
        AcGePoint3d ptTextY = ptDef + vecOffsetY;

        return CCreateEnt::CreateDimOrdinate(ptDef, ptTextX, ptTextY);
}
```

2.10.4 说明

（1）在 CCalculation 类中增加一个函数 RelativePoint，用于根据相对直角坐标来计算一个点的位置：

```
AcGePoint3d CCalculation::RelativePoint(const AcGePoint3d& pt,
                    double x, double y)
{
        AcGePoint3d ptReturn(pt.x + x, pt.y + y, pt.z);
        return ptReturn;
}
```

（2）在项目中注册一个命令 AddDimension，用于测试本节创建的函数：

```
void ZffCHAP2AddDimension()
{
        // 指定起始点位置
        AcGePoint3d pt1(200, 160, 0);
        AcGePoint3d pt2= CCalculation::RelativePoint(pt1, -40, 0);
        AcGePoint3d pt3 = CCalculation::PolarPoint(pt2,
            7 * CCalculation::PI() / 6, 20);
        AcGePoint3d pt4 = CCalculation::RelativePoint(pt3, 6, -10);
        AcGePoint3d pt5 = CCalculation::RelativePoint(pt1, 0, -20);

        // 绘制外轮廓线
        CCreateEnt::CreateLine(pt1, pt2);
        CCreateEnt::CreateLine(pt2, pt3);
```

```
CCreateEnt::CreateLine(pt3, pt4);
CCreateEnt::CreateLine(pt4, pt5);
CCreateEnt::CreateLine(pt5, pt1);

// 绘制圆形
AcGePoint3d ptCenter1, ptCenter2;
ptCenter1 = CCalculation::RelativePoint(pt3, 16, 0);
ptCenter2 = CCalculation::RelativePoint(ptCenter1, 25, 0);
CCreateEnt::CreateCircle(ptCenter1, 3);
CCreateEnt::CreateCircle(ptCenter2, 4);

AcGePoint3d ptTemp1, ptTemp2;
// 水平标注
ptTemp1 = CCalculation::RelativePoint(pt1, -20, 3);
CCreateEnt::CreateDimRotated(pt1, pt2, ptTemp1, 0);

// 垂直标注
ptTemp1 = CCalculation::RelativePoint(pt1, 4, 10);
CCreateEnt::CreateDimRotated(pt1, pt5, ptTemp1,
        CCalculation::PI() / 2);

// 转角标注
ptTemp1 = CCalculation::RelativePoint(pt3, -3, -6);
CCreateEnt::CreateDimRotated(pt3, pt4, ptTemp1,
        7 * CCalculation::PI() / 4);

// 对齐标注
ptTemp1 = CCalculation::RelativePoint(pt2, -3, 4);
CCreateEnt::CreateDimAligned(pt2, pt3, ptTemp1,
        AcGeVector3d(4, 10, 0), "new position");

// 角度标注
ptTemp1 = CCalculation::RelativePoint(pt5, -5, 5);
CCreateEnt::CreateDim3PtAngular(pt5, pt1, pt4, ptTemp1);

// 半径标注
ptTemp1 = CCalculation::PolarPoint(ptCenter1,
        CCalculation::PI() / 4, 3);
CCreateEnt::CreateDimRadial(ptCenter1, ptTemp1, -3);
```

```
        // 直径标注
        ptTemp1 = CCalculation::PolarPoint(ptCenter2,
                CCalculation::PI() / 4, 4);
        ptTemp2 = CCalculation::PolarPoint(ptCenter2,
                CCalculation::PI() / 4, -4);
        CCreateEnt::CreateDimDiametric(ptTemp1, ptTemp2, 0);

        // 坐标标注
        CCreateEnt::CreateDimOrdinate(ptCenter2, AcGeVector3d(0, -10, 0),
                AcGeVector3d(10, 0, 0));
    }
```

（3）编译运行程序，在AutoCAD 2002 中执行AddDimension命令，能够得到如图 2.22所示的结果。

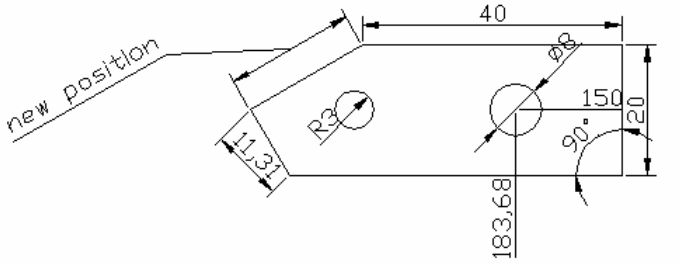

图2.22　程序运行结果

2.10.5 小结

学习本节内容之后，读者需要掌握下面的知识点：

- 尺寸文本的替代：所有的函数都可直接指定 dimText 参数实现尺寸文本的替代。在创建标注之后，还可以使用 AcDbDimension::setDimensionText 函数来设置尺寸文本的内容。
- 尺寸文本的移动：使用 setTextPosition 函数可以实现尺寸文本的移动，在移动文字之前，最好使用 setDimtmove 函数设置文字和尺寸线移动时的关系，例如设置在移动文字时，不移动尺寸线，但是在两者之间加引线。
- 尺寸标注的关联性：无法直接使用 AcDbDimension 及其派生类创建关联性的尺寸标注，要实现尺寸标注的关联性，必须使用反应器。关于反应器的具体知识，请参见**错误！未找到引用源。**的内容。

2.11 创建和编辑对象时的动态拖动技术

2.11.1 说明

本节所介绍的实例，允许用户创建一个圆弧长度的标注，在标注过程中可以选择标注文本的位置，其使用方法与 AutoCAD 中提供的标注操作完全一致。

另一个实例则模拟了 MOVE 命令，不过仅能移动文字（AcDbText 类）对象。

2.11.2 思路

对于 AutoCAD 内建图形对象，要在创建或编辑中实现动态拖动的效果，只能考虑使用 acedGrRead 函数。该函数能够跟踪鼠标的移动，其定义为：

```
int acedGrRead(
        int track,
        int * type,
        struct resbuf * result);
```

track 指定了该函数的控制位，可以选择如下的位（可以将位值相加，设置多种条件）：

- Bit 0（输入 1 作为参数值）：返回拖动的坐标。如果用户设置了这个位，当用户移动鼠标或其它顶点设备时，acedGrRead 将 type 设置为 5，result 设置为（X，Y）坐标。
- Bit 1（输入 2 作为参数值）：返回所有的关键值，包括函数和光标的所有代码。
- Bit 2（输入 4 作为参数值）：根据 type 参数的值来控制光标的显示，共有三种选项。type 设置为 0 时，显示一般的十字丝；type 设置为 1 时，不显示光标或十字丝；type 设置为 2 时，显示选择实体的小方框。
- Bit 3 （输入 8 作为参数值）：不显示错误，当用户按下 Ctrl+C 快捷键的时候，控制台暂停消息。

type 参数返回输入设备及其种类，result 参数被设置为从用户获得的参数。

使用函数的一般结构是：

```
int track = 1, type;        // 控制位和输入设备类型
struct resbuf result;       // 保存鼠标拖动时的动态坐标
while (track > 0)
{
    acedGrRead(track, &type, &result); // 追踪光标移动
    ptText[X] = result.resval.rpoint[X];        // 获得用户输入点的位
置

    ptText[Y] = result.resval.rpoint[Y];

    // 使用获得的坐标
    {
```

```
                ……
            }

            if (type == 3)                          // 如果用户按下了
鼠标左键，跳出循环
            {
                track = 0;
            }
        }
```

下面的两个例子中，能够发现具体的使用方法。

2.11.3 步骤

（1）使用ObjectARX向导创建一个新工程，命名为GrRead，使用MFC。将前面的CreateEnts项目中的CreateEnt.h、CreateEnt.cpp、ModifyEnt.h、ModifyEnt.cpp、Calculation.h和Calculation.cpp六个文件复制到当前项目的目录下，然后在VC++ 6.0 中，选择【Project/Add To Project/Files】菜单项，系统会弹出如图 2.23所示的对话框，将上述的六个文件添加到项目中。

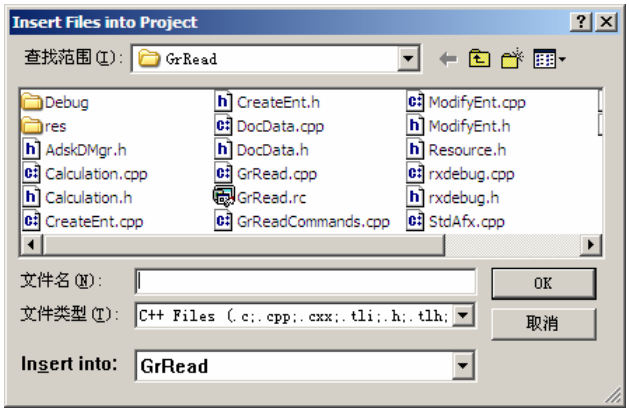

图2.23　将已经存在的文件添加到当前项目中

实际上，上面所进行的操作是在 VC 中引入已经存在的类的方法，也就是首先复制文件到当前项目中，然后将文件添加到项目中，此时在 ClassView 视图中自动添加三个类：CCreateEnt、CCalculation 和 CModifyEnt。

（2）注册一个新命令AddDimension，其各项参数设置如图 2.24所示。

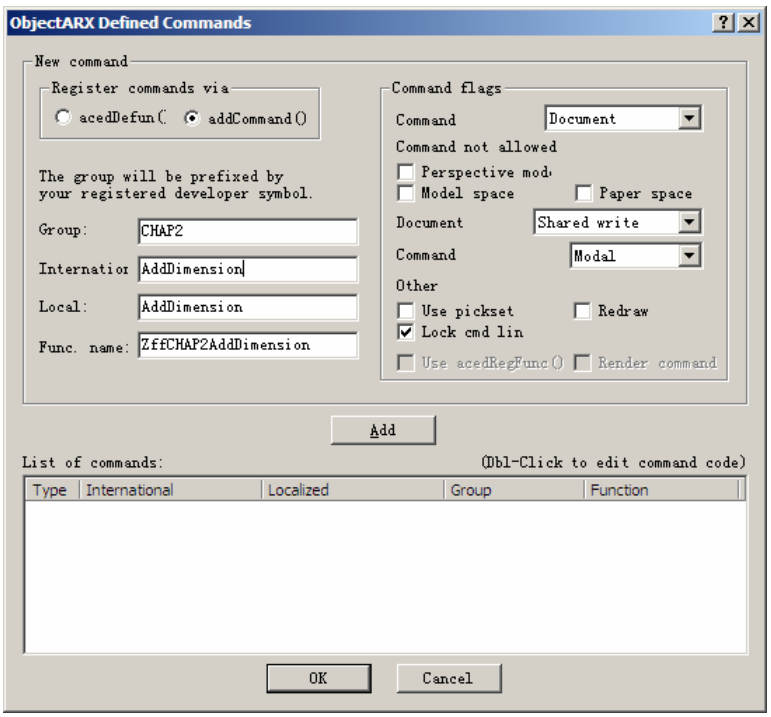

图2.24 注册 AddDimension 命令

（2）AddDimension 函数可以与用户交互，创建圆弧长度的标注，其实现函数为：

```
void ZffCHAP2AddDimension()
{
        // 提示用户选择圆弧
        ads_name en;
        ads_point pt;
        if (acedEntSel("选择所要标注的圆弧:", en, pt) != RTNORM)
                return;

        // 获得选择对象的指针
        AcDbObjectId arcId;
        Acad::ErrorStatus es = acdbGetObjectId(arcId, en);
        if (es != Acad::eOk)
                return;
        AcDbEntity *pEnt;
        es = acdbOpenAcDbEntity(pEnt, arcId, AcDb::kForRead);

        // 判断选择的对象是否是圆弧
        if (!pEnt->isKindOf(AcDbArc::desc()))
        {
                pEnt->close();
```

```
        return;
    }
    AcDbArc *pArc = AcDbArc::cast(pEnt);

    // 获得圆弧的特征点位置
    AcGePoint3d ptCenter, ptStart, ptEnd, ptMiddle;
    ptCenter = pArc->center();
    es = pArc->getStartPoint(ptStart);
    es = pArc->getEndPoint(ptEnd);
    double length;
    es = pArc->getDistAtPoint(ptEnd, length);
    es = pArc->getPointAtDist(length / 2, ptMiddle);
    pEnt->close();

    // 创建三点角度标注
    CString strLength;
    strLength.Format("%.2f", length);
    AcDbObjectId dimId;
    dimId = CCreateEnt::CreateDim3PtAngular(ptCenter, ptStart,
        ptEnd, ptMiddle, strLength);

    // 拖动鼠标改变标注文字的位置
    AcGePoint3d ptText;
    int track = 1, type; //track=1
    struct resbuf result;      // 保存鼠标拖动时的动态坐标
    while (track > 0)
    {
        acedGrRead(track, &type, &result); // 追踪光标移动
        ptText[X] = result.resval.rpoint[X];
        ptText[Y] = result.resval.rpoint[Y];

        // 设置拖动位置为标注文本的插入点
        acdbOpenAcDbEntity(pEnt, dimId, AcDb::kForWrite);
        if (pEnt->isKindOf(AcDb3PointAngularDimension::desc()))
        {
            AcDb3PointAngularDimension *pDim;
            pDim = AcDb3PointAngularDimension::cast(pEnt);

            if (pDim != NULL)
```

```
                {
                    pDim->setTextPosition(ptText);
                }
            }
            pEnt->close();

            if (type == 3)      // 如果用户按下了鼠标左键
            {
                track = 0;
            }
        }
    }
```

提示用户选择一个实体使用 acedEntSel 函数，该函数能够获得所选择实体的 ads_name，然后使用 acdbGetObjectId 函数可以将 ads_name 转化为 AcDbObjectId，进而通过 acdbOpenAcDbEntity 函数获得对象的指针，访问其各种特性。

获取圆弧的圆心、起点和终点都有直接调用的函数，但是却无法直接获得圆弧的中点。AcDbCurve 类的 getDistAtPoint 函数用于获得曲线上某一点到起点的距离，getPointAtDist 函数用于获得曲线上距离起点一定长度的点，因此可以用下面的方法获得圆弧的中点：使用 getDistAtPoint 函数获得圆弧的长度；使用 getPointAtDist 函数根据长度获得圆弧的中点。

当用户在图形窗口中移动鼠标时，acedGrRead 函数会获得当前光标位置的坐标，使用 acdbOpenAcDbEntity 函数获得指向新创建的标注对象的指针，修改标注文本的位置之后，记得关闭该对象。

（3）注册一个 MoveText 命令，用于根据用户的选择移动文字，其实现代码为：

```
void ZffCHAP2MoveText()
{
    ads_name entName;
    ads_point ptPick, ptBase;
    if (acedEntSel("\n选择所要移动的文字：", entName, ptPick) !=
RTNORM)
    {
        return;
    }

    AcDbObjectId txtId;
    AcDbText *pText = NULL;
    AcDbEntity *pEnt = NULL;
    Acad::ErrorStatus es = acdbGetObjectId(txtId, entName);
    if (es != Acad::eOk)
    {
```

```
            return;
        }

        AcGePoint3d ptInsertOld(0, 0, 0);
        acdbOpenObject(pEnt, txtId, AcDb::kForWrite);
        if (pEnt->isKindOf(AcDbText::desc()))
        {
            pText = AcDbText::cast(pEnt);

            if (pText != NULL)
            {
                ptInsertOld = pText->position();
            }
        }
        pEnt->close();

        if (acedGetPoint(NULL, "\n选择基点：", ptBase) != RTNORM)
        {
            return;
        }
        acedPrompt("\n选择第二点：");

        AcGePoint3d ptInsertNew(0, 0, 0);
        AcGePoint3d ptPick3d = asPnt3d(ptBase);

        // 鼠标拖动部分
        int track = 1, type; //track=1
        struct resbuf result;      // 保存鼠标拖动时的动态坐标
        while (track > 0)
        {
            acedGrRead(track, &type, &result); // 追踪光标移动
            ptInsertNew[X] = result.resval.rpoint[X] - ptPick3d[X] +
ptInsertOld[X];
            ptInsertNew[Y] = result.resval.rpoint[Y] - ptPick3d[Y] +
ptInsertOld[Y];

            // 设置拖动位置为直线的终点坐标
            acdbOpenObject(pEnt, txtId, AcDb::kForWrite);
            if (pEnt->isKindOf(AcDbText::desc()))
```

```
                {
                    pText = AcDbText::cast(pEnt);

                    if (pText != NULL)
                    {
                        pText->setPosition(ptInsertNew);
                    }
                }
                pEnt->close();

                if (type == 3)      // 如果用户按下了鼠标左键
                {
                    track = 0;
                }
            }
        }
```

2.11.4 效果

（1）编译运行程序，在AutoCAD 2002 中，使用ARC命令创建一段圆弧，使用TEXT命令创建一个文字对象，如图 2.25所示。

图2.25 创建圆弧和文字

（2）执行AddDimension命令，选择所要标注的圆弧，然后拖动鼠标，确定标注文字之后单击左键完成标注的创建，如图 2.26所示。

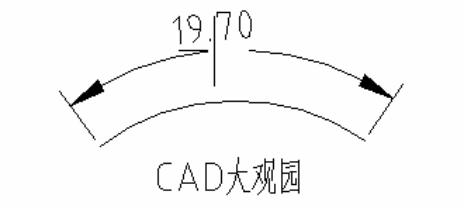

图2.26 拖动效果

（3）执行 MoveText 命令，按照命令提示进行操作：

命令: movetext
选择所要移动的文字：
选择基点：
选择第二点：

现在，相信你已经可以根据自己的需要来使用 acedGrRead 函数了。

2.11.5 小结

学习本章知识之后，读者需要掌握下面的知识点：

- 提示用户选择一个实体的方法。
- 创建和编辑对象时使用 acedGrRead 函数实现拖动效果。

2.12 获得某一图层上所有的直线

2.12.1 说明

本节介绍的实例，能将当前图形中“测试”图层上所有直线对象的颜色变为红色。该实例演示了块表记录遍历器的使用，为获取图形中某一类具有相同特征的实体提供了一种方法。

2.12.2 思路

ObjectARX 提供了一种称为遍历器的类，用来遍历（逐个访问）某一集合中所有的对象，譬如，遍历当前图形中所有的图层、实体等。

下面的代码显示了块表记录遍历器的使用要点：

```
// 创建块表记录遍历器
AcDbBlockTableRecordIterator *pItr;          // 块表记录遍历器
pBlkTblRcd->newIterator(pItr);
AcDbEntity *pEnt;                            // 遍历的临时实体指针
for (pItr->start(); !pItr->done(); pItr->step())
{
    // 利用遍历器获得每一个实体
    pItr->getEntity(pEnt, AcDb::kForWrite);

    // 对pEnt所指向的实体进行各种编辑
    ……

    // 注意需要关闭实体
    pEnt->close();
```

```
}
delete pItr;          // 遍历器使用完毕之后一定要删除!
```

块表记录遍历器的使用非常简单，简单的说就是三个步骤：创建遍历器；使用遍历器遍历实体；删除遍历器。

2.12.3 步骤

在 VC++ 6.0中，使用 ObjectARX 向导创建一个名为 GetEntsOnLayer 的项目，注册一个名为 GetEnts 的命令，其实现函数为：

```
void ZffCHAP2GetEnts()
{
    // 判断是否存在名称为“测试”的图层
    AcDbLayerTable *pLayerTbl;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pLayerTbl, AcDb::kForRead);
    if (!pLayerTbl->has("测试"))
    {
        acutPrintf("\n当前图形中未包含\"测试\"图层!");
        pLayerTbl->close();
        return;
    }
    AcDbObjectId layerId;                    // “测试”图层的ID
    pLayerTbl->getAt("测试", layerId);
    pLayerTbl->close();

    // 获得当前数据库的块表
    AcDbBlockTable *pBlkTbl;
    acdbHostApplicationServices()->workingDatabase()
        ->getBlockTable(pBlkTbl, AcDb::kForRead);

    // 获得模型空间的块表记录
    AcDbBlockTableRecord *pBlkTblRcd;
    pBlkTbl->getAt(ACDB_MODEL_SPACE, pBlkTblRcd,
AcDb::kForRead);
    pBlkTbl->close();

    // 创建块表记录遍历器
    AcDbBlockTableRecordIterator *pItr;          // 块表记录遍历器
    pBlkTblRcd->newIterator(pItr);
    AcDbEntity *pEnt;                            // 遍历的临时实体指针
```

```
        for (pItr->start(); !pItr->done(); pItr->step())
        {
            // 利用遍历器获得每一个实体
            pItr->getEntity(pEnt, AcDb::kForWrite);

            // 是否在“测试”图层上
            if (pEnt->layerId() == layerId)
            {
                // 是否是直线
                AcDbLine *pLine = AcDbLine::cast(pEnt);
                if (pLine != NULL)
                {
                pLine->setColorIndex(1);          // 将直线的颜色修改为红色
                }
            }

            // 注意需要关闭实体
            pEnt->close();
        }
        delete pItr;                    // 遍历器使用完毕之后一定要删除！
        pBlkTblRcd->close();
    }
```

在获得指定名称的层表记录之前，要判断当前图形中是否包含指定的图层，可以使用 AcDbLayerTable::has 函数来实现。如果块表中包含与指定名称相同的层表记录，该函数返回 true，否则返回 false。

使用遍历器时必须注意，遍历一个集合对象要使用其对应的遍历器，例如本节的实例遍历块表记录就是用了块表记录遍历器。声明块表记录遍历器指针之后，还要使用 newIterator 函数创建当前图形模型空间块表记录的遍历器。遍历器在使用完毕后一定要删除，否则就会引起 AutoCAD 的错误退出。

判断实体是否在指定的图层上，可以使用两种方法：

- 使用 AcDbEntity 类的 layer 函数获得实体所在图层的名称，然后与指定图层的名称进行比较，例如（acutDelString 函数需要添加对 acutmem.h 头文件的包含）：

```
char *layerName = pEnt->layer();
if (strcmp(layerName, "测试") == 0)
{
    // 执行需要的操作
    ……
}
acutDelString(layerName);              // 释放layer函数返回的
```

字符串所占用的内存

- 使用 AcDbEntity 类的 layerId 函数获得实体所在图层的 ID，然后与指定图层的 ID 进行比较。本节的实例中使用的就是这样方法。

提示：如何知道什么时候需要释放字符串的内存？实际上，在ObjectARX的帮助文档中，凡是需要特殊注意的函数，都已经给出了相关的说明，如图 2.27所示。

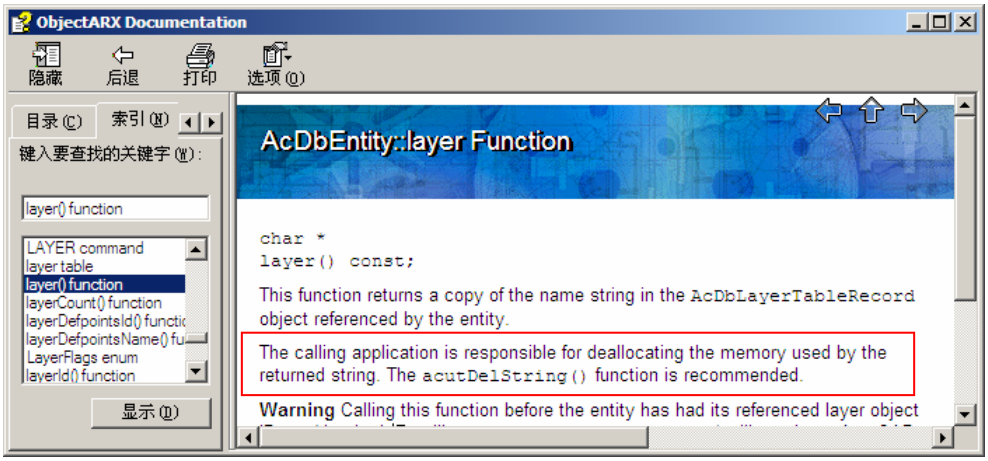

图2.27 函数的特别注意之处

2.12.4 效果

编译运行程序，在 AutoCAD 2002中创建几个新图层，其中包含一个名为“测试”的图层，然后创建若干个图形对象，将其分别放置在不同的图层中。

执行 GetEnts 命令，能够发现，“测试”图层中所有的直线变为红色，与程序设计的初衷完全一致。

2.12.5 小结

学习本节实例之后，读者可以在此基础上进行扩展：

- 获得当前图形中所有半径小于 5 的圆。
- 获得当前图形中所有颜色为红色的实体。
- 获得当前图形中位于“测试”图层上的实体的集合。

本节的实例仅讨论了如何对满足条件的实体进行编辑，但是并未介绍如何获得满足特定条件的实体的集合，实际上实现这一点已经非常简单了。一般来说，用 AcDbObjectIdArray 类作为对象的集合比较合适，可以在使用遍历器之前声明一个 AcDbObjectIdArray 类的对象：

```
AcDbObjectIdArray entIds;
```

而在判断实体满足一定条件之后，可以将实体的 ID 添加到数组中：

```
entIds.append(pEnt->objectId());
```

这样，在删除遍历器之后，就得到了包含所有满足条件的实体的 ID 数组。

2.13 创建三维实体

2.13.1 说明

ObjectARX 中提供了三类创建三维实体的方法：创建标准形状的实体、拉伸面域创建实体和旋转面域创建实体。本节通过创建长方体、圆锥体、弹簧和一个皮带轮实体来介绍这三种方法。

仅能用三类方法来创建实体，所能得到的实体的形状还是很有限的，布尔运算提供了实体的交、并、补运算，能够根据已有的实体“组合”出复杂形状的实体。本节通过对两个长方体进行布尔运算来介绍其使用方法。

2.13.2 思路

在 ObjectARX 中，AcDb3dSolid 类用于代表 AutoCAD 中的三维实体，提供了创建和合并实体的一些方法，与使用 AutoCAD 命令来创建实体类似。但是，ACIS 实体才是实体真正的几何表示，AcDb3dSolid 类只是 ACIS 实体的容器和接口，该类中并没有提供直接操作 ACIS 实体边、顶点和面的方法。要遍历 ACIS 实体中隐含（无法直接访问子实体）的边、面和顶点，必须使用 ObjectARX 开发包中的 BREP 应用程序开发接口（API）。

1. 长方体

AcDb3dSolid 类仅提供了一个不包含任何参数的构造函数，用于创建一个“空”实体，在构造 AcDb3dSolid 对象之后，必须使用其成员函数来完成实体的创建。createBox 函数用于创建长方体，其定义为：

```
virtual Acad::ErrorStatus createBox(double xLen, double yLen, double zLen);
```

其中，xLen、yLen 和 zLen 分别指定长方体的长、宽和高，该函数将会创建一个中心位于世界坐标系原点的长方体，并且其长、宽、高分别平行于世界坐标系的 X、Y 和 Z 轴。

2. 圆锥体

ObjectARX 中并未直接提供创建圆锥体的方法，而是将其包含在创建平截头体（圆柱体和圆锥体都是其中的一种）的函数 createFrustum 中，该函数被定义为：

```
virtual Acad::ErrorStatus createFrustum(
                    double height,
                    double xRadius,
                    double yRadius,
                    double topXRadius);
```

其中，height 表示平截头体的高度，xRadius 表示底面在 X 轴方向的半径，yRadius 表示底面在 Y 轴方向的半径，topXRadius 表示顶面在 X 轴方向的半径。如果要创建一个圆锥体，就可以将 topXRadius 参数设置为 0，并且保证 xRadius 和 yRadius 的值相等。

3. 拉伸面域创建实体

AcDb3dSolid 类中的 extrudeAlongPath 函数用于拉伸面域创建一个实体，其定义为：

```
virtual Acad::ErrorStatus extrudeAlongPath(
                const AcDbRegion* region,
                const AcDbCurve* path);
```

其中，region 是一个指向作为拉伸截面的面域的指针，path 是一个指向作为拉伸路径的曲线的指针。值得注意的是，在执行 extrudeAlongPath 函数时，region 和 path 都必须是模型空间中的实体，否则会引发一个异常。

AcDb3dSolid 类的另一个函数 extrude 用于沿面域所在平面的法线方向拉伸面域创建新的实体，并且可以指定拉伸时的斜切角度。

4. 旋转面域创建实体

AcDb3dSolid 类中的 revolve 函数用于绕给定的轴线旋转面域而生成实体，其定义为：

```
virtual Acad::ErrorStatus revolve(
                const AcDbRegion* region,
                const AcGePoint3d& axisPoint,
                const AcGeVector3d& axisDir,
                double angleOfRevolution);
```

其中，region 是一个指向作为旋转截面的面域的指针；axisPoint 指定旋转轴线上的一点；axisDir 指定了旋转轴的方向，和 axisPoint 共同确定旋转轴的具体位置；angleOfRevolution 指定旋转面域的角度（弧度值来表示）。

5. 布尔运算

AcDb3dSolid 类中的 booleanOper 函数用于在两个实体之间执行布尔运算，其定义为：

```
virtual Acad::ErrorStatus booleanOper(
                AcDb::BoolOperType operation,
                AcDb3dSolid* solid);
```

其中，operation 指定了进行布尔运算的方式，包括 AcDb::kBoolUnite（并集）、AcDb::kBoolIntersect（交集）和 AcDb::kBoolSubtract（差集）三种类型；solid 是一个指向布尔运算的另一个实体的指针。

2.13.3 步骤

（1）在 VC++ 6.0 中，使用 ObjectARX 向导创建一个新工程，工程名称为 Create3dSolid。注册一个命令 AddBox，用于创建长方体，其实现函数为：

```
void ZffCHAP2AddBox()
{
                AcDb3dSolid *pSolid = new AcDb3dSolid();
                Acad::ErrorStatus es = pSolid->createBox(40, 50, 30);
```

```
        if (es != Acad::eOk)
        {
            acedAlert("创建长方体失败!");
            delete pSolid;
            return;
        }

        // 使用几何变换矩阵移动长方体
        AcGeMatrix3d xform;
        AcGeVector3d vec(100, 100, 100);
        xform.setToTranslation(vec);
        pSolid->transformBy(xform);

        // 将长方体添加到模型空间
        PostToModelSpace(pSolid);
    }
```

上面的代码中，将 createBox 函数的返回值与 Acad::eOk 比较，判断该函数是否正确执行，如果没有正确执行就进行相应的错误处理。值得一提的是，ObjectARX 中许多函数的返回值类型都是 Acad::ErrorStatus，这是一个重要的错误处理手段，对于程序的调试也非常有帮助。

PostToModelSpace 函数是一个自定义函数，用于将指定的实体添加到当前图形的模型空间，在前面的几节已经多次使用此函数，其定义为：

```
AcDbObjectId PostToModelSpace(AcDbEntity* pEnt)
{
    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
            ->getBlockTable(pBlockTable, AcDb::kForRead);

    AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
            AcDb::kForWrite);

    AcDbObjectId entId;
    pBlockTableRecord->appendAcDbEntity(entId, pEnt);

    pBlockTable->close();
    pBlockTableRecord->close();
    pEnt->close();

    return entId;
```

```
}
```

注意，PostToModelSpace 函数的定义部分必须放在 ZffCHAP2AddBox 函数之前才能被正确编译，这是 C 语言的要求，如果是在 C++中用类的成员函数来实现，就不必考虑这一点。

（2）注册一个命令 AddCylinder，用于创建圆锥体，其实现函数为：

```
void ZffCHAP2AddCylinder()
{
        // 创建特定参数的圆柱体（实际上是一个圆锥体）
        AcDb3dSolid *pSolid = new AcDb3dSolid();
        pSolid->createFrustum(30, 10, 10, 0);

        // 将圆锥体添加到模型空间
        PostToModelSpace(pSolid);
}
```

要保证创建的实体是圆锥，就必须将 createFrustum 函数的第 4 个参数值设置为 0。

（3）注册一个命令 AddSpire，用于在图形窗口中创建一个三维弹簧模型，其实现函数为：

```
void ZffCHAP2AddSpire()
{
        // 指定创建螺旋线的参数
        double radius, deltaVertical; // 半径和每一周在垂直方向的增量
        double number, segment;              // 螺旋线的旋转圈数和组成一圈的分段数

        radius = 30, deltaVertical = 12;
        number = 5, segment = 30;

        // 计算点的个数和角度间隔
        int n = number * segment;   // 点的个数实际上是n+1
        double angle = 8 * atan(1) / segment; // 两点之间的旋转角度

        // 计算控制点的坐标
        AcGePoint3dArray points;   // 控制点坐标数组
        for (int i = 0; i < n+1; i++)
        {
                AcGePoint3d vertex;
                vertex[X] = radius * cos(8 * i * atan(1) / segment);
                vertex[Y] = radius * sin(8 * i * atan(1) / segment);
                vertex[Z] = i * deltaVertical / segment;
                points.append(vertex);
```

```
                    }

                    // 创建螺旋线路径
                    AcDb3dPolyline *p3dPoly = new
AcDb3dPolyline(AcDb::k3dSimplePoly,
                            points);

                    // 将路径添加到模型空间
                    AcDbObjectId spireId = PostToModelSpace(p3dPoly);

                    // 创建一个圆作为拉伸的截面
                    AcGeVector3d vec(0, 1, 0);              // 圆所在平面的法矢量
                    AcGePoint3d ptCenter(30, 0, 0);         // 圆心位置与半径的大小有关
                    AcDbCircle *pCircle = new AcDbCircle(ptCenter, vec, 3);
                    AcDbObjectId circleId = PostToModelSpace(pCircle);

                    // 根据圆创建一个面域
                    AcDbObjectIdArray boundaryIds, regionIds;
                    boundaryIds.append(circleId);
                    regionIds = CreateRegion(boundaryIds);

                    // 打开拉伸截面和拉伸路径
                    AcDbRegion *pRegion;
                    acdbOpenObject(pRegion, regionIds.at(0), AcDb::kForRead);
                    AcDb3dPolyline *pPoly;
                    acdbOpenObject(pPoly, spireId, AcDb::kForRead);

                    // 进行拉伸操作
                    AcDb3dSolid *pSolid = new AcDb3dSolid();
                    pSolid->extrudeAlongPath(pRegion, pPoly);
                    PostToModelSpace(pSolid);

                    pPoly->close();
                    pRegion->close();
        }
```

首先创建一个三维螺旋线作为拉伸路径，使用小段的三维多段线来模拟三维螺旋线，segment 参数指定了每一圈螺旋线的分段数。需要注意的一点是，需要计算的顶点个数实际上是 number*segment+1，因为对于一条连续的曲线来说，间隔数总是比节点数少 1。

atan(1)用来代替 π，其值为 π/4，因此 4*atan(1)的值就是 π。这种替代比直接用一个常

量“const double PI = 3.1415926;”要精确得多，在本书中一般会用其替代π。

代码中使用了两个 ObjectARX 中的数组类型：AcGePoint3dArray 和 AcDbObjectIdArray。这两个类均派生自 AcArray 类，AcArray 类的使用非常简单，length 函数用于获得数组对象的元素个数；at 函数用于获得指定索引的元素的值；append 函数用于向数组添加元素；[]运算符的作用与 at 函数相同。

其中，CreateRegion 是一个自定义函数，用于根据给定的边界创建面域，该函数与 2.7 节的同名函数完全相同，其实现代码为：

```
AcDbObjectIdArray CreateRegion(const AcDbObjectIdArray& curveIds)
{
        AcDbObjectIdArray regionIds;    // 生成的面域的ID数组
        AcDbVoidPtrArray curves;        // 指向作为面域边界的曲线的指针的
数组
        AcDbVoidPtrArray regions;       // 指向创建的面域对象的指针的数组
        AcDbEntity *pEnt;               // 临时指针，用来关闭边界曲线
        AcDbRegion *pRegion;       // 临时对象，用来将面域添加到模型空间

        // 用curveIds初始化curves
        for (int i = 0; i < curveIds.length(); i++)
        {
            acdbOpenAcDbEntity(pEnt, curveIds.at(i), AcDb::kForRead);
            if (pEnt->isKindOf(AcDbCurve::desc()))
            {
                curves.append(static_cast<void*>(pEnt));
            }
        }

        Acad::ErrorStatus es = AcDbRegion::createFromCurves(curves,
regions);
        if (es == Acad::eOk)
        {
            // 将生成的面域添加到模型空间
            for (i = 0; i < regions.length(); i++)
            {
                // 将空指针（可指向任何类型）转化为指向面域的指针
                pRegion = static_cast<AcDbRegion*>(regions[i]);
                pRegion->setDatabaseDefaults();
                AcDbObjectId regionId;
                regionId = PostToModelSpace(pRegion);
                regionIds.append(regionId);
```

```
            }
        }
        else   // 如果创建不成功，也要删除已经生成的面域
        {
            for (i = 0; i < regions.length(); i++)
            {
                delete (AcRxObject*)regions[i];
            }
        }

        // 关闭作为边界的对象
        for (i = 0; i < curves.length(); i++)
        {
            pEnt = static_cast<AcDbEntity*>(curves[i]);
            pEnt->close();
        }

        return regionIds;
    }
```

（4）注册一个命令 RevolveEnt，用于旋转面域生成一个实体，其实现函数为：

```
    void ZffCHAP2RevolveEnt()
    {
        // 设置顶点的坐标
        AcGePoint3d vertex[5];
        vertex[0] = AcGePoint3d(15, 0, 0);
        vertex[1] = AcGePoint3d(45, 0, 0);
        vertex[2] = AcGePoint3d(35, 9, 0);
        vertex[3] = AcGePoint3d(41, 18, 0);
        vertex[4] = AcGePoint3d(15, 18, 0);
        AcGePoint3dArray points;
        for (int i = 0; i <= 4; i++)
        {
            points.append(vertex[i]);
        }

        // 创建作为旋转截面的多段线
        AcDb3dPolyline *p3dPoly = new
AcDb3dPolyline(AcDb::k3dSimplePoly,
            points, true);
```

```
        AcDbObjectId polyId = PostToModelSpace(p3dPoly);

        // 将闭合的多段线转化成面域
        AcDbObjectIdArray boundaryIds, regionIds;
        boundaryIds.append(polyId);
        regionIds = CreateRegion(boundaryIds);

        // 进行旋转操作
        AcDbRegion *pRegion;
        Acad::ErrorStatus es = acdbOpenObject(pRegion, regionIds.at(0),
AcDb::kForRead);
        AcDb3dSolid *pSolid = new AcDb3dSolid();
        es = pSolid->revolve(pRegion, AcGePoint3d::kOrigin,
              AcGeVector3d(0, 1, 0), 8 * atan(1));
        PostToModelSpace(pSolid);

        pRegion->close();
    }
```

在编程过程中，如果需要使用世界坐标系的原点作为参数，可以直接使用AcGePoint3d::kOrigin；如果需要一个矢量作为参数，可以直接使用 AcGeVector3d(0, 1, 0)作为参数；如果需要一个点作为参数，同样可以使用 AcGePoint3d(10, 10, 0)作为参数。

（5）注册一个命令 Boolean，用于对两个长方体进行布尔运算，其实现函数为：

```
    void ZffCHAP2Boolean()
    {
        // 创建两个长方体
        AcDb3dSolid *pSolid1 = new AcDb3dSolid();
        pSolid1->createBox(40, 50, 30);
        AcDb3dSolid *pSolid2 = new AcDb3dSolid();
        pSolid2->createBox(40, 50, 30);

        // 使用几何变换矩阵移动长方体
        AcGeMatrix3d xform;
        AcGeVector3d vec(20, 25, 15);
        xform.setToTranslation(vec);
        pSolid1->transformBy(xform);

        // 将长方体添加到模型空间
        AcDbObjectId solidId1 = PostToModelSpace(pSolid1);
        AcDbObjectId solidId2 = PostToModelSpace(pSolid2);
```

```
                    // 进行布尔运算，生成新的实体
                    acdbOpenObject(pSolid1, solidId1, AcDb::kForWrite);
                    acdbOpenObject(pSolid2, solidId2, AcDb::kForWrite);

                    Acad::ErrorStatus es = pSolid1->booleanOper(AcDb::kBoolUnite,
pSolid2);
                    assert(pSolid2->isNull());
                    pSolid2->erase(); // 将其删除

                    pSolid2->close(); // 删除之后还是需要关闭该实体
                    pSolid1->close();
        }
```

上面的代码中，assert 是一个断言函数（不是一个宏），用于检测一个表达式的值，如果表达式的值为 FALSE，系统会弹出一个提示对话框，并且终止程序的执行。这对于需要参数验证的函数，提供了一个非常好的机制，一般可以在函数开始执行时对参数的有效性进行验证。

booleanOper 函数不会自动删除 pSolid2，因此需要手工删除 pSolid2，删除之后仍然需要使用 close 函数将其关闭，也就是将该对象的控制权交给 AutoCAD。为什么在删除实体之后仍需要关闭实体，并将对象控制权交给 AutoCAD？因为执行 erase 函数之后系统不会立即从图形中将指定的实体删除，而是将其“删除位”打开，在 AutoCAD 保存图形时该实体将不会再被保存，下一次重新打开图形时该实体就不再存在。

提示：为什么 erase 函数执行时不是立即删除实体？这是为了提供 Undo（撤消）的操作，只要未关闭图形，一个被删除的实体就可以使用 erase(kfalse)来撤消删除。

2.13.4 效果

编译连接程序，启动AutoCAD 2002，加载生成的ARX文件，分别执行AddBox、AddCylinder、AddSpire、RevolveEnt和Boolean命令，能够得到图 2.28所示的结果（为了便于观察，已将各个图形移动到适当的位置）。

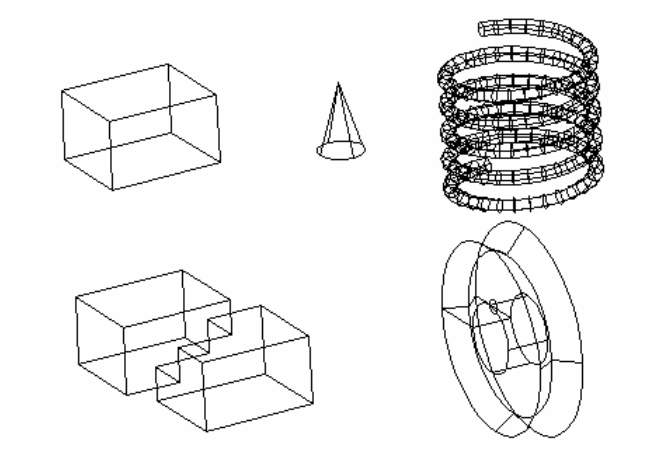

图2.28 执行命令得到的图形

2.13.5 小结

学习本节的实例之后，读者可以自行练习 createSphere、createTorus、createWedge 和 extrude 等函数，并结合布尔运算，创建形状更为复杂的三维实体。

2.14 利用 Transform 实现复制、移动等操作

2.14.1 说明

创建 ObjectARX 的应用程序经常要和用户进行交互，很可能会对实体执行一些移动、旋转、镜像等操作，这种情况下可以直接使用 acedCommand 函数调用相关的 AutoCAD 内部命令，也可以使用 AcDbEntity 类的 transformBy 函数，对实体进行相应的变换操作。

实际上使用第一种方法在某些特定的情况下有一个致命的问题。由于 AutoCAD 内部命令和 ObjectARX 注册的命令在不同的线程中执行，因此有可能导致使用 acedCommand 函数的命令执行过程发生错乱，下面可以用一个例子来证明。

使用 ObjectARX 向导创建一个新工程，其名称设置为 TransformEnt，使用 ObjectARX 嵌入工具栏的【ObjectARX commands】按钮注册一个名称为 TestCommand 的新命令，该命令的实现代码为：

```
void ZffTransformTestCommand()
{
        acedCommand(RTSTR, "_move",
                RTNONE);

        AfxMessageBox("先移动实体还是先弹出对话框?");
```

```
}
```

在 AutoCAD 2002 中执行此命令，你会发现在进行移动操作之前，对话框就已经弹出来了，如果在一个线程内部决不会发生，因此可以证明 AutoCAD 内部命令和 ObjectARX 注册的命令不在同一个线程内部执行。基于这一点，对实体进行变换操作最好的方法还是使用 transformBy 函数。

有一点值得注意，**在 ObjectARX 编程中尽量不要使用中文的标点符号，特别是不能在 acutPrintf、acedGetPoint 等向命令窗口输出文本的函数中使用**，某些情况下使用中文标点符号会导致命令窗口中本应显示的中文提示变成乱码。

本节的程序介绍使用 transformBy 函数对实体进行几何变换，其中的一些方法使用 ObjectARX 中提供的一些基于 COM（组件对象模型）的全局函数来实现。

2.14.2 思路

1. 使用transformBy函数进行几何变换

transformBy 是 AcDbEntity 类的一个成员函数，该函数使用一个 AcGeMatrix3d 参数对实体进行相应的几何变换，其原型为：

```
virtual Acad::ErrorStatus transformBy(const AcGeMatrix3d& xform);
```

所有 AcDbEntity 的派生类都实现了这个虚函数，因此所有的实体都可以使用这种方法进行几何变换。

AcGeMatrix3d是一个几何类，用于表示一个四维矩阵，其基本形式如图 2.29所示。

$$\begin{matrix} a_{00} & a_{01} & a_{02} & t_0 \\ a_{10} & a_{11} & a_{12} & t_1 \\ a_{20} & a_{21} & a_{22} & t_2 \\ 0 & 0 & 0 & 1 \end{matrix}$$

图2.29　变换矩阵的内容

尽管可以像普通的矩阵那样修改变换矩阵中的元素，但是多种变换叠加的矩阵参数计算起来非常麻烦，所幸 AcGeMatrix3d 提供了一些很有用的成员函数：

- setToTranslation：生成一个移动对象的矩阵。
- setToRotation：生成旋转矩阵。
- setToScaling：生成比例缩放矩阵。
- setToMirroring：生成镜像矩阵。

2. 复制实体

AcDbObject 类拥有一个 clone 函数，能否生成一个调用者的克隆对象，并返回指向克隆对象的指针。由于所有实体对应的类都间接继承于 AcDbObject 类（AcDbEntity 类从 AcDbObject 类继承），因此所有实体都可以用这种方法进行克隆。

clone 函数仅仅会生成对象的一个克隆，对于实体对象来说，这样还没有完成复制操作的全部。在创建实体时我们已经了解到，创建实体仅仅是第一个步骤，还必须把它添加到模型空间中才能被显示出来，对于克隆得到的实体同样需要这样做。

3. 使用AcAxXXX全局函数

在 ObjectARX 中有一系列 AcAx 开头的全局函数，这些函数通过 COM 的方式来让 AutoCAD 完成一些操作，这一节我们能使用 AcAxMove、AcAxRotate 和 AcAxScaleEntity 函数分别完成移动、旋转和缩放实体的操作。

如果从来没有接触过 COM 相关的知识，你可能对 AcAxMove 的定义感到迷惑：

```
HRESULT AXAUTOEXP AcAxMove(
                AcDbObjectId& objId,
                VARIANT fromPoint,
                VARIANT toPoint);
```

VARIANT 是什么东西？VARIANT 是在 COM 中使用的一种特殊数据类型，为什么不能直接使用 C 和 C++中的数据类型？因为 COM 是微软用于解决组件之间数据交换的一种技术，COM 对象建立在二进制可执行代码级的基础上，并由此来实现多种语言开发的组件对象可以进行交互。

因此开发 COM 所使用的数据类型是独立于特定语言的，我们开发 COM 客户程序（调用 COM 对象的程序）也必须使用一些 COM 所规定的数据类型。这里不会再多介绍关于 COM 的任何知识，你也不要希望在这里弄明白什么是 COM，只需要知道 AcAxMove、AcAxRotate 和 AcAxScaleEntity 函数是基于 COM 的。

2.14.3 步骤

（1）使用 ObjectARX 向导创建一个新工程，命名为 TransformEnt。选择【Insert/New Class】菜单项，在系统弹出的【New Class】对话框中，选择【Class Type】为【Generic Class】，输入 CTransUtil 作为类的名称，单击【OK】按钮完成类的创建。

（2）在 CTransUtil 类中添加一个函数 Move，使用 transformBy 函数对实体进行移动：

```
Acad::ErrorStatus CTransUtil::Move(AcDbObjectId entId, const AcGePoint3d
&ptFrom, const AcGePoint3d &ptTo)
{
        // 构建用于实现移动实体的矩阵
        AcGeVector3d vec(ptTo[X] - ptFrom[X], ptTo[Y] - ptFrom[X],
                ptTo[Z] - ptFrom[Z]);
        AcGeMatrix3d mat;
        mat.setToTranslation(vec);

        AcDbEntity *pEnt = NULL;
        Acad::ErrorStatus es = acdbOpenObject(pEnt, entId,
AcDb::kForWrite);
```

```
        if (es != Acad::eOk)
            return es;
        es = pEnt->transformBy(mat);
        pEnt->close();

        return es;
    }
```

使用 transformBy 函数移动实体的关键在于构建符合要求的 AcGeMatrix3d 对象，AcGeMatrix3d 类的 setToTranslation 函数用于完成这个功能，它所接受的参数是一个三维矢量，因此先根据移动的基点和目的点构建了一个 AcGeVector3d 对象。

（3）添加一个函数 AcMove，使用 AcAxMove 全局函数完成对实体的移动：

```
BOOL CTransUtil::AcMove(AcDbObjectId entId, const AcGePoint3d &ptFrom,
                                    const AcGePoint3d &ptTo)
{
        // 将AcGePoint3d类型的点坐标进行类型转换
        VARIANT *pvaFrom = Point3dToVARIANT(ptFrom);
        VARIANT *pvaTo = Point3dToVARIANT(ptTo);

        BOOL bRet = SUCCEEDED(AcAxMove(entId, *pvaFrom, *pvaTo));
        delete pvaFrom;
        delete pvaTo;

        return bRet;
}
```

Point3dToVARIANT 是一个自定义函数，能根据 AcGePoint3d 对象生成一个 VARIANT 类型的对象，并返回指向该对象的指针。Point3dToVARIANT 函数的定义为：

```
static VARIANT* Point3dToVARIANT(const AcGePoint3d &point)
{
        COleSafeArray *psa = new COleSafeArray();
        DOUBLE dblValues[] = {point[X], point[Y], point[Z]};
        psa->CreateOneDim(VT_R8, 3, dblValues);

        return (LPVARIANT)(*psa);
}
```

static 关键字限制了 Point3dToVARIANT 函数的作用域，在该文件之外不能使用这个函数。在 C++编程中，应该给函数尽量小的作用域，全局函数更要尽量避免出现，使用 static 函数限制函数的作用域是一个好的方法。

由于 VARIANT 和 COleSafeArray 都无法直接作为函数的返回值，它们的复制操作必须使用专门的函数，而不像 C++中的标准类型直接可以复制拷贝，也就是说，下面的函数返回

值总是无效的。

```
static COleSafeArray Point3dToVARIANT(const AcGePoint3d &point)
{
        COleSafeArray sa;
        DOUBLE dblValues[] = {point[X], point[Y], point[Z]};
        sa.CreateOneDim(VT_R8, 3, dblValues);

        return sa;
}
```

由于在 Point3dToVARIANT 函数中使用 new 关键字动态分配了内存，那么调用它的函数必须释放返回值的内存空间，在 AcMove 函数中就分别释放了 pvaFrom 和 pvaTo 所指向变量的空间。

（4）添加一个函数 Copy，能够生成给定 ID 的实体的一个克隆，并且按照 ptFrom 和 ptTo 生成的矢量移动生成的克隆对象，其实现代码为：

```
BOOL CTransUtil::Copy(AcDbObjectId entId, const AcGePoint3d &ptFrom,
                                          const AcGePoint3d &ptTo)
{
        AcDbEntity *pEnt = NULL;
        if (acdbOpenObject(pEnt, entId, AcDb::kForRead) != Acad::eOk)
                return FALSE;

        AcDbEntity *pCopyEnt = AcDbEntity::cast(pEnt->clone());
        AcDbObjectId copyEntId;
        if (pCopyEnt)
                copyEntId = PostToModelSpace(pCopyEnt);

        Move(copyEntId, ptFrom, ptTo);

        return TRUE;
}
```

clone 函数能够生成调用者的一个克隆，该函数是 AcDbObject 类的一个成员函数，其原型为：

```
virtual AcRxObject* clone() const;
```

由于函数的返回值不是 AcDbEntity 类型的指针，因此必须通过一个很常用的方法进行实体指针的升级：

```
AcDbEntity *pCopyEnt = AcDbEntity::cast(pEnt->clone());
```

Copy 函数中，PostToModelSpace 函数用于将实体添加到模型空间，同样使用 static 关键字来限制其名称的可见性：

```
static AcDbObjectId PostToModelSpace(AcDbEntity* pEnt)
```

```
{
        AcDbBlockTable *pBlockTable;
        acdbHostApplicationServices()->workingDatabase()
            ->getBlockTable(pBlockTable, AcDb::kForRead);

        AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);

        AcDbObjectId entId;
    pBlockTableRecord->appendAcDbEntity(entId, pEnt);

        pBlockTable->close();
    pBlockTableRecord->close();
    pEnt->close();

        return entId;
}
```

这个函数相信大家现在已经非常亲切了，以后再出现的时候我们就不再占用篇幅给出其定义代码。

（5）添加一个函数

```
Acad::ErrorStatus CTransUtil::Rotate(AcDbObjectId entId,
                                     const AcGePoint2d &ptBase,
                                     double angle)
{
        // 构建变换矩阵
        AcGeMatrix3d mat;
        mat.setToRotation(angle, AcGeVector3d::kZAxis,
                AcGePoint3d(ptBase[X], ptBase[Y], 0.));

        // 对实体进行变换
        AcDbEntity *pEnt = NULL;
        Acad::ErrorStatus es = acdbOpenObject(pEnt, entId,
AcDb::kForWrite);
        if (es != Acad::eOk)
            return es;
        es = pEnt->transformBy(mat);
        pEnt->close();
```

```
        return es;
    }
```

2.14.4 效果

2.14.5 小结

注意，使用 AcAxMove 函数需要链接 axauto15.lib 库。

第3章 块和属性

块和属性的相关讨论看起来不应出现在这里，但是通过本章的知识了解块和属性与图形数据库的关系之后，你会发现创建块与第 2 章中的创建基本对象非常相似。本章介绍块和属性的相关应用，包括创建和插入普通图块、创建和插入带有属性的图块，以及在对话框中预览块的图标。

3.1 创建块定义

3.1.1 说明

本节的实例创建了一个块定义，也就是实现了 AutoCAD 中创建块的操作。块定义中包含两条直线和一个圆，创建块定义之后，在 AutoCAD 2002中选择【插入 / 块】菜单项，就能将定义的块插入到图形中。

3.1.2 思路

首先重新看一下图2.1，必须深入理解数据库的结构，才能对数据库中各种对象的操作挥洒自如。所有的实体都保存在块表记录中，而块表记录则存储在块表中。实际上，用户在AutoCAD中定义块相当于增加了一个块表记录，块表记录的名称就是块定义的名称。

为了给大家更直观的印象，使用一个Autodesk的数据库监视工具来显示块表的结构。加载配套光盘中“inspector 2002\Release\AsdkInspector.arx”文件，能够得到如图3.1所示的界面。从图中可以看出，当前图形的块表中包含了五个记录：*Model_Space、*Paper_Space、*Paper_Space0、“圆形”和“矩形”，其中前三条记录是图形数据库默认的三条记录，分别代表模型空间、图纸空间中的“布局1”和“布局2”，后两条记录代表图形中有两个块定义“圆形”和“矩形”。

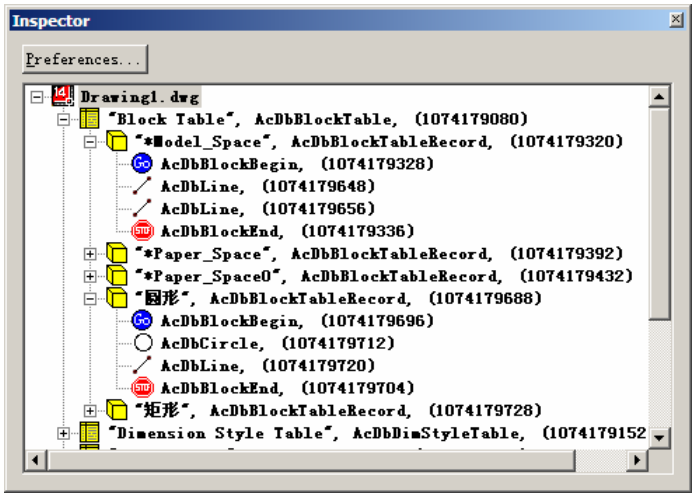

图3.1 数据库监视器的界面

在每个块表记录下都有 AcDbBlockBegin 和 AcDbBlockEnd 两个节点，两者之间的节点表示块表记录中存在的实体，例如“圆形”块定义包含了一条直线和一个圆。比较之后还可以发现，除了名称不同，用户自定义的“圆形”、“矩形”块表记录和图形数据库中的 *Model_Space 块表记录在结构上没有任何区别。

于是，仿照在图形数据库中创建实体的步骤，给出创建块定义的一般步骤：

（1）获得当前图形数据库的块表，向其中添加一条新的块表记录。

（2）创建组成块定义的实体，将其添加到新的块表记录中。

（3）关闭块表、块表记录和新创建的实体。

3.1.3 步骤

在 VC++ 6.0中，使用 ObjectARX 向导创建一个新工程，命名为 MakeBlkDef。注册一个新命令 AddBlk，其实现函数为：

```
void ZffCHAP3AddBlk()
{
        // 获得当前图形数据库的块表
        AcDbBlockTable *pBlkTbl;
        acdbHostApplicationServices()->workingDatabase()
                ->getBlockTable(pBlkTbl, AcDb::kForWrite);

        // 创建新的块表记录
        AcDbBlockTableRecord *pBlkTblRcd;
        pBlkTblRcd = new AcDbBlockTableRecord();

        // 根据用户的输入设置块表记录的名称
        char blkName[40];
```

```
                    if (acedGetString(Adesk::kFalse, "\n输入图块的名称：", blkName) !=
RTNORM)
                    {
                        pBlkTbl->close();
                        delete pBlkTblRcd;
                        return;
                    }
                    pBlkTblRcd->setName(blkName);

                    // 将块表记录添加到块表中
                    AcDbObjectId blkDefId;
                    pBlkTbl->add(blkDefId, pBlkTblRcd);
                    pBlkTbl->close();

                    // 向块表记录中添加实体
                    AcGePoint3d ptStart(-10, 0, 0), ptEnd(10, 0, 0);
                    AcDbLine *pLine1 = new AcDbLine(ptStart, ptEnd); // 创建一条直线
                    ptStart.set(0, -10, 0);
                    ptEnd.set(0, 10, 0);
                    AcDbLine *pLine2 = new AcDbLine(ptStart, ptEnd); // 创建一条直线
                    AcGeVector3d vecNormal(0, 0, 1);
                    AcDbCircle *pCircle = new AcDbCircle(AcGePoint3d::kOrigin,
vecNormal, 6);

                    AcDbObjectId entId;
                    pBlkTblRcd->appendAcDbEntity(entId, pLine1);
                    pBlkTblRcd->appendAcDbEntity(entId, pLine2);
                    pBlkTblRcd->appendAcDbEntity(entId, pCircle);

                    // 关闭实体和块表记录
                    pLine1->close();
                    pLine2->close();
                    pCircle->close();
                    pBlkTblRcd->close();
        }
```

上面的代码中要求用户输入块定义的名称，使用 acedGetString 函数来获得用户输入的字符串，该函数定义为：

```
        int acedGetString(
                    int cronly,
```

```
                const char * prompt,
                char * result);
```

cronly 指定用户输入的字符串中是否可以包含空格，可以输入 Adesk::kTrue 或者 Adesk::kFalse；prompt 指定了在命令行提示用户输入的文本；result 则保存了用户输入的结果。

要想正确编译程序，必须添加对头文件“dbents.h”的包含。

3.1.4 效果

（1）编译运行程序，在 AutoCAD 2002中执行 AddBlk 命令，命令行会提示“输入图块的名称：”，输入 Center 作为图块的名称，按下 Enter 键完成操作，就在当前图形中创建了名为 Center 的块定义。

（2）在 AutoCAD 2002中，选择【插入 / 块】菜单项，系统会弹出如所示的【插入】对话框，单击【确定】按钮关闭该对话框。

图3.2 设置插入块的参数

（3）在图形窗口中拾取一点作为块参照的插入点，能够得到如图3.3所示的块参照。

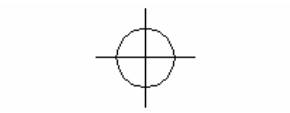

图3.3 插入的块参照

3.1.5 小结

学习本节内容之后，读者要掌握下面的知识点：

- AutoCAD 图形数据库中实体、块的存储方式。
- AutoCAD 数据库监视器的使用。
- 使用 acedGetString 函数来获得用户输入的字符串。

3.2 插入块参照

3.2.1 说明

本实例允许用户输入一个块参照的名称，指定块参照的插入点等参数，然后将块参照插入到模型空间。

3.2.2 思路

如果对 AutoCAD 的操作比较熟悉，就会知道 AutoCAD 有块定义和块参照两个概念。这两个概念有什么关系呢？块定义不是一个实体，而是一种对实体的描述，通过定义块获得；块参照则是一种实体，图形窗口中显示的“块”都是块参照，通过插入块获得。

在 ObjectARX 编程中，块定义通过块表记录来保存，而块参照由 AcDbBlockReference 类来表示。既然块参照是一种实体，那么创建块参照的过程与创建一条直线似乎不应该有什么区别，事实的确如此！

AcDbBlockReference 类的构造函数定义为：

```
AcDbBlockReference(
                const AcGePoint3d& position,
                AcDbObjectId blockTableRec);
```

position 是块参照的插入点；blockTableRec 是块参照所参照的块表记录（块定义）的 ID。

3.2.3 步骤

在 VC++ 6.0中，使用 ObjectARX 向导创建一个新的项目，名称为 InsertBlkRef。注册一个名称为 InsertBlk 的命令，其实现代码为：

```
void ZffCHAP3InsertBlk()
{
        // 获得用户输入的块定义名称
        char blkName[40];
        if (acedGetString(Adesk::kFalse, "\n输入图块的名称：", blkName) !=
RTNORM)
        {
                return;
        }

        // 获得当前数据库的块表
        AcDbBlockTable *pBlkTbl;
        acdbHostApplicationServices()->workingDatabase()
                ->getBlockTable(pBlkTbl, AcDb::kForWrite);
```

```
// 查找用户指定的块定义是否存在
CString strBlkDef;
strBlkDef.Format("%s", blkName);
if (!pBlkTbl->has(strBlkDef))
{
    acutPrintf("\n当前图形中未包含指定名称的块定义！");
    pBlkTbl->close();
    return;
}

// 获得用户输入的块参照的插入点
ads_point pt;
if (acedGetPoint(NULL, "\n输入块参照的插入点：", pt) != RTNORM)
{
    pBlkTbl->close();
    return;
}
AcGePoint3d ptInsert = asPnt3d(pt);

// 获得用户指定的块表记录
AcDbObjectId blkDefId;
pBlkTbl->getAt(strBlkDef, blkDefId);

// 创建块参照对象
AcDbBlockReference *pBlkRef = new AcDbBlockReference(ptInsert,
blkDefId);

// 将块参照添加到模型空间
AcDbBlockTableRecord *pBlkTblRcd;
pBlkTbl->getAt(ACDB_MODEL_SPACE, pBlkTblRcd,
AcDb::kForWrite);
AcDbObjectId entId;
pBlkTblRcd->appendAcDbEntity(entId, pBlkRef);

// 关闭数据库的对象
pBlkRef->close();
pBlkTblRcd->close();
pBlkTbl->close();
```

```
}
```

为了增加函数的容错能力，增加了判断当前图形的块表中是否包含与指定名称匹配的块表记录的语句，使用 AcDbBlockTable::has 函数进行检查。

acedGetPoint 函数能够暂停程序，等待用户输入一个点，然后获得该点的坐标。该函数定义为：

```
int acedGetPoint(
                const ads_point pt,
                const char * prompt,
                ads_point result);
```

pt在当前UCS中指定了一个相对基点，如果指定了该参数，当用户移动鼠标选择要输入的点时，系统会在光标到基点之间创建一根临时直线，如果不需要基点，可以直接输入NULL；prompt将会显示在命令行中，给用户提供指导信息，如果不需要该参数，输入NULL即可；result参数返回用户输入的点。关于acedGetPoint函数返回值的问题，请参考第5章　中关于获得用户输入信息的相关内容。

ads_point 是原来的 ADS 编程中定义的一种数据类型，其定义为：

```
typedef ads_real ads_point[3];
```

而 ads_real 则被定义为：

```
typedef double     ads_real;
```

可以看出，ads_point 实际上是一个三维浮点数组，它至今仍在与 ADS 相关的编程中使用。从 ads_point 转换到 AcGePoint3d 类型的点，即可以通过数组元素直接赋值，也可以通过 asPnt3d 函数直接转化：

- 通过数组元素交换：

```
ptInsert[X] = pt[X];
ptInsert[Y] = pt[Y];
ptInsert[Z] = pt[Z];
```

- 使用 asPnt3d 函数（需要包含 geassign.h 头文件）：

```
AcGePoint3d ptInsert = asPnt3d(pt);
```

之所以可以使用 X、Y 和 Z 直接作为数组的下标，是由于这三个字母在 ObjectARX 有特殊的定义：

```
enum { X = 0, Y = 1, Z = 2 };
```

3.2.4 效果

编译运行程序，在 AutoCAD 2002中创建一个块定义，然后执行 InsertBlk 命令，按照命令提示进行操作：

```
命令: insertblk
输入图块的名称：circle
输入块参照的插入点：
```

完成操作后，就能在图形窗口中插入指定的块参照，其在 X、Y 和 Z 方向的插入比例均为1。

如果用户输入的图块名称未在当前图形中定义，系统则会在命令行提示“当前图形中未包含指定名称的块定义!”，并且自动退出。

3.2.5 小结

学习本节实例之后，需要掌握下面的知识点：

- 结合图形数据库的基本结构，熟悉创建块参照的过程。
- acedGetPoint 函数的使用。
- ads_point 和 AcGePoint3d 的转化。

3.3 创建带有属性的块定义

3.3.1 说明

本实例创建一个带有属性的块定义，当用户使用 AutoCAD 中的 INSERT 命令向图形窗口中插入该图块时，系统会提示用户输入属性值。

3.3.2 思路

与创建普通的块定义相比，创建带有属性块定义的步骤完全相同，唯一的不同点就是在块定义中包含了一个属性定义的对象。

在 ObjectARX 中，AcDbAttributeDefinition 类表示属性定义对象，属性定义是 AutoCAD 的一种图形对象（对应于 AutoCAD 中的“属性”），可以直接创建该类的一个对象，然后将其添加到块表记录中。

3.3.3 步骤

在 VC++ 6.0中，使用 ObjectARX 向导创建一个名为 BlkWithAttribute 的项目，注册一个命令 AddBlk，其实现函数为：

```
void ZffCHAP3AddBlk()
{
        // 获得当前图形数据库的块表
        AcDbBlockTable *pBlkTbl;
        acdbHostApplicationServices()->workingDatabase()
            ->getBlockTable(pBlkTbl, AcDb::kForWrite);

        // 创建新的块表记录
        AcDbBlockTableRecord *pBlkTblRcd;
```

```
                pBlkTblRcd = new AcDbBlockTableRecord();

                // 根据用户的输入设置块表记录的名称
                char blkName[40];
                if (acedGetString(Adesk::kFalse, "\n输入图块的名称：", blkName) !=
RTNORM)
                {
                        pBlkTbl->close();
                        delete pBlkTblRcd;
                        return;
                }
                pBlkTblRcd->setName(blkName);

                // 将块表记录添加到块表中
                AcDbObjectId blkDefId;
                pBlkTbl->add(blkDefId, pBlkTblRcd);
                pBlkTbl->close();

                // 向块表记录中添加实体
                AcGePoint3d ptStart(-10, 0, 0), ptEnd(10, 0, 0);
                AcDbLine *pLine1 = new AcDbLine(ptStart, ptEnd); // 创建一条直线
                ptStart.set(0, -10, 0);
                ptEnd.set(0, 10, 0);
                AcDbLine *pLine2 = new AcDbLine(ptStart, ptEnd); // 创建一条直线
                AcGeVector3d vecNormal(0, 0, 1);
                AcDbCircle *pCircle = new AcDbCircle(AcGePoint3d::kOrigin,
vecNormal, 6);

                // 创建一个属性　输入直径
                AcDbAttributeDefinition *pAttDef = new AcDbAttributeDefinition(
                        ptEnd, "20", "直径", "输入直径");

                AcDbObjectId entId;
                pBlkTblRcd->appendAcDbEntity(entId, pLine1);
                pBlkTblRcd->appendAcDbEntity(entId, pLine2);
                pBlkTblRcd->appendAcDbEntity(entId, pCircle);
                pBlkTblRcd->appendAcDbEntity(entId, pAttDef);

                // 关闭实体和块表记录
```

```
        pLine1->close();
        pLine2->close();
        pCircle->close();
        pAttDef->close();
        pBlkTblRcd->close();
    }
```

与创建普通的块定义相比，上面的函数多创建了一个属性定义，并将其添加到新建的块表记录中。创建属性定义和向块表记录中添加属性定义的语句，在代码中用粗体表示。

AcDbAttributeDefinition 类是 AcDbText 类的一个派生类，其构造函数定义为：

```
AcDbAttributeDefinition(
        const AcGePoint3d& position,
        const char* text,
        const char* tag,
        const char* prompt,
        AcDbObjectId style = AcDbObjectId::kNull);
```

position 是属性定义的插入点；text 是属性定义默认的显示文字；tag 是属性定义的标记文字；prompt 是属性定义的提示文字；style 是文字样式表记录的 ID，用来指定属性定义所使用的文字样式。

3.3.4 效果

编译运行程序，在 AutoCAD 2002中执行 AddBlk 命令，输入 blk 作为要创建的块定义的名称，完成带属性的块定义的创建。

在AutoCAD 2002中，选择【插入 / 块】菜单项，系统会弹出如图3.4所示的对话框。选择blk块定义，然后单击【确定】按钮。

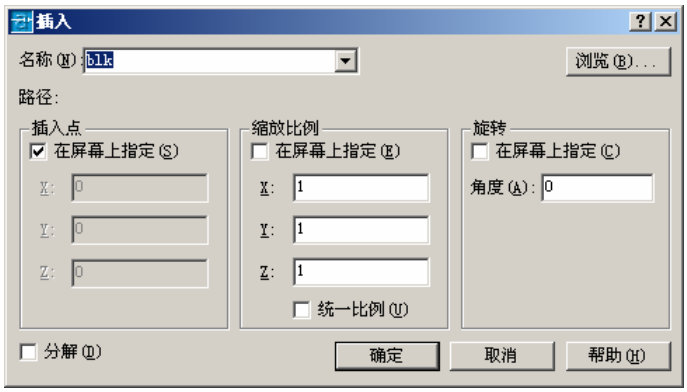

图3.4 选择要插入的块定义

按照命令行的提示进行操作：

命令: INSERT

指定插入点或 [比例(S)/X/Y/Z/旋转(R)/预览比例(PS)/PX/PY/PZ/预览旋转(PR)]:

输入属性值

输入直径 <20>: 40

完成操作后，得到如图3.5所示的结果。

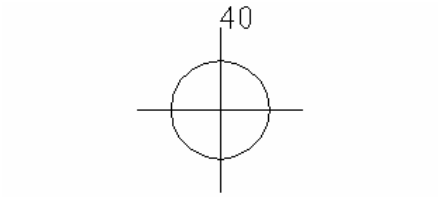

图3.5　插入块参照的结果

3.3.5 小结

学习本节内容之后，读者需要掌握下面的知识点：

- 创建带属性的块与普通块定义的区别。
- 创建属性定义的方法。

3.4 插入带有属性的块参照

3.4.1 说明

前面一节已经介绍了在 AutoCAD 中插入带属性的块参照的方法，本节将要介绍在程序中插入带属性块的方法。

3.4.2 思路

与插入普通的块定义相比，插入带有属性的块定义比较复杂。插入带有属性的块参照，实际上包含块参照和属性两个部分，属性不能直接存在于图形窗口中，而必须依附于块参照存在。

插入带有属性的块参照，可以按照下面的步骤：

（1）插入一个普通的块参照；

（2）使用遍历器遍历块参照对应的块表记录，如果找到一个属性定义，就创建一个属性，并且附加到块参照上。

3.4.3 步骤

打开 BlkWithAttribute 项目，在其中注册 InsertBlk 命令，该命令的实现函数为：

```
void ZffCHAP3InsertBlk()
```

```
{
        // 获得用户输入的块定义名称
        char blkName[40];
        if (acedGetString(Adesk::kFalse, "\n输入图块的名称：", blkName) !=
RTNORM)
        {
                return;
        }

        // 获得当前数据库的块表
        AcDbBlockTable *pBlkTbl;
        acdbHostApplicationServices()->workingDatabase()
                ->getBlockTable(pBlkTbl, AcDb::kForWrite);

        // 查找用户指定的块定义是否存在
        CString strBlkDef;
        strBlkDef.Format("%s", blkName);
        if (!pBlkTbl->has(strBlkDef))
        {
                acutPrintf("\n当前图形中未包含指定名称的块定义！");
                pBlkTbl->close();
                return;
        }

        // 获得用户输入的块参照的插入点
        ads_point pt;
        if (acedGetPoint(NULL, "\n输入块参照的插入点：", pt) != RTNORM)
        {
                pBlkTbl->close();
                return;
        }
        AcGePoint3d ptInsert = asPnt3d(pt);

        // 获得用户指定的块表记录
        AcDbObjectId blkDefId;
        pBlkTbl->getAt(strBlkDef, blkDefId);

        // 创建块参照对象
        AcDbBlockReference *pBlkRef = new AcDbBlockReference(ptInsert,
```

```
blkDefId);

                    // 将块参照添加到模型空间
                    AcDbBlockTableRecord *pBlkTblRcd;
                    pBlkTbl->getAt(ACDB_MODEL_SPACE, pBlkTblRcd,
AcDb::kForWrite);
                    pBlkTbl->close();
                    AcDbObjectId entId;
                    pBlkTblRcd->appendAcDbEntity(entId, pBlkRef);

                    // 判断指定的块表记录是否包含属性定义
                    AcDbBlockTableRecord *pBlkDefRcd;
                    acdbOpenObject(pBlkDefRcd, blkDefId, AcDb::kForRead);
                    if (pBlkDefRcd->hasAttributeDefinitions())
                    {
                        AcDbBlockTableRecordIterator *pItr;
                        pBlkDefRcd->newIterator(pItr);
                        AcDbEntity *pEnt;

                        for (pItr->start(); !pItr->done(); pItr->step())
                        {
                            pItr->getEntity(pEnt, AcDb::kForRead);

                            // 检查是否是属性定义
                            AcDbAttributeDefinition *pAttDef;
                            pAttDef = AcDbAttributeDefinition::cast(pEnt);
                            if (pAttDef != NULL)
                            {
                                // 创建一个新的属性对象
                                AcDbAttribute *pAtt = new AcDbAttribute();
                                // 从属性定义获得属性对象的对象特性
                                pAtt->setPropertiesFrom(pAttDef);
                                // 设置属性对象的其他特性
                                pAtt->setInvisible(pAttDef->isInvisible());
                                AcGePoint3d ptBase = pAttDef->position();
                                ptBase += pBlkRef->position().asVector();
                                pAtt->setPosition(ptBase);
                                pAtt->setHeight(pAttDef->height());
                                pAtt->setRotation(pAttDef->rotation());
```

```
                        // 获得属性对象的Tag、Prompt和TextString
                        char *pStr;
                        pStr = pAttDef->tag();
                        pAtt->setTag(pStr);
                        free(pStr);
                        pStr = pAttDef->prompt();
                        acutPrintf("%s%s", "\n", pStr);
                        free(pStr);
                        pAtt->setFieldLength(30);
                        pAtt->setTextString("40");

                        // 向块参照追加属性对象
                        pBlkRef->appendAttribute(pAtt);
                        pAtt->close();
                    }
                    pEnt->close();
                }
                delete pItr;
            }

            // 关闭数据库的对象
            pBlkRef->close();
            pBlkTblRcd->close();
            pBlkDefRcd->close();
        }
```

遍历块表记录的方法在上一章已经介绍，如果查找到一个属性定义，就创建一个新的属性，属性的各项特性与属性定义有关：

- 属性的图层特性与属性定义保持一致，使用 setPropertiesFrom 函数实现。
- 属性的可见性与属性定义一致。
- 属性的高度和角度与属性定义一致。
- 属性的插入点：属性定义的插入点与块参照插入点的矢量和。
- 属性的标记文字、提示文字与属性定义保持一致。

AcDbAttributeDefinition 类的 tag 和 prompt 函数返回值均是指向字符串的指针，那么在使用时可以遵照下面的方法：

```
char *pStr;
// 使用pStr
……
free(pStr);
```

可以用 ObjectARX 的全局函数 acutDelString 来代替 free 函数。

3.4.4 效果

编译运行程序，在 AutoCAD 2002中首先执行 AddBlk 命令，输入 blk 作为块定义的名称。执行 InsertBlk 命令，按照命令提示进行操作：

```
命令: insertblk
输入图块的名称：blk
输入块参照的插入点：          〖拾取一点作为块参照的插入点〗
输入直径
```

完成操作后，就能将带有属性的块参照插入到当前图形中。

3.4.5 小结

学习本节内容之后，读者需要掌握下面的知识点：

- 使用 setPropertiesFrom 函数复制另一个实体的对象特性。
- 根据属性定义和块参照插入点计算属性的插入点。

3.5 在对话框中查看块定义的图标

3.5.1 说明

本节的实例提供了一个块定义查看器，对话框中会显示当前图形中所有的块定义，选择某一个块参照之后，如果它包含预览图标，则会在对话框中显示预览图标，否则会显示一个空白的位图。

除了能够显示块定义的预览图标之外，还能够查看图形中参照了指定块定义的块参照的数量，以及块定义是否包含属性的信息。

3.5.2 思路

AcDbBlockTableRecord 类的 getPreviewIcon 函数能够提取出块表记录中的预览图标数据，其定义为：

```
Acad::ErrorStatus
        getPreviewIcon(
        PreviewIcon & previewIcon) const;
```

PreviewIcon 在 ObjectARX 中被定义为：

```
typedef AcArray<Adesk::UInt8> PreviewIcon;
```

要从该数组中获得块定义的预览图标，

3.5.3 步骤

（1）在VC++ 6.0中，使用ObjectARX向导创建一个新工程，命名为GetBlkPreviewIcon。在当前项目中，使用资源编辑器创建一个对话框资源，其控件布置如图3.6所示。

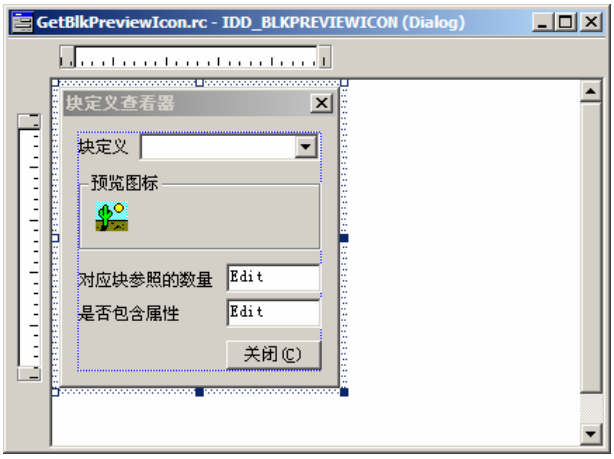

图3.6 创建对话框资源

关于创建对话框资源的方法，可以参照第8章 的相关内容。对话框中各控件的ID设置为：

- 对话框：IDD_BLKPREVIEWICON。
- 列表框：IDC_BLKDEF_LIST。
- 图片框：IDC_BITMAP。
- 块参照数量文本框：IDC_NUM_BLKDEF。
- 是否包含属性文本框：IDC_HAS_ATT。
- “关闭”按钮：IDCANCEL。

（2）单击ObjectARX嵌入工具栏的“ObjectARX MFC Support”按钮，系统会弹出如图3.7所示的对话框。选择CacUiDialog作为要创建的新类的基类，输入CpreviewBlkDlg作为新类的名称。单击【Create Class】按钮，在当前项目中创建一个与对话框关联的新类，然后单击【OK】按钮关闭该对话框。

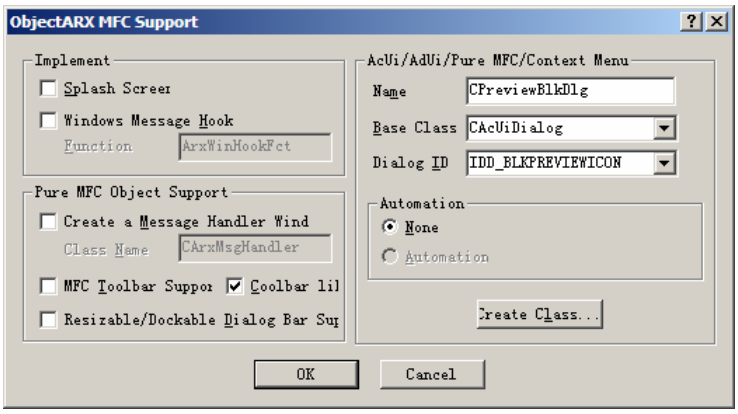

图3.7　设置新类的名称和基类

（3）在VC++ 6.0中，选择【Insert/Bitmap】菜单项，系统会弹出如图3.8所示的对话框。从资源列表中选择Bitmap，然后单击【New】按钮，向项目中添加一个空白的位图。

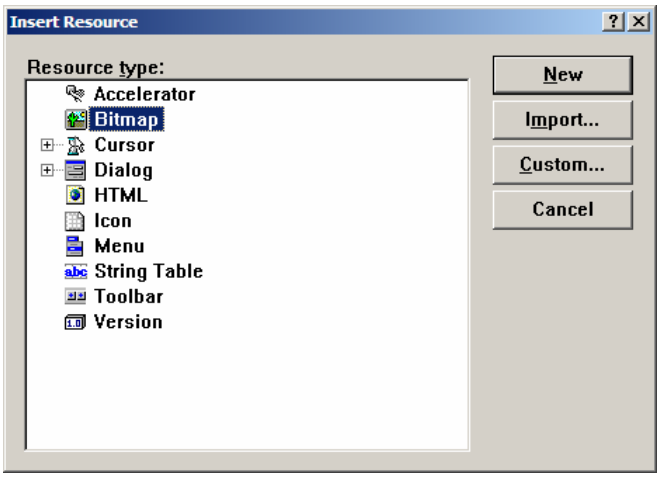

图3.8　插入位图资源

（4）切换到ResourceView选项板，右键单击位图的名称，从弹出的快捷菜单中选择【Properties】菜单项，系统会弹出如图3.9所示的对话框。设置位图的ID为IDB_BLANK，文件名称设置为blank.bmp，位图的宽度设置为30，高度设置为30。

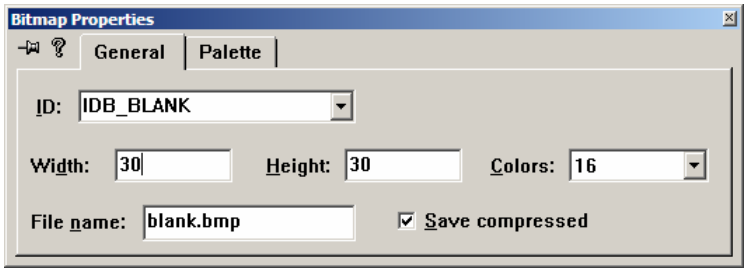

图3.9　设置位图资源的 ID 和保存位置

（5）在对话框资源模板中选择显示位图的图片框，按下Enter键，系统会弹出如图3.10所示的属性对话框。从【Type】组合框中选择Bitmap，从【Image】组合框中选择IDB_BLANK。

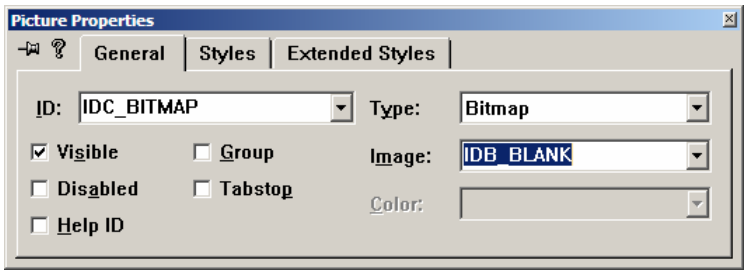

图3.10 设置图片框的内容

（6）按下Ctrl+W快捷键，系统会显示【MFC ClassWizard】对话框，切换到【Member Variables】选项卡。利用MFC的映射机制，为几个控件创建映射的成员变量，如图3.11所示。

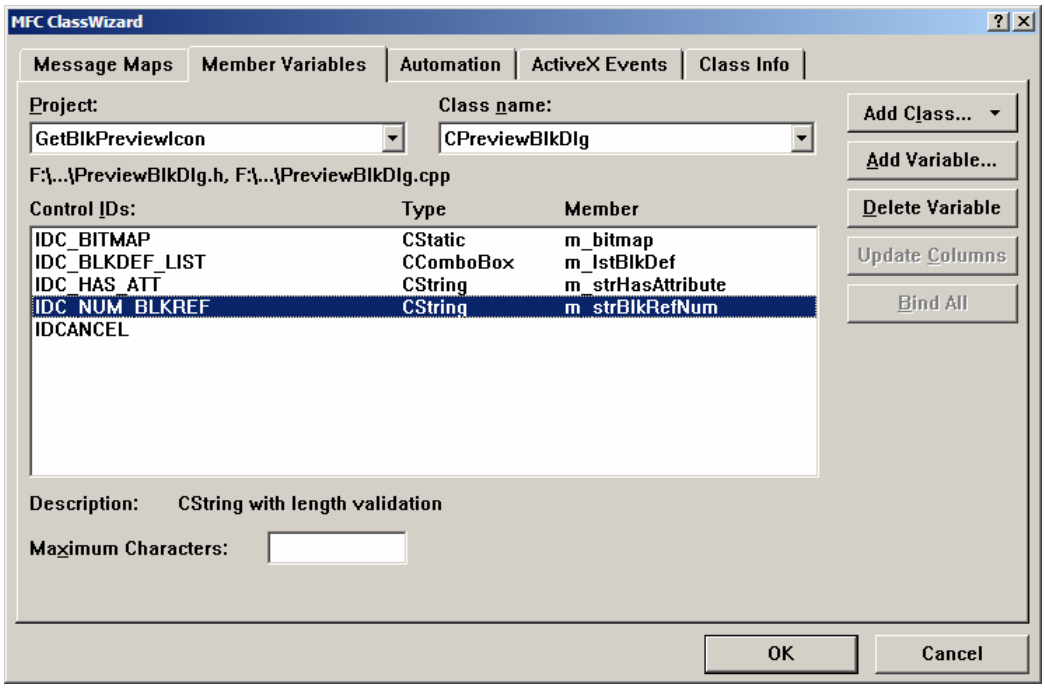

图3.11 创建控件映射的成员变量

（7）切换到【MFC ClassWizard】对话框的【Message Maps】选项卡，利用MFC的消息映射机制，改写对话框的WM_INITDIALOG消息（ON_WM_INITDIALOG）和组合框的CBN_SELCHANGE消息，如图3.12所示。

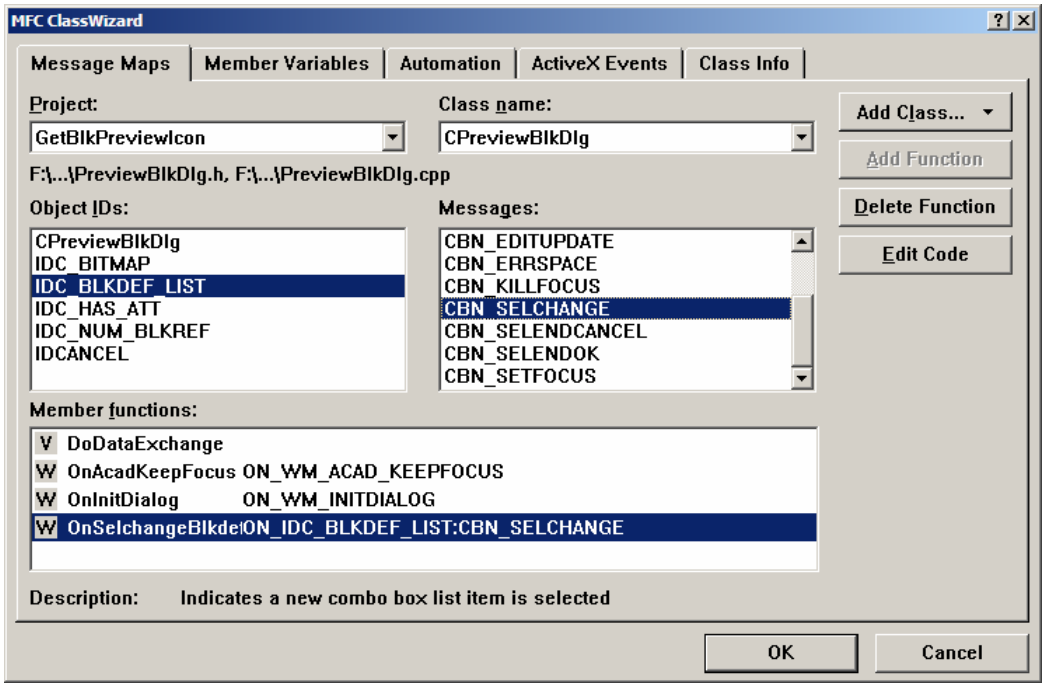

图3.12　重写对话框和组合框的消息

（8）在【ClassView】选项卡中，右键单击CpreviewBlkDlg类，从弹出的快捷菜单中选择【Add Member Function】菜单项，系统会弹出如图3.13所示的对话框。输入函数的返回值类型、名称和参数列表，单击【OK】按钮关闭该对话框。

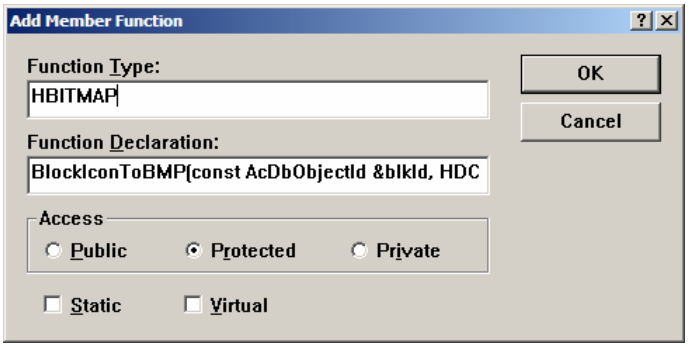

图3.13　输入成员函数的原型

（9）函数 BlockIconToBMP 用于获得指定的块表记录的预览图像，输入参数为块表记录的 ID 和设备描述表的句柄，返回值为位图句柄，其实现代码为：

```
HBITMAP CPreviewBlkDlg::BlockIconToBMP(const AcDbObjectId &blkId, HDC hdc)
{
        Acad::ErrorStatus es;
        AcDbBlockTableRecord *pBlkTblRcd = NULL;  // 块表记录的指针
        AcArray<Adesk::UInt8> icon;                // 保存预览图标的
```

数组

```
                    // 获得保存块表记录的预览图标的数组
                    try
                    {
                        es = acdbOpenObject(pBlkTblRcd, blkId, AcDb::kForRead);

                        if (es != Acad::eOk)
                            throw 1;

                        if (!pBlkTblRcd->hasPreviewIcon())        // 如果块定义不包含预
览图标
                        {
                            pBlkTblRcd->close();
                            return NULL;
                        }

                        es = pBlkTblRcd->getPreviewIcon(icon);
                        if (es != Acad::eOk)
                            throw 2;

                        es = pBlkTblRcd->close();
                        if (es != Acad::eOk)
                            throw 3;

                    }
                    catch (...)
                    {
                        pBlkTblRcd->close();
                        return NULL;
                    }

                    // 由icon数组获得可显示的位图
                    BITMAPINFOHEADER ih;          // 位图信息头
                    memcpy(&ih, icon.asArrayPtr(), sizeof(ih));
                    size_t memsize = sizeof(BITMAPINFOHEADER) + ((1<<ih.biBitCount)
* sizeof(RGBQUAD));
                    LPBITMAPINFO bi = (LPBITMAPINFO)malloc(memsize);      // 位图
信息
```

```
        memcpy(bi, icon.asArrayPtr(), memsize);
        HBITMAP hbm = CreateDIBitmap(hdc, &ih, CBM_INIT,
            icon.asArrayPtr() + memsize, bi, DIB_RGB_COLORS);
        free(bi);

        return hbm;
    }
```

AcDbBlockTableRecord 类的 hasPreviewIcon 函数用于判断指定的块表记录是否包含预览图标，getPreviewIcon 函数则可以从块表记录中获得预览图标的相关数据。

（10）在对话框的初始化函数中，添加获得当前图形中用户定义的所有块表记录，并将其添加到组合框中，其相关代码为：

```
BOOL CPreviewBlkDlg::OnInitDialog()
{
    CAcUiDialog::OnInitDialog();

    // 获得当前图形的块表
    AcDbBlockTable *pBlkTbl;
    AcDbBlockTableRecord *pBlkTblRcd;
    acdbHostApplicationServices()->workingDatabase()
        ->getBlockTable(pBlkTbl, AcDb::kForRead);

    // 遍历块表，获得用户自定义块表记录的名称，将其添加到组合框中
    AcDbBlockTableIterator *pItr;
    pBlkTbl->newIterator(pItr);
    for (pItr->start(); !pItr->done(); pItr->step())
    {
        pItr->getRecord(pBlkTblRcd, AcDb::kForRead);

        char *pName;
        pBlkTblRcd->getName(pName);
        CString strName(pName);
        acutDelString(pName);

        if (strName.Compare(ACDB_MODEL_SPACE) != 0 &&
            strName.Compare(ACDB_PAPER_SPACE) != 0 &&
            strName.Compare("*Paper_Space0"))
        {
            m_lstBlkDef.AddString(strName);
        }
```

```
            pBlkTblRcd->close();
        }
        delete pItr;
        pBlkTbl->close();

        return TRUE;  // return TRUE unless you set the focus to a control
                      // EXCEPTION: OCX Property Pages should return
FALSE
    }
```

（11）响应组合框选相切换事件的函数，能够显示选择的块定义的预览图标、图形中相应的块参照的数量，以及块定义是否包含属性的信息，其定义代码为：

```
    void CPreviewBlkDlg::OnSelchangeBlkdefList()
    {
        // 显示预览图标
        CPaintDC dc(this);
        AcDbBlockTable *pBlkTbl;
        acdbHostApplicationServices()->workingDatabase()
            ->getBlockTable(pBlkTbl, AcDb::kForRead);

        AcDbObjectId blkTblRcdId;
        CString strBlkDefName;
        m_lstBlkDef.GetLBText(m_lstBlkDef.GetCurSel(), strBlkDefName);
        Acad::ErrorStatus es = pBlkTbl->getAt(strBlkDefName, blkTblRcdId);

       HBITMAP hBitmap = BlockIconToBMP(blkTblRcdId, dc.GetSafeHdc());

        m_bitmap.SetBitmap(hBitmap);

        // 获得块表记录的指针
        AcDbBlockTableRecord *pBlkTblRcd;
        pBlkTbl->getAt(strBlkDefName, pBlkTblRcd, AcDb::kForRead);

        // 获得块参照的数量
        AcDbBlockReferenceIdIterator *pItr;
        pBlkTblRcd->newBlockReferenceIdIterator(pItr);
        int number = 0;
        for (pItr->start(); !pItr->done(); pItr->step())
        {
```

```
            number++;
        }
        m_strBlkRefNum.Format("%d", number);

        // 获得块参照是否包含属性
        if (pBlkTblRcd->hasAttributeDefinitions())
        {
            m_strHasAttribute = "是";
        }
        else
        {
            m_strHasAttribute = "否";
        }

        pBlkTblRcd->close();
        pBlkTbl->close();

        UpdateData(FALSE);
    }
```

（12）在 PreviewBlkDlg.h 文件中，添加下面的包含语句：

```
#include "Resource.h"
```

在 PreviewBlkDlg.cpp 文件中，添加下面的包含语句：

```
#include "dbsymtb.h"
#include "acutmem.h"
```

（13）注册一个新命令 ViewBlk，能够显示“块定义查看器”对话框，其实现函数为：

```
void ZffCHAP3ViewBlk()
{
        // 显示“块定义查看器”对话框
        CAcModuleResourceOverride resOverride;          // 防止资源冲突
        CPreviewBlkDlg dlg;
        dlg.DoModal();
}
```

3.5.4 效果

编译运行程序，在 AutoCAD 2002中，创建若干个块定义，然后执行 ViewBlk 命令，系统会弹出如所示的对话框。

图3.14 程序运行界面

从【块定义】组合框中选择一个块定义，系统会显示其预览图标、对应块参照的数量，以及是否包含属性的信息，如图3.15所示。

图3.15 显示块定义的预览图标

3.5.5 小结

学习本节内容之后，读者应该掌握下面的要点：

- 获得当前图形中所有块定义名称的方法。
- 获得图形中某一类块参照的数量。
- 判断块定义是否包含属性定义。
- 获得块定义的预览图标。

第4章 符号表

在探讨图形数据库的基本结构时已经了解到，图形数据库由符号表和命名对象字典组成。符号表是 AutoCAD 中的一种容器对象，保存了对应的符号表记录，用来实现 AutoCAD 中的某种对象：

- 块表（AcDbBlockTable）：包含模型空间、图纸空间和用户创建的块定义，块表记录中保存了图形数据库中的实体。
- 层表（AcDbLayerTable）：保存了图形中所有的图层，可通过 AutoCAD 中的 LAYER 命令查看。
- 文字样式表（AcDbTextStyleTable）：存储图形中的文字样式，通过 AutoCAD 中的 STYLE 命令查看。
- 线型表（AcDbLinetypeTable）：保存了图形中加载的线型，通过 AutoCAD 中的 LTYPE 命令查看。
- 视图表（AcDbViewTable）：存储了图形中保存的视图，通过 AutoCAD 中的 VIEW 命令查看。
- UCS 表（AcDbUCSTable）：保存图形中的 UCS（用户坐标系），通过 AutoCAD 的 UCS 命令访问。
- 视口表（AcDbViewportTable）：保存图形中视口的设置，通过 VPORT 命令访问。
- 注册应用程序表（AcDbRegAppTable）：在使用扩展数据等特性时，需要根据应用程序名称来区别不同程序使用的数据，这就需要使用注册一个应用程序。AutoCAD 中未提供直接访问的命令。
- 标注样式表（AcDbDimStyleTable）：保存标注样式，通过 AutoCAD 中的 DIMSTYLE 命令访问。

从数据库获得各种符号表的方法大同小异，除了可以使用 getSymbolTable 函数，还可根据符号表的具体类型调用相应的函数，如获得块表使用 getBlockTable 函数，获得层表使用 getLayerTable 函数。

所有的符号表都继承自 AcDbSymbolTable 类，该类包含了下面几个函数：

- add：向符号表添加一条新的记录，各种符号表实现的形式略有不同。
- getAt：获得符号表中特定名称的记录。
- has：判断符号表中是否包含指定的记录。
- newIterator：创建一个符号表遍历器，访问符号表中的所有记录。

4.1 操作图层

4.1.1 说明

本实例演示了创建新的层表记录、删除已有的层表记录、使用“颜色”对话框设置某一图层的颜色，以及导出和导入所有图层及特性的方法，基本涵盖了所有的层表操作。

4.1.2 思路

创建新的图层，实际上就是创建一个新的层表记录，并将其添加到层表中。

修改图层的颜色，可以从层表中获得指定的记录，然后使用 AcDbLayerTableRecord 类的 setColor 函数设置层表记录的颜色。

删除一个图层，需要首先从层表中获得指定的层表记录，然后将层表记录设置一个“删除”的标记。

导出图层列表和图层特性，需要使用层表遍历器访问每一个层表记录，将层表记录的名称、颜色、线型和线宽以“,”作分隔符连接成一个 Cstring 类型的字符串对象，然后使用 CstdioFile 类的 WriteString 函数写入到文本文件中。

导入图层列表的步骤和导出的步骤完全相反，先使用 CstdioFile 类的 ReadString 函数逐行读取文本文件的内容，以“,”作分隔符解析出图层名称、颜色、线型和线宽，并在当前图形中加入这些图层。

4.1.3 步骤

（1）在 VC++ 6.0中，使用 ObjectARX 向导创建一个新项目，命名为 OperateLayer。注册 NewLayer 命令，用于在图形中创建一个新图层，其实现函数为：

```
void ZffCHAP4NewLayer()
{
        // 提示用户输入新建图层的名称
        char layerName[100];
        if (acedGetString(Adesk::kFalse, "\n输入新图层的名称：",
                layerName) != RTNORM)
        {
                return;
        }

        // 获得当前图形的层表
        AcDbLayerTable *pLayerTbl;
        acdbHostApplicationServices()->workingDatabase()
                ->getLayerTable(pLayerTbl, AcDb::kForWrite);
```

```
        // 是否已经包含指定的层表记录
        if (pLayerTbl->has(layerName))
        {
            pLayerTbl->close();
            return;
        }

        // 创建新的层表记录
        AcDbLayerTableRecord *pLayerTblRcd;
        pLayerTblRcd = new AcDbLayerTableRecord();
        pLayerTblRcd->setName(layerName);

        // 将新建的层表记录添加到层表中
        AcDbObjectId layerTblRcdId;
        pLayerTbl->add(layerTblRcdId, pLayerTblRcd);

        acdbHostApplicationServices()->workingDatabase()
            ->setClayer(layerTblRcdId);

        pLayerTblRcd->close();
        pLayerTbl->close();
    }
```

由于要在块表中添加新的块表记录，在获得块表的时候需要将其以“写”模式打开，对应于代码中就是在 getLayerTable 函数中使用了 AcDb::kForWrite 参数。向块表添加新的记录之前，可以使用 has 函数判断图形中是否已经包含了同名的层。

AcDbDatabase 类的 setClayer 函数能够设置图形的当前图层。

（2）注册新的命令 LayerColor，用于修改指定图层的颜色，其实现函数为：

```
    void ZffCHAP4LayerColor()
    {
        // 提示用户输入要修改的图层名称
        char layerName[100];
        if (acedGetString(Adesk::kFalse, "\n输入图层的名称：",
            layerName) != RTNORM)
        {
            return;
        }

        // 获得当前图形的层表
```

```
        AcDbLayerTable *pLayerTbl;
        acdbHostApplicationServices()->workingDatabase()
                ->getLayerTable(pLayerTbl, AcDb::kForRead);

        // 判断是否包含指定名称的层表记录
        if (!pLayerTbl->has(layerName))
        {
                pLayerTbl->close();
                return;
        }

        // 获得指定层表记录的指针
        AcDbLayerTableRecord *pLayerTblRcd;
        pLayerTbl->getAt(layerName, pLayerTblRcd, AcDb::kForWrite);

        // 弹出“颜色”对话框
        AcCmColor oldColor = pLayerTblRcd->color();
        int nCurColor = oldColor.colorIndex();      // 图层修改前的颜色
        int nNewColor = oldColor.colorIndex();          // 用户选择的颜色
        if (acedSetColorDialog(nNewColor, Adesk::kFalse, nCurColor))
        {
                AcCmColor color;
                color.setColorIndex(nNewColor);
                pLayerTblRcd->setColor(color);
        }

        pLayerTblRcd->close();
        pLayerTbl->close();
}
```

在获得特定的块表记录指针时，使用了使用了 AcDbLayerTable 类的 getAt 函数，并且使用 AcDb::kForWrite 参数，将块表记录以“写”模式打开。

AcedSetColorDialog 函数能够弹出【选择颜色2对话框，并且返回用户选择的结果，该函数定义为：

```
Adesk::Boolean acedSetColorDialog(
        int& nColor,
        Adesk::Boolean bAllowMetaColor,
        int nCurLayerColor);
```

nColor 参数指定了显示【选择颜色】对话框时的默认颜色，并且在函数返回值后保存用户选择的新颜色；bAllowMetaColor 参数限定在【选择颜色】对话框中是否可以选择“随层”

或“随块”；nCurLayerColor 参数指定当前图层的颜色。

AcCmColor 代表颜色对象，可以通过颜色索引来构建一个新的颜色对象。通过颜色索引，可以将【选择颜色】对话框的结果设置为指定图层的颜色，其相关代码为：

```
AcCmColor color;
color.setColorIndex(nNewColor);
pLayerTblRcd->setColor(color);
```

（3）注册一个新命令 DelLayer，用于从图形中删除指定的图层，其实现函数为：

```
void ZffCHAP4DelLayer()
{
        // 提示用户输入要修改的图层名称
        char layerName[100];
        if (acedGetString(Adesk::kFalse, "\n输入图层的名称：",
                layerName) != RTNORM)
        {
                return;
        }

        // 获得当前图形的层表
        AcDbLayerTable *pLayerTbl;
        acdbHostApplicationServices()->workingDatabase()
                ->getLayerTable(pLayerTbl, AcDb::kForRead);

        // 判断是否包含指定名称的层表记录
        if (!pLayerTbl->has(layerName))
        {
                pLayerTbl->close();
                return;
        }

        // 获得指定层表记录的指针
        AcDbLayerTableRecord *pLayerTblRcd;
        pLayerTbl->getAt(layerName, pLayerTblRcd, AcDb::kForWrite);
        pLayerTblRcd->erase();                    // 为其设置“删除”标记

        pLayerTblRcd->close();
        pLayerTbl->close();
}
```

删除一个数据库对象非常简单：将其以“写”模式打开，调用 AcDbObject 类的 erase 函数，最后关闭该对象即可。

（4）注册一个新命令 ExportLayer，用于将当前图形中存在的所有图层及其特性导出到一个文本文件中，其实现函数为：

```
void ZffCHAP4ExportLayer()
{
        // 创建所要导出的文本文件
        CStdioFile f;
        CFileException e;
        char *pFileName = "C:\\layers.txt";
        if (!f.Open(pFileName, CFile::modeCreate | CFile::modeWrite, &e))
        {
                acutPrintf("\n创建导出文件失败！");
                return;
        }

        // 获得层表指针
        AcDbLayerTable *pLayerTbl;
        AcDbLayerTableRecord *pLayerTblRcd;
        acdbHostApplicationServices()->workingDatabase()
                ->getLayerTable(pLayerTbl, AcDb::kForRead);

        // 使用遍历器访问每一条层表记录
        AcDbLayerTableIterator *pItr;
        pLayerTbl->newIterator(pItr);
        for (pItr->start(); !pItr->done(); pItr->step())
        {
                pItr->getRecord(pLayerTblRcd, AcDb::kForRead);

                // 输出图层的信息
                CString strLayerInfo;                          // 图层名称
                char *layerName;
                pLayerTblRcd->getName(layerName);
                strLayerInfo = layerName;
                free(layerName);
                strLayerInfo += ",";                           // 分隔符

                CString strColor;                              // 图层颜色
                AcCmColor color = pLayerTblRcd->color();
                strColor.Format("%d", color.colorIndex());
                strLayerInfo += strColor;
```

```
                    strLayerInfo += ",";

                    CString strLinetype;                                    // 图层线型
                    AcDbLinetypeTableRecord *pLinetypeTblRcd;
                    acdbOpenObject(pLinetypeTblRcd,
pLayerTblRcd->linetypeObjectId(),
                        AcDb::kForRead);
                    char *linetypeName;
                    pLinetypeTblRcd->getName(linetypeName);
                    pLinetypeTblRcd->close();
                    strLinetype = linetypeName;
                    free(linetypeName);
                    strLayerInfo += strLinetype;
                    strLayerInfo += ",";

                    CString strLineWeight;                                  // 图层
的线宽
                    AcDb::LineWeight lineWeight = pLayerTblRcd->lineWeight();
                    strLineWeight.Format("%d", lineWeight);
                    strLayerInfo += strLineWeight;

                    // 将图层特性写入到文件中
                    f.WriteString(strLayerInfo);
                    f.WriteString("\n");

                    pLayerTblRcd->close();
                }
                delete pItr;
                pLayerTbl->close();
        }
```

CstdioFile 类的 Open 函数能够打开指定位置的文件，这里使用 CFile::modeCreate 作为打开标记，能够在指定的位置创建文件。

最终的结果是将图层的信息以“图层名称，颜色，线型，线宽”格式输出，因此在获得图层的名称和特性之后，关键在于将这些特性组合成一个 Cstring 类型的变量。所幸，Cstring 类提供了“＋”运算符，能够将两个字符串连接起来组成一个新的字符串，例如：

```
strLayerInfo += strLineWeight;
```

像文件中写入字符串使用了 CstdioFile 类的 WriteString 函数，注意需要单独写入一个换行字符，保证每个图层的特性单独成行。

（5）注册新命令 ImportLayer，能够按照文本文件中的图层列表在当前图形中创建图层，

并且符合图层列表中的各项特性，其实现函数为：

```
void ZffCHAP4ImportLayer()
{
        // 打开所要导入的文本文件
        CStdioFile f;
        CFileException e;
        char *pFileName = "C:\\layers.txt";
        if (!f.Open(pFileName, CFile::modeRead, &e))
        {
            acutPrintf("\n打开导入文件失败！");
            return;
        }

        // 获得层表指针
        AcDbLayerTable *pLayerTbl;
        AcDbLayerTableRecord *pLayerTblRcd;
        acdbHostApplicationServices()->workingDatabase()
            ->getLayerTable(pLayerTbl, AcDb::kForWrite);

        // 读取文件中的每一行数据
        CString strLineText;                    // 一行文字
        while (f.ReadString(strLineText))
        {
            // 跳过空行
            if (strLineText.IsEmpty())
                continue;

            // 解析出图层名称、颜色、线型和线宽
            CStringArray layerInfos;
            if (!GetFieldText(strLineText, layerInfos))
                continue;

            // 创建新的层表记录，或者打开存在的块表记录
            AcDbLayerTableRecord *pLayerTblRcd;
            AcDbObjectId layerTblRcdId;
            if (pLayerTbl->has(layerInfos.GetAt(0)))
            {
                pLayerTbl->getAt(layerInfos.GetAt(0), layerTblRcdId);
            }
```

```
                    else
                    {
                        pLayerTblRcd = new AcDbLayerTableRecord();
                        pLayerTblRcd->setName(layerInfos.GetAt(0));
                        pLayerTbl->add(layerTblRcdId, pLayerTblRcd);
                        pLayerTblRcd->close();
                    }

                    acdbOpenObject(pLayerTblRcd, layerTblRcdId,
AcDb::kForWrite);

                    // 设置层表记录的颜色
                    AcCmColor color;
                    Adesk::UInt16 colorIndex = atoi(layerInfos.GetAt(1));
                    color.setColorIndex(colorIndex);
                    pLayerTblRcd->setColor(color);

                    // 设置线型
                    AcDbLinetypeTable *pLinetypeTbl;
                    AcDbObjectId linetypeId;
                    acdbHostApplicationServices()->workingDatabase()
                        ->getLinetypeTable(pLinetypeTbl, AcDb::kForRead);
                    if (pLinetypeTbl->has(layerInfos.GetAt(2)))
                    {
                        pLinetypeTbl->getAt(layerInfos.GetAt(2), linetypeId);

                    }
                    else
                    {
                        pLinetypeTbl->getAt("Continous", linetypeId);
                    }
                    pLayerTblRcd->setLinetypeObjectId(linetypeId);
                    pLinetypeTbl->close();

                    // 设置线宽
                    AcDb::LineWeight lineWeight =
(AcDb::LineWeight)atol(layerInfos.GetAt(3));
                    pLayerTblRcd->setLineWeight(lineWeight);
```

```
                pLayerTblRcd->close();
            }

            pLayerTbl->close();
        }
```

使用 CFile::modeRead 参数调用 CstdioFile 类的 Open 函数，打开指定位置的文本文件，然后使用 ReadString 函数逐行读取文本文件中的内容。

读取一行文本之后，需要根据分隔符（“，”）来解析出图层的名称、颜色、线型和线宽，使用 GetFieldText 函数来实现，该函数的实现代码为：

```
        BOOL GetFieldText(CString strLineText, CStringArray &fields)
        {
            if (strLineText.Find(",", 0) == -1)          // 如果找不到英文逗号，函数退出
            {
                return FALSE;
            }

            int nLeftPos = 0, nRightPos = 0;          // 查找分隔符的起始位置

            while ((nRightPos = strLineText.Find(",", nRightPos)) != -1)
            {
                fields.Add(strLineText.Mid(nLeftPos, nRightPos - nLeftPos));

                nLeftPos = nRightPos + 1;
                nRightPos++;
            }
            // 最后一个列的数据
            fields.Add(strLineText.Mid(nLeftPos));

            return TRUE;
        }
```

Cstring 类提供了一系列用于查找字符或者切割字符串的函数，使得解析字符串变量非常简单。例如 Find 函数能在指定的字符串中查找匹配子串的第一个位置，Mid 函数则能解析出指定字符串中指定长度的子串。

获得某一图层的名称、颜色、线型和线宽之后，就可以创建新的层表记录，并且使用 setColor 函数设置图层的颜色，setLinetypeObjectId 函数设置图层的线型，setLineWeight 函数设置图层的线宽。

4.1.4 效果

（1）编译运行程序，在AutoCAD 2002中执行NewLayer命令，然后输入NewLayer作为图层名称，完成图层的创建。选择【格式 / 图层】菜单项，系统会弹出如图4.1所示的【图层特性管理器】，图层列表中已经包含了新创建的NewLayer图层。

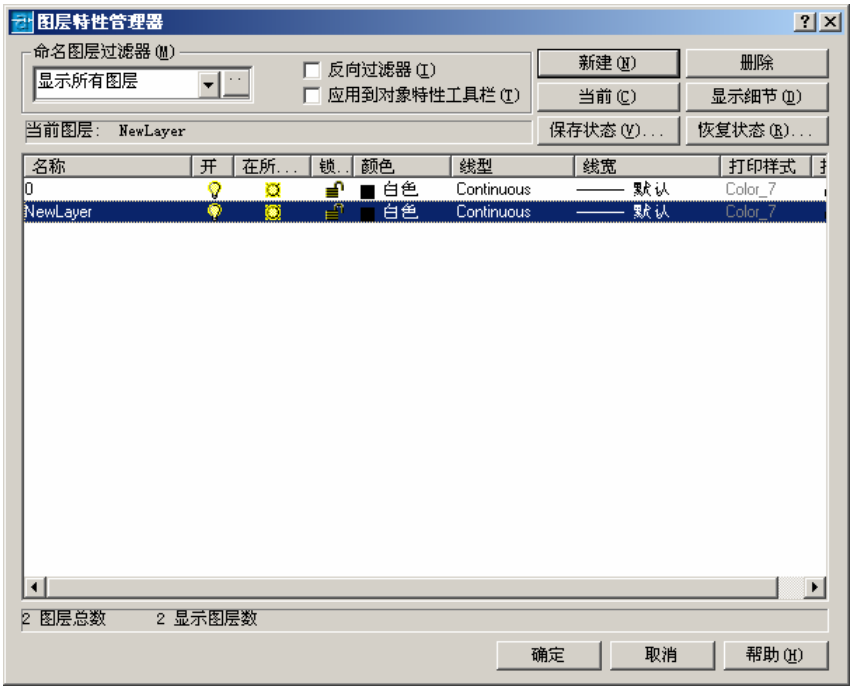

图4.1　显示新创建的图层

（2）使用“图层特性”工具栏，将0层作为当前图层。执行 DelLayer 命令，输入 NewLayer 作为图层名称，按下 Enter 键完成图层的删除。

由于 AutoCAD 不允许删除图形的当前图层，因此在删除 NewLayer 图层之前，必须保证它不是当前图层。

（3）执行LayerColor命令，输入0作为图层名称，系统会弹出如图4.2所示的【选择颜色】对话框。从调色板中选择合适的颜色，单击【确定】按钮关闭该对话框，在【图层特性管理器】中查看0层，就会发现0层的颜色已变为蓝色。

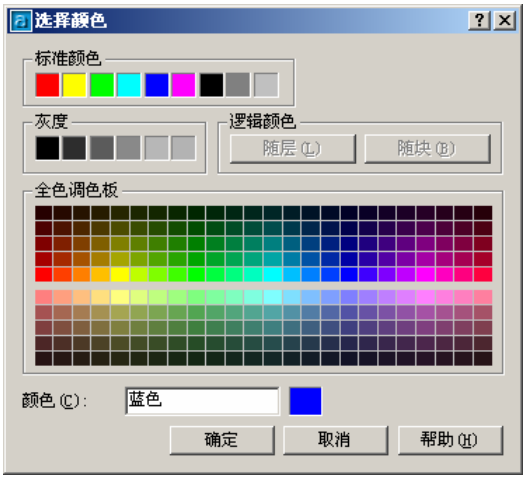

图4.2 设置0层的颜色

（4）在【图层特性管理器】中，创建若干个新图层，并且设置适当的颜色、线型和线宽，如图4.3所示。

图4.3 创建要导出的图层、设置图层特性

（5）执行ExportLayer命令，将图层列表及图层特性导出到文本文件中，得到的文件layers.txt内容如图4.4所示。

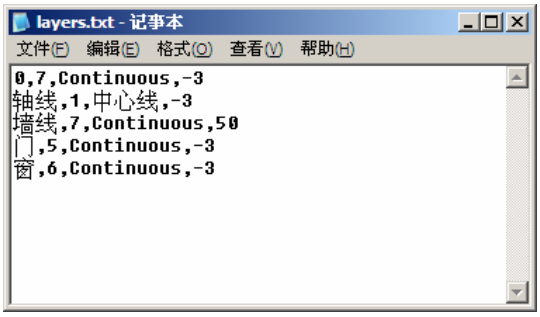

图4.4　导出文件的内容

（6）关闭当前的图形，创建一个新图形，执行 ImportLayer 命令，然后在【图层特性管理器】中查看图层列表，会发现新图形的图层及其特性与上一个图形完全一致。

4.1.5　小结

学习本节内容之后，读者需要掌握下面的要点：

- 删除数据库对象的方法。
- 使用【选择颜色】对话框。
- 逐行读写文本文件的方法。
- 获得和设置图层的颜色、线型和线宽。

4.2　创建字体样式

4.2.1　说明

在 AutoCAD 中可以使用 STYLE 命令创建新的字体样式，包括设置样式名、选择字体文件和确定字体效果三个步骤。本实例演示了创建字体样式的方法。

4.2.2　思路

使用 ObjectARX 创建字体样式，需要执行下面的步骤：

（1）获得当前图形的字体样式表；
（2）创建新的字体样式表记录对象；
（3）用 setName 函数设置字体样式表记录的名称；
（4）用 setFileName 函数设置字体样式表记录的字体；
（5）用 setXScale 函数设置字体样式的高宽比；
（6）将新的字体样式表记录添加到字体样式表中。

4.2.3 步骤

在 VC++ 6.0中，使用 ObjectARX 向导创建一个新的项目，命名为 AddTextStyle，注册一个新命令 AddStyle，用于创建新的字体样式，其实现函数为：

```
void ZffCHAP4AddStyle()
{
        // 获得字体样式表
        AcDbTextStyleTable *pTextStyleTbl;
        acdbHostApplicationServices()->workingDatabase()
                ->getTextStyleTable(pTextStyleTbl, AcDb::kForWrite);

        // 创建新的字体样式表记录
        AcDbTextStyleTableRecord *pTextStyleTblRcd;
        pTextStyleTblRcd = new AcDbTextStyleTableRecord();

        // 设置字体样式表记录的名称
        pTextStyleTblRcd->setName("仿宋体");

        // 设置字体文件名称
        pTextStyleTblRcd->setFileName("simfang.ttf");

        // 设置高宽比例
        pTextStyleTblRcd->setXScale(0.7);

        // 将新的记录添加到字体样式表
        pTextStyleTbl->add(pTextStyleTblRcd);

        pTextStyleTblRcd->close();
        pTextStyleTbl->close();
}
```

上面的代码中使用的是 TrueType 字体，如果 AutoCAD 自身的 SHX 字体，就无需指定字体文件的扩展名，例如：

```
pTextStyleTblRcd->setFileName("romans");
```

此外，字体的名称不一定与字体文件的名称相同。打开控制面板，进入“字体”文件夹，右键单击“仿宋体”图标，从弹出的快捷菜单中选择【属性】菜单项，系统会弹出如图4.5所示的对话框，显示了字体文件的名称。

图4.5　查看字体的文件名

4.2.4　效果

编译运行程序，在AutoCAD 2002中执行AddStyle命令，完成字体样式的创建。选择【格式 / 文字样式】菜单项，系统会弹出如图4.6所示的【字体样式】对话框。在【样式名】列表中，已经包含了新建的【仿宋体】样式。

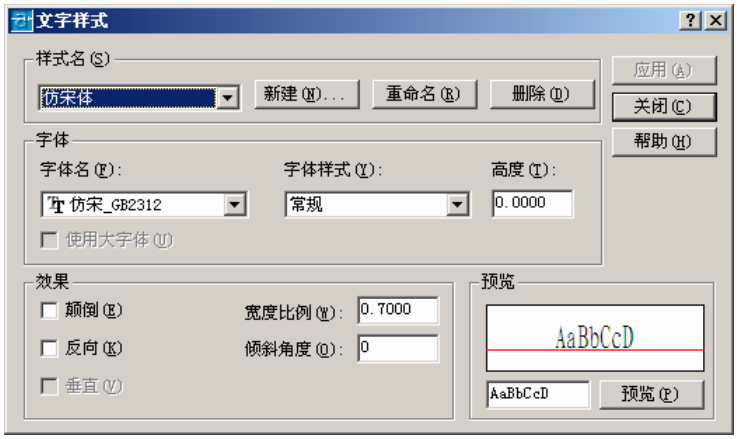

图4.6　查看新建的字体样式

4.2.5　小结

学习本节内容之后，读者杨掌握下面的要点：

- 设置字体样式的方法。

- 查看某一字体的字体文件名称。

4.3 创建标注样式

4.3.1 说明

本实例能够创建一个新的标注样式，在程序中设置了标注样式的名称、箭头大小、尺寸界线超出尺寸线的长度、文字和标注线的位置关系，以及标注文字的高度。

4.3.2 思路

与创建文字样式类似，在 ObjectARX 中创建标注样式可以按照下面的步骤进行：

（1）创建一个新的标注样式表记录对象；

（2）设置标注样式表记录的各项特性，例如标注样式的名称、文字高度、箭头大小等；

（3）将新的标注样式表记录添加到当前图形的标注样式表中。

4.3.3 步骤

在 VC++ 6.0中，使用 ObjectARX 向导创建一个新项目，命名为 AddDimStyle。注册一个命令 AddDimStyle，用于创建新的标注样式，其实现函数为：

```
void ZffCHAP4AddDimStyle()
{
        // 获得要创建的标注样式名称
        char styleName[100];
        if (acedGetString(Adesk::kFalse, "\n输入新样式的名称：",
                styleName) != RTNORM)
        {
                return;
        }

        // 获得当前图形的标注样式表
        AcDbDimStyleTable *pDimStyleTbl;
        acdbHostApplicationServices()->workingDatabase()
                ->getDimStyleTable(pDimStyleTbl, AcDb::kForWrite);

        if (pDimStyleTbl->has(styleName))
        {
                pDimStyleTbl->close();
```

```
            return;
        }

        // 创建新的标注样式表记录
        AcDbDimStyleTableRecord *pDimStyleTblRcd;
        pDimStyleTblRcd = new AcDbDimStyleTableRecord();

        // 设置标注样式的特性
        pDimStyleTblRcd->setName(styleName);        // 样式名称
        pDimStyleTblRcd->setDimasz(3);              // 箭头长度
        pDimStyleTblRcd->setDimexe(3);              // 尺寸界线与标注
点的偏移量
        pDimStyleTblRcd->setDimtad(1);              // 文字位于标注线
的上方
        pDimStyleTblRcd->setDimtxt(3);              // 标注文字的高度

        // 将标注样式表记录添加到标注样式表中
        pDimStyleTbl->add(pDimStyleTblRcd);

        pDimStyleTblRcd->close();
        pDimStyleTbl->close();
    }
```

4.3.4 效果

（1）编译运行程序，在 AutoCAD 2002中执行 AddDimStyle 命令，输入 MyStyle 作为新建的标注样式的名称，按下 Enter 键完成标注样式的创建。

（2）选择【格式 / 标注样式】菜单项，系统会弹出如所示的【标注样式管理器】，在标注样式列表中已经包含了新建的标注样式。

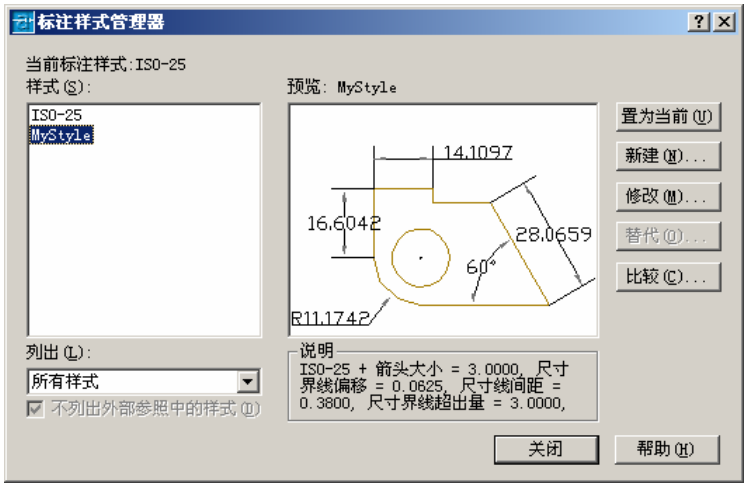

图4.7 创建了新的标注样式

（3）从样式列表中选择MyStyle样式，单击【修改】按钮，可以查看该样式的各种特性，如图4.8所示。

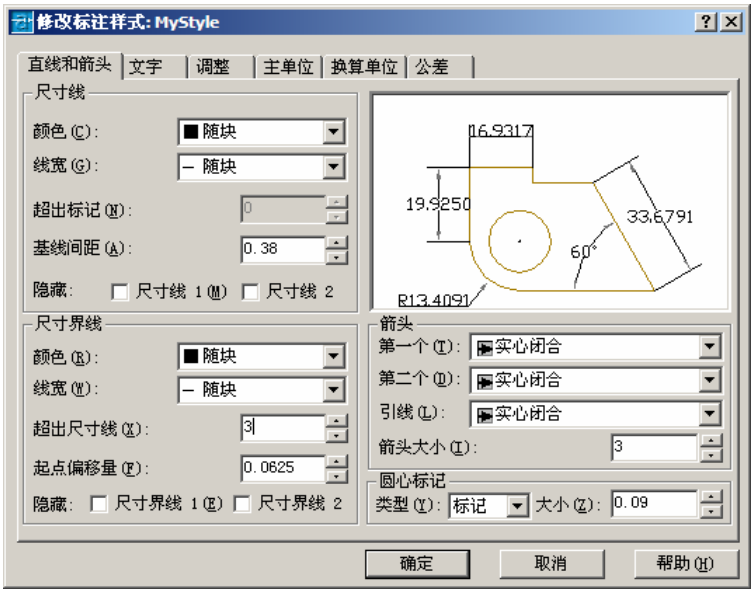

图4.8 查看标注样式的特性

4.3.5 小结

在AutoCAD中创建标注样式时，用户可以选择一种已经存在的标注样式作为基础样式，如图4.9所示。创建新的标注样式之后，新样式的所有特性与基础样式保持一致，用户只需按照自己的要求对部分特性进行修改即可，非常方便。

图4.9　选择基础样式

在ObjectARX编程中，同样可以实现。如果在创建新的标注样式时，仅设置标注样式的名称，其他的特性均采用默认值，得到的标注样式与ISO-25样式仅有20个不同之处，如图4.10所示。

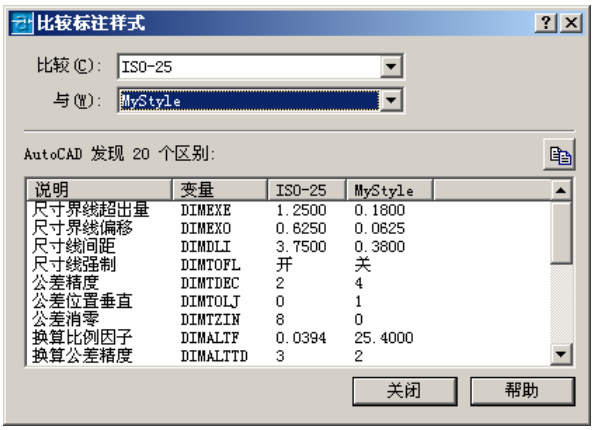

图4.10　比较标注样式

因此，在设置新标注样式的特性时，可以先获得系统中存在的标注样式，用已有的标注样式的特性来设置新的标注样式即可。下面的代码演示用已有的标注样式的部分特性来设置新建的标注样式：

```
// 创建新的标注样式表记录
AcDbDimStyleTableRecord *pDimStyleTblRcd;
pDimStyleTblRcd = new AcDbDimStyleTableRecord();

// 获得已经存在的标注样式ISO-25
AcDbDimStyleTableRecord *pOldStyle;
pDimStyleTbl->getAt("ISO-25", pOldStyle, AcDb::kForRead);

// 设置新标注样式的特性
pDimStyleTblRcd->setName(styleName);
pDimStyleTblRcd->setDimtxt(pOldStyle->dimtxt());
pDimStyleTblRcd->setDimasz(pOldStyle->dimasz());
pDimStyleTblRcd->setDimexe(pOldStyle->dimexe());
pDimStyleTblRcd->setDimtad(pOldStyle->dimtad());
```

4.4 视图

4.4.1 说明

在 AutoCAD 中，视图是指图形窗口显示的内容，使用 ZOOM 命令能够缩放视图，使用 PAN 命令能够移动视图，使用 VIEW 命令能管理视图。本节的实例，将要模拟窗口缩放、范围缩放、和比例缩放的实现。

4.4.2 思路

ObjectARX 中的 AcDbViewTableRecord 类用于表示 AutoCAD 中的视图，它从 AcDbAbstractViewTableRecord 类继承了多个成员函数。在调整视图时，一般要先获得当前视图，设置某些特性，然后使用 acedSetCurrentView 函数来更新视图。

获得当前视图在 ObjectARX 中并未直接提供相关的函数，只能通过查询系统变量的值获得当前视图的特性，创建一个新的视图对象，对其设置这些特性，将其作为当前视图。下面给出了与视图有关的系统变量：

- VIEWMODE：当前视口的“查看”模式。
- VIEWCTR：当前视口中视图的中心点（UCS 坐标）。
- LENSLENGTH：当前视口透视图中的镜头焦距长度（单位为毫米）。
- TARGET：当前视口中目标点的位置（以 UCS 坐标表示）。
- VIEWDIR：当前视口的观察方向（UCS）。
- VIEWSIZE：当前视口的视图高度（图形单位）。
- SCREENSIZE：以像素为单位的当前视口的大小（X 和 Y 值）。
- VIEWTWIST：当前视口的视图扭转角。
- TILEMODE：将模型选项卡或最后一个布局选项卡置为当前。
- CVPORT：当前视口的标识码。
- FRONTZ：当前视口中前向剪裁平面到目标平面的偏移量。
- BACKZ：获得当前视口后向剪裁平面到目标平面的偏移值。

在使用 setCenterPoint、setHeight 和 setWidth 等函数时需要注意，这些函数输入的参数是在 DCS（显示坐标系）中度量的，而从系统变量获取的参数值，可能是在 WCS、UCS 或者 DCS 中(在 AutoCAD 的帮助系统中有系统变量的参考)。因此，在某些场合中要使用 acedTrans 函数在各种坐标系之间转换坐标，该函数定义为：

```
int acedTrans(
        const ads_point pt,
        const struct resbuf * from,
        const struct resbuf * to,
        int disp,
        ads_point result);
```

pt 为输入点或向量；from 指定了表示 pt 参数所使用的坐标系类型；to 指定了转换结果 result 所使用的坐标系类型；disp 用来指定 pt 的类型，如果 disp 被设置为0，pt 被作为一个点，否则（disp 设置为非0）pt 被作为一个向量。

from 和 to 都是结果缓冲区变量，其如何代表坐标系类型呢？from 和 to 参数的使用可以遵循下面的规则：

- 如果结果类型码为整形（restype == RTSHORT），返回值为 0（rint == 0）表示 WCS，值为 1 表示 UCS，值为 2 表示 DCS。
- 如果结果类型码为实体名（restype == RTENAME），返回值为一个实体名或者选择集的名称，表示 ECS（实体坐标系）。
- 如果结果类型码为一个三维挤压向量（extrusion vector）（restype == RT3DPOINT），同样表示 ECS。

更新视图可以使用全局函数 acedSetCurrentView，该函数定义为：

```
Acad::ErrorStatus acedSetCurrentView(
                    AcDbViewTableRecord * pVwRec,
                    AcDbViewport * pVP);
```

pVwRec 指定了要用于更新的视图对象；pVP 指定了要更新的视口，如果输入 NULL 表示更新对象为当前视口。

4.4.3 步骤

（1）在 VC++ 6.0中，使用 ObjectARX 向导创建一个新项目，命名为 ChangeView。注册一个新命令 ZoomScale，用于实现视图的比例缩放，其实现函数为：

```
void ZffCHAP4ZoomScale()
{
        // 提示用户输入缩放的比例因子
        ads_real scale;
        if (acedGetReal("\n输入缩放比例因子：", &scale) != RTNORM)
                return;

        // 获得当前视图
        AcDbViewTableRecord view = GetCurrentView();

        // 修改视图
        view.setWidth(view.width() / scale);
        view.setHeight(view.height() / scale);

        // 更新视图
        acedSetCurrentView(&view, NULL);
}
```

其中，GetCurrentView 是一个自定义函数，用于获得当前的视图，该函数保存在

ChangeViewCommands.cpp 文件中，位于所有命令实现函数的前面，这样才能保证在命令实现函数中直接调用该函数。当然，按照 C 语言的语法，可以在调用该函数之前声明它。

GetCurrentView 函数的实现代码为：

```
AcDbViewTableRecord GetCurrentView()
{
        AcDbViewTableRecord view;
        struct resbuf rb;
        struct resbuf wcs, ucs, dcs;  // 转换坐标时使用的坐标系统标记

        wcs.restype = RTSHORT;
        wcs.resval.rint = 0;
        ucs.restype = RTSHORT;
        ucs.resval.rint = 1;
        dcs.restype = RTSHORT;
        dcs.resval.rint = 2;

        // 获得当前视口的“查看”模式
        acedGetVar("VIEWMODE", &rb);
        view.setPerspectiveEnabled(rb.resval.rint & 1);
        view.setFrontClipEnabled(rb.resval.rint & 2);
        view.setBackClipEnabled(rb.resval.rint & 4);
        view.setFrontClipAtEye(!(rb.resval.rint & 16));

        // 当前视口中视图的中心点（UCS坐标）
        acedGetVar("VIEWCTR", &rb);
        acedTrans(rb.resval.rpoint, &ucs, &dcs, 0, rb.resval.rpoint);
        view.setCenterPoint(AcGePoint2d(rb.resval.rpoint[X],
                rb.resval.rpoint[Y]));

        // 当前视口透视图中的镜头焦距长度（单位为毫米）
        acedGetVar("LENSLENGTH", &rb);
        view.setLensLength(rb.resval.rreal);

        // 当前视口中目标点的位置（以 UCS 坐标表示）
        acedGetVar("TARGET", &rb);
        acedTrans(rb.resval.rpoint, &ucs, &wcs, 0, rb.resval.rpoint);
        view.setTarget(AcGePoint3d(rb.resval.rpoint[X],
                rb.resval.rpoint[Y], rb.resval.rpoint[Z]));
```

```
// 当前视口的观察方向（UCS）
acedGetVar("VIEWDIR", &rb);
acedTrans(rb.resval.rpoint, &ucs, &wcs, 1, rb.resval.rpoint);
view.setViewDirection(AcGeVector3d(rb.resval.rpoint[X],
        rb.resval.rpoint[Y], rb.resval.rpoint[Z]));

// 当前视口的视图高度（图形单位）
acedGetVar("VIEWSIZE", &rb);
view.setHeight(rb.resval.rreal);
double height = rb.resval.rreal;

// 以像素为单位的当前视口的大小（X 和 Y 值）
acedGetVar("SCREENSIZE", &rb);
view.setWidth(rb.resval.rpoint[X] / rb.resval.rpoint[Y] * height);

// 当前视口的视图扭转角
acedGetVar("VIEWTWIST", &rb);
view.setViewTwist(rb.resval.rreal);

// 将模型选项卡或最后一个布局选项卡置为当前
acedGetVar("TILEMODE", &rb);
int tileMode = rb.resval.rint;
// 设置当前视口的标识码
acedGetVar("CVPORT", &rb);
int cvport = rb.resval.rint;

// 是否是模型空间的视图
bool paperspace = ((tileMode == 0) && (cvport == 1)) ? true : false;
view.setIsPaperspaceView(paperspace);

if (!paperspace)
{
        // 当前视口中前向剪裁平面到目标平面的偏移量
        acedGetVar("FRONTZ", &rb);
        view.setFrontClipDistance(rb.resval.rreal);

        // 获得当前视口后向剪裁平面到目标平面的偏移值
        acedGetVar("BACKZ", &rb);
        view.setBackClipDistance(rb.resval.rreal);
```

```
        }
        else
        {
                view.setFrontClipDistance(0.0);
                view.setBackClipDistance(0.0);
        }

        return view;
}
```

初学者通常会认为存在一个与 acedSetCurrentView 对应的库函数，用于获得当前视图，但是在 ObjectARX 中确实没有这样的函数，因此获得当前视图必须从系统变量中获得视图的参数，构建这样的视图对象。

（2）注册一个命令 ZoomWindow，用于模拟窗口缩放的功能，其实现函数为：

```
void ZffCHAP4ZoomWindow()
{
        // 提示用户选择定义缩放窗口的两个角点
        ads_point pt1, pt2;
        if (acedGetPoint(NULL, "\n输入第一个角点：", pt1) != RTNORM)
                return;
        if (acedGetCorner(pt1, "\n输入第二个角点：", pt2) != RTNORM)
                return;

        struct resbuf wcs, ucs, dcs; // 转换坐标时使用的坐标系统标记
        wcs.restype = RTSHORT;
        wcs.resval.rint = 0;
        ucs.restype = RTSHORT;
        ucs.resval.rint = 1;
        dcs.restype = RTSHORT;
        dcs.resval.rint = 2;

        acedTrans(pt1, &ucs, &dcs, 0, pt1);
        acedTrans(pt2, &ucs, &dcs, 0, pt2);

        AcDbViewTableRecord view = GetCurrentView();

        // 设置视图的中心点
        view.setCenterPoint(AcGePoint2d((pt1[X] + pt2[X]) / 2,
                (pt1[Y] + pt2[Y]) / 2));
```

```
                    // 设置视图的高度和宽度
                    view.setHeight(fabs(pt1[Y] - pt2[Y]));
                    view.setWidth(fabs(pt1[X] - pt2[X]));

                    // 将视图对象设置为当前视图
                    acedSetCurrentView(&view, NULL);
        }
```

（3）注册新命令 ZoomExtents，用于实现在各种视图（包括透视模式）中的范围缩放，其实现函数为：

```
        void ZffCHAP4ZoomExtents()
        {
                    AcDbBlockTable *pBlkTbl;
                    AcDbBlockTableRecord *pBlkTblRcd;
                    acdbHostApplicationServices()->workingDatabase()
                            ->getBlockTable(pBlkTbl, AcDb::kForRead);
                    pBlkTbl->getAt(ACDB_MODEL_SPACE, pBlkTblRcd,
AcDb::kForRead);
                    pBlkTbl->close();

                    // 获得当前图形中所有实体的最小包围盒
                    AcDbExtents extent;
                    extent.addBlockExt(pBlkTblRcd);
                    pBlkTblRcd->close();

                    // 计算长方形的顶点
                    ads_point pt[7];
                    pt[0][X] = pt[3][X] = pt[4][X] = pt[7][X] = extent.minPoint().x;
                    pt[1][X] = pt[2][X] = pt[5][X] = pt[6][X] = extent.maxPoint().x;
                    pt[0][Y] = pt[1][Y] = pt[4][Y] = pt[5][Y] = extent.minPoint().y;
                    pt[2][Y] = pt[3][Y] = pt[6][Y] = pt[7][Y] = extent.maxPoint().y;
                    pt[0][Z] = pt[1][Z] = pt[2][Z] = pt[3][Z] = extent.maxPoint().z;
                    pt[4][Z] = pt[5][Z] = pt[6][Z] = pt[7][Z] = extent.minPoint().z;

                    // 将长方体的所有角点转移到DCS中
                    struct resbuf wcs, dcs; // 转换坐标时使用的坐标系统标记
                    wcs.restype = RTSHORT;
                    wcs.resval.rint = 0;
                    dcs.restype = RTSHORT;
                    dcs.resval.rint = 2;
```

```
        acedTrans(pt[0], &wcs, &dcs, 0, pt[0]);
        acedTrans(pt[1], &wcs, &dcs, 0, pt[1]);
        acedTrans(pt[2], &wcs, &dcs, 0, pt[2]);
        acedTrans(pt[3], &wcs, &dcs, 0, pt[3]);
        acedTrans(pt[4], &wcs, &dcs, 0, pt[4]);
        acedTrans(pt[5], &wcs, &dcs, 0, pt[5]);
        acedTrans(pt[6], &wcs, &dcs, 0, pt[6]);
        acedTrans(pt[7], &wcs, &dcs, 0, pt[7]);

        // 获得所有角点在DCS中最小的包围矩形
        double xMax = pt[0][X], xMin = pt[0][X];
        double yMax = pt[0][Y], yMin = pt[0][Y];
        for (int i = 1; i <= 7; i++)
        {
            if (pt[i][X] > xMax)
                xMax = pt[i][X];
            if (pt[i][X] < xMin)
                xMin = pt[i][X];
            if (pt[i][Y] > yMax)
                yMax = pt[i][Y];
            if (pt[i][Y] < yMin)
                yMin = pt[i][Y];
        }

        AcDbViewTableRecord view = GetCurrentView();

        // 设置视图的中心点
        view.setCenterPoint(AcGePoint2d((xMin + xMax) / 2,
            (yMin + yMax) / 2));

        // 设置视图的高度和宽度
        view.setHeight(fabs(yMax - yMin));
        view.setWidth(fabs(xMax - xMin));

        // 将视图对象设置为当前视图
        Acad::ErrorStatus es = acedSetCurrentView(&view, NULL);
    }
```

为了获得当前模型空间所有实体的最小包围盒，使用了 AcDbExtents 类的 addBlockExt 函数。由于包围盒的尺寸和位置是在 WCS 中表示的，因此需要先获得8个角点的坐标，将其

转换到 DCS 中，然后再计算最小的包围矩形，最终将包围矩形作为视图显示的范围。

4.4.4 效果

（1）编译运行程序，在AutoCAD 2002中，创建一个长方体，选择【视图 / 三维动态观察器】菜单项，使用三维动态观察器调整视图，得到如图4.11所示的结果。

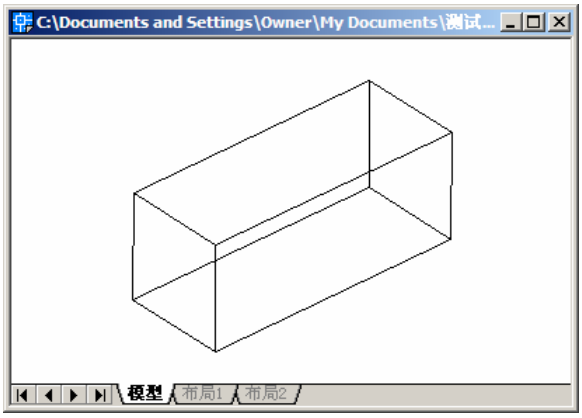

图4.11　初始的视图

（2）执行ZoomScale命令，输入2作为缩放比例因子，也就是将视图放大2倍，能够得到如图4.12所示的结果。

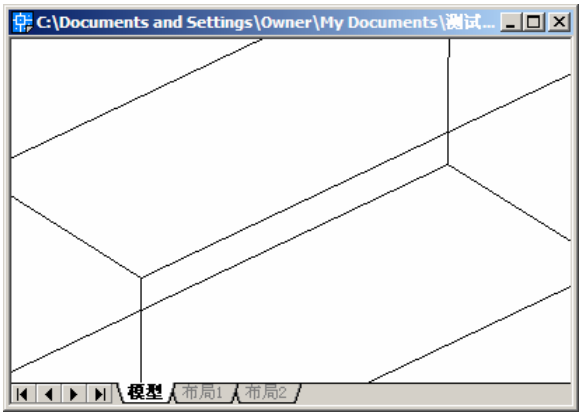

图4.12　将视图放大 2 倍

（3）执行ZoomWindow命令，拾取两点来定义缩放窗口，能够得到如图4.13所示的结果。

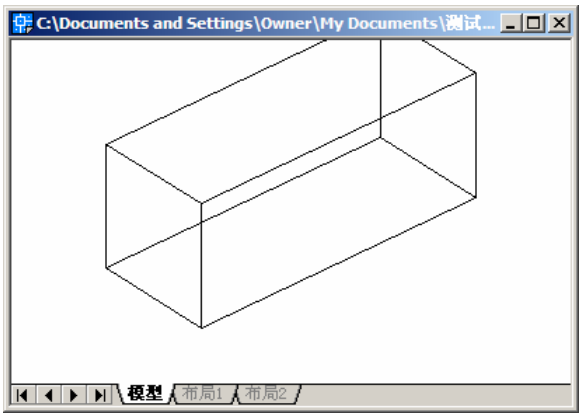

图4.13 窗口缩放的结果

（4）执行ZoomExtents命令，实现范围缩放，能够得到如图4.14所示的结果。如果执行ZOOM命令，并选择E选项（Extents），将会得到同样的结果。

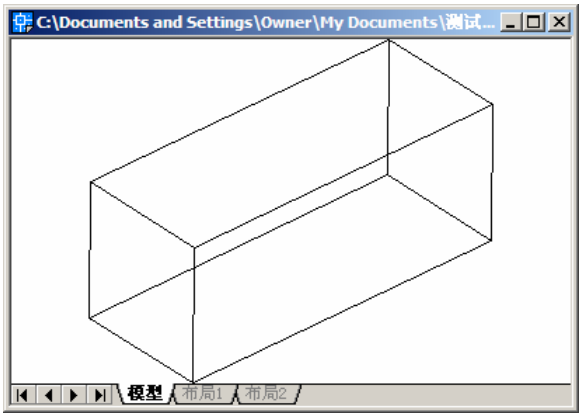

图4.14 范围缩放的结果

4.4.5 小结

学习本节内容之后，读者需要掌握下面的要点：

- 坐标系之间的转换。
- 获得若干个实体的包围盒。使用 AcDbEntity 类的 getGeomExtents 函数获得一个实体的包围盒，然后使用 AcDbExtents 类的 addExt 函数将该实体的包围盒与已有的包围盒合并。
- 获得当前视图。
- 调整视图的方法。
- 通过 AutoCAD 的系统变量获得某些特性。

4.5 视口

4.5.1 说明

本节的实例演示在模型空间或者图纸空间创建4个等大的视口，并使用 ADS 的方法将4个视口应用到当前的工作空间。

4.5.2 思路

在 ObjectARX 中有两个代表视口的类：AcDbViewportTableRecord 和 AcDbViewport。其中，AcDbViewportTableRecord 类表示模型空间的视口（在 AutoCAD 中称为平铺视口），AcDbViewport 类则表示图纸空间的视口（在 AutoCAD 中称为浮动视口）。

在模型空间中创建平铺视口，与创建一个普通的符号表记录并没有太多的不同，其步骤为：

（1）创建一个视口表记录，使用 setLowerLeftCorner 和 setUpperRightCorner 函数设置视口的角点，使用 setName 函数设置视口的名称。与其他的符号表记录不同，视口表记录的名称可以相同，相同名称的视口表记录被作为一组记录。当用户要求显示某个名称的视口时，同组的所有视口都会被显示出来，这就解释了实现 4 个视口的方法。

（2）获得当前图形的视口表，将创建的视口表记录添加到视口表中。

（3）关闭视口表记录和视口表。

在图纸空间创建浮动视口，除了可以采用上面的方法之外，还可以直接向图纸空间添加 AcDbViewport 类的对象。

4.5.3 步骤

（1）在 VC++ 6.0中，使用 ObjectARX 向导新建一个项目，命名为 Viewports。注册一个新命令 Create4VPorts，用于创建4个等大的视口，其实现函数为：

```
void ZffCHAP4Create4VPorts()
{
        // 获得视口表
        AcDbViewportTable *pVPortTbl;
        acdbHostApplicationServices()->workingDatabase()
                ->getViewportTable(pVPortTbl, AcDb::kForWrite);

        // 分别创建四个视口
        AcGePoint2d pt1, pt2;
        AcDbViewportTableRecord *pVPortTblRcd1=new
AcDbViewportTableRecord;
        pt1.set(0, 0);
```

```
                pt2.set(0.5, 0.5);
                pVPortTblRcd1->setLowerLeftCorner(pt1);
                pVPortTblRcd1->setUpperRightCorner(pt2);
                pVPortTblRcd1->setName("4VPorts");

                AcDbViewportTableRecord *pVPortTblRcd2=new
AcDbViewportTableRecord;
                pt1.set(0.5, 0);
                pt2.set(1, 0.5);
                pVPortTblRcd2->setLowerLeftCorner(pt1);
                pVPortTblRcd2->setUpperRightCorner(pt2);
                pVPortTblRcd2->setName("4VPorts");

                AcDbViewportTableRecord *pVPortTblRcd3=new
AcDbViewportTableRecord;
                pt1.set(0, 0.5);
                pt2.set(0.5, 1);
                pVPortTblRcd3->setLowerLeftCorner(pt1);
                pVPortTblRcd3->setUpperRightCorner(pt2);
                pVPortTblRcd3->setName("4VPorts");

                AcDbViewportTableRecord *pVPortTblRcd4=new
AcDbViewportTableRecord;
                pt1.set(0.5, 0.5);
                pt2.set(1, 1);
                pVPortTblRcd4->setLowerLeftCorner(pt1);
                pVPortTblRcd4->setUpperRightCorner(pt2);
                pVPortTblRcd4->setName("4VPorts");

                pVPortTbl->add(pVPortTblRcd1);
                pVPortTbl->add(pVPortTblRcd2);
                pVPortTbl->add(pVPortTblRcd3);
                pVPortTbl->add(pVPortTblRcd4);

                pVPortTbl->close();
                pVPortTblRcd1->close();
                pVPortTblRcd2->close();
                pVPortTblRcd3->close();
                pVPortTblRcd4->close();
```

```
        // 判断当前的空间
        struct resbuf rb;
        acedGetVar("TILEMODE", &rb);
        if (rb.resval.rint == 1)          // 当前工作空间是模型空间
        {
            acedCommand(RTSTR, ".-VPORTS", RTSTR, "R",
                RTSTR, "4VPorts", RTNONE);
        }
        else                              // 当前工作空间是图纸空间
        {
            acedCommand(RTSTR, ".-VPORTS", RTSTR, "R",
                RTSTR, "4VPorts", RTSTR, "", RTNONE);
        }
    }
```

（2）注册一个新命令 CreateVPortInSpace，用于在图纸空间创建一个新的视口，这次直接向图纸空间的块表记录添加一个 AcDbViewport 对象，其实现函数为：

```
    void ZffCHAP4CreateVPortInSpace()
    {
        // 指定当前布局
        Acad::ErrorStatus es =
acdbHostApplicationServices()->layoutManager()
            ->setCurrentLayout("布局1");
        if (es != Acad::eOk)
        {
            return;
        }

        // 获得块表
        AcDbBlockTable *pBlkTbl;
        acdbHostApplicationServices()->workingDatabase()
            ->getBlockTable(pBlkTbl, AcDb::kForRead);

        // 获得图纸空间的块表记录
        AcDbBlockTableRecord *pBlkTblRcd;
        pBlkTbl->getAt(ACDB_PAPER_SPACE, pBlkTblRcd,
AcDb::kForWrite);
        pBlkTbl->close();
```

```
    // 创建新的布局对象
    AcDbViewport *pViewport = new AcDbViewport();
    pViewport->setCenterPoint(AcGePoint3d(100, 50, 0));
    pViewport->setHeight(80);
    pViewport->setWidth(120);

    // 将新的布局对象添加到图纸空间块表记录中
    AcDbObjectId viewportId;
    pBlkTblRcd->appendAcDbEntity(viewportId, pViewport);

    pViewport->close();
    pBlkTblRcd->close();

    // 将新建的视口作为当前视口
    AcDbViewport *pVP;
    acdbOpenObject(pVP, viewportId, AcDb::kForWrite);
    pViewport->setOn();
    acedSetCurrentVPort(pVP);
    pVP->close();
}
```

创建浮动视口之后，要想启用该视口，必须使用 setOn 函数。

4.5.4 效果

（1）编译运行程序，在AutoCAD 2002中执行Create4VPorts命令，能够得到如图4.15所示的结果。

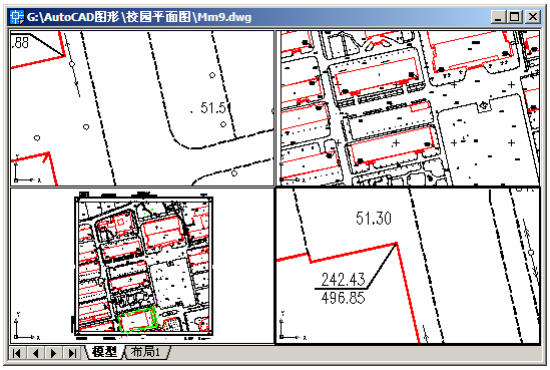

图4.15 创建四个等大的视口

（2）执行CreateVPortInSpace命令，能够在图纸空间创建一个视口，如图4.16所示。

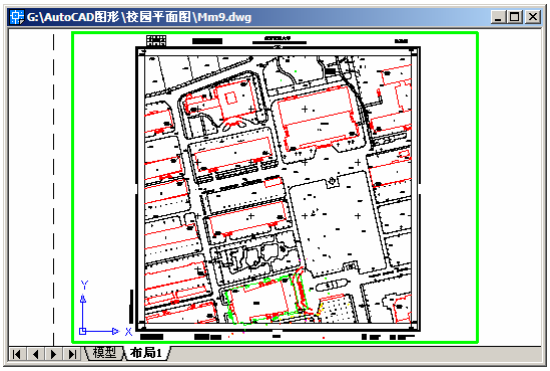

图4.16　在图纸空间创建一个视口

4.5.5 小结

学习本节内容之后，读者应该掌握下面的要点：

- 判断当前的工作空间。
- 选择一个布局作为当前布局。
- 设置当前视口。

4.6 UCS

4.6.1 说明

在 AutoCAD 中，用户可以通过创建 UCS 来不断调整图形的参考系，简化绘图中计算的繁琐，这在三维建模时最为普遍。本节的实例演示了创建新的 UCS、应用已有的 UCS、移动 UCS 的原点、旋转 UCS，以及在 UCS 中创建图形对象（ObjectARX 中创建各种实体均在 WCS 中）。

4.6.2 思路

在 ObjectARX 中创建 UCS 的方法与 AutoCAD 应用中三点法创建 UCS 类似，由原点、X 轴和 Y 轴方向来决定 UCS 的位置，AcDbUCSTableRecord 类的 setOrigin、setXAxis 和 setYAxis 三个函数分别用来设置原点、X 轴和 Y 轴的方向。

全局函数 acedSetCurrentUCS 用于设置当前 UCS，该函数接受一个 AcGeMatrix3d 类型的参数。该参数是一个几何变换矩阵，定义了 UCS 到 WCS 转换的对应关系。要将一个 UCS 设置为当前 UCS，必须首先获得其相对于 WCS 的变换矩阵，然后使用 acedSetCurrentUCS 函数。

ObjectARX中创建实体时指定的坐标必须是WCS中的坐标，但是很多情况下需要根据

UCS中的坐标来创建实体，这就必须使用acedTrans函数将UCS中的坐标转换到WCS中，关于该函数的详细介绍参考4.4 节的内容。

4.6.3 步骤

（1）在 VC++ 6.0中，使用 ObjectARX 向导创建一个新的项目，命名为 OperateUCS。注册一个命令 NewUcs，创建一个新的 UCS，并将其添加到 UCS 表中，其实现函数为：

```
void ZffCHAP4NewUCS()
{
        // 获得当前图形的UCS表
        AcDbUCSTable *pUcsTbl;
        acdbHostApplicationServices()->workingDatabase()
                ->getUCSTable(pUcsTbl, AcDb::kForWrite);

        // 定义UCS的参数
        AcGePoint3d ptOrigin(0, 0, 0);
        AcGeVector3d vecXAxis(1, 1, 0);
        AcGeVector3d vecYAxis(-1, 1, 0);

        // 创建新的UCS表记录
        AcDbUCSTableRecord *pUcsTblRcd = new AcDbUCSTableRecord();

        // 设置UCS的参数
        Acad::ErrorStatus es = pUcsTblRcd->setName("NewUcs");
        if (es != Acad::eOk)
        {
                delete pUcsTblRcd;
                pUcsTbl->close();
                return;
        }
        pUcsTblRcd->setOrigin(ptOrigin);
        pUcsTblRcd->setXAxis(vecXAxis);
        pUcsTblRcd->setYAxis(vecYAxis);

        // 将新建的UCS表记录添加到UCS表中
        es = pUcsTbl->add(pUcsTblRcd);
        if (es != Acad::eOk)
        {
                delete pUcsTblRcd;
                pUcsTbl->close();
```

```
            return;
        }

        // 关闭对象
        pUcsTblRcd->close();
        pUcsTbl->close();
    }
```

（2）注册一个命令 SetCurUcs，将 UCS 表中已经存在的一个 UCS 设置为当前 UCS，其实现函数为：

```
    void ZffCHAP4SetCurUcs()
    {
        // 提示用户输入UCS的名称
        char ucsName[40];
        if (acedGetString(NULL, "\n输入用户坐标系的名称：", ucsName) !=
RTNORM)
            return;

        // 获得指定的UCS表记录
        AcDbUCSTable *pUcsTbl;
        acdbHostApplicationServices()->workingDatabase()
            ->getUCSTable(pUcsTbl, AcDb::kForRead);
        if (!pUcsTbl->has(ucsName))
        {
            pUcsTbl->close();
            return;
        }
        AcDbUCSTableRecord *pUcsTblRcd;
        pUcsTbl->getAt(ucsName, pUcsTblRcd, AcDb::kForRead);

        // 获得UCS的变换矩阵
        AcGeMatrix3d mat;
        AcGeVector3d vecXAxis, vecYAxis, vecZAxis;
        vecXAxis = pUcsTblRcd->xAxis();
        vecYAxis = pUcsTblRcd->yAxis();
        vecZAxis = vecXAxis.crossProduct(vecYAxis);
        mat.setCoordSystem(pUcsTblRcd->origin(), vecXAxis,
            vecYAxis, vecZAxis);

        // 关闭UCS表和UCS表记录
```

```
        pUcsTblRcd->close();
        pUcsTbl->close();

        // 设置当前的UCS
        acedSetCurrentUCS(mat);
}
```

在 AcDbUCSTableRecord 类仅保存了 UCS 的原点位置、X 轴和 Y 轴方向，但是建立 UCS 的变换矩阵需要获得其 Z 轴方向，这就要使用 AcGeVector3d 类的 crossProduct 函数。crossProduct 函数能够返回两个矢量的向量积，这样就能根据 UCS 的 X、Y 轴方向获得 Z 方向。

（3）注册一个命令 MoveUcsOrigin，用于移动当前 UCS 的原点，其实现函数为：

```
void ZffCHAP4MoveUcsOrigin()
{
        // 获得当前UCS的变换矩阵
        AcGeMatrix3d mat;
        Acad::ErrorStatus es = acedGetCurrentUCS(mat);

        // 根据变换矩阵获得UCS的参数
        AcGePoint3d ptOrigin;
        AcGeVector3d vecXAxis, vecYAxis, vecZAxis;
        mat.getCoordSystem(ptOrigin, vecXAxis, vecYAxis, vecZAxis);

        // 移动UCS的原点
        AcGeVector3d vec(100, 100, 0);
        ptOrigin += vec;

        // 更新变换矩阵
        mat.setCoordSystem(ptOrigin, vecXAxis, vecYAxis, vecZAxis);

        // 应用新的UCS
        acedSetCurrentUCS(mat);
}
```

（4）注册一个命令 RotateUcs，用于将当前 UCS 绕 Z 轴旋转60°，其实现函数为：

```
void ZffCHAP4RotateUcs()
{
        // 获得当前UCS的变换矩阵
        AcGeMatrix3d mat;
        Acad::ErrorStatus es = acedGetCurrentUCS(mat);
```

```
        // 根据变换矩阵获得UCS的参数
        AcGePoint3d ptOrigin;
        AcGeVector3d vecXAxis, vecYAxis, vecZAxis;
        mat.getCoordSystem(ptOrigin, vecXAxis, vecYAxis, vecZAxis);

        // 绕Z轴旋转60度
        vecXAxis.rotateBy(60 * atan(1) * 4 / 180, vecZAxis);
        vecYAxis.rotateBy(60 * atan(1) * 4 / 180, vecZAxis);

        // 更新变换矩阵
        mat.setCoordSystem(ptOrigin, vecXAxis, vecYAxis, vecZAxis);

        // 应用新的UCS
        acedSetCurrentUCS(mat);
    }
```

（5）注册一个命令 AddEntInUcs，用于根据 UCS 中的坐标创建实体，其实现函数为：

```
    void ZffCHAP4AddEntInUcs()
    {
        // 转换坐标系的标记
        struct resbuf wcs, ucs;
        wcs.restype = RTSHORT;
        wcs.resval.rint = 0;
        ucs.restype = RTSHORT;
        ucs.resval.rint = 1;

        // 提示用户输入直线的起点和终点
        ads_point pt1, pt2;
        if (acedGetPoint(NULL, "拾取直线的起点：", pt1) != RTNORM)
            return;
        if (acedGetPoint(pt1, "拾取直线的终点：", pt2) != RTNORM)
            return;

        // 将起点和终点坐标转换到WCS
        acedTrans(pt1, &ucs, &wcs, 0, pt1);
        acedTrans(pt2, &ucs, &wcs, 0, pt2);

        // 创建直线
        AcDbLine *pLine = new AcDbLine(asPnt3d(pt1), asPnt3d(pt2));
```

```
                    AcDbBlockTable *pBlkTbl;
                    acdbHostApplicationServices()->workingDatabase()
                            ->getBlockTable(pBlkTbl, AcDb::kForRead);

                    AcDbBlockTableRecord *pBlkTblRcd;
                    pBlkTbl->getAt(ACDB_MODEL_SPACE, pBlkTblRcd,
AcDb::kForWrite);
                    pBlkTbl->close();

                    pBlkTblRcd->appendAcDbEntity(pLine);

                    pLine->close();
                    pBlkTblRcd->close();
        }
```

4.6.4 效果

（1）编译运行程序，在 AutoCAD 2002中执行 NewUcs 命令，在图形中创建一个新的UCS。执行 UCS 命令，按照命令行提示进行操作：

```
命令: ucs
当前 UCS 名称: *世界*
输入选项 [新建(N)/移动(M)/正交(G)/上一个(P)/恢复(R)/保存(S)/删除(D)/应用(A)/?/世界(W)] <世界>: ?          〖查询保存的UCS〗
输入要列出的 UCS 名称 <*>:    〖直接按下Enter键〗
当前 UCS 名称: *世界*
已保存的坐标系:

"NewUcs"
    原点 = <0.0000，0.0000，0.0000>，X 轴 = <0.7071，0.7071，0.0000>
    Y 轴 = <-0.7071，0.7071，0.0000>，Z 轴 = <0.0000，0.0000，1.0000>

输入选项 [新建(N)/移动(M)/正交(G)/上一个(P)/恢复(R)/保存(S)/删除(D)/应用(A)/?/世界(W)] <世界>: *取消*
```

（2）执行 SetCurUcs 命令，输入 NewUcs 作为 UCS 名称，将其设置为当前 UCS。观察结果之后，使用 UCS 命令将当前 UCS 设置为 WCS（执行 UCS 命令后直接按下 Enter 键）。

（3）执行 MoveUcsOrigin 命令，能够将当前 UCS 的原点移动到（100，100，0），其坐标轴的方向保持不变。执行 Rotate 命令，能够将当前 UCS 绕 Z 轴旋转60°。

（4）执行 AddEntInUcs 命令，拾取两点作为直线的起点和终点，能够创建一条直线。为了对比效果，可以将 AddEntInUcs 命令的实现函数中的下面两条语句注释掉，再此执行程序对比结果（确保当前 UCS 不与 WCS 重合）：

```
// 将起点和终点坐标转换到WCS
acedTrans(pt1, &ucs, &wcs, 0, pt1);
acedTrans(pt2, &ucs, &wcs, 0, pt2);
```

4.6.5 小结

学习本节内容之后，读者需要掌握下面的知识点：

- UCS 和 WCS 之间依靠变换矩阵建立联系。
- 获得当前 UCS 的参数。
- 修改当前 UCS。

第5章 使用 ADSRX

ADS 是 AutoCAD 开发系统的简称，从 AutoCAD R10 版时 Autodesk 开始提供这种开发方式（实际上后来演变成现在的 ObjectARX），它是一个带有 Autodesk 提供的库文件和头文件的 C 语言编程环境。

新的 ObjectARX 出现之后，保留了一部分 ADS 的内容，这部分内容经过修改后被称为 ADSRX，在 ObjectARX 用于完成一些特定的功能：

- 获取用户输入。
- 创建和使用 AutoCAD 的选择集。
- 使用 DCL 对话框（由于 DCL 对话框的使用不够方便，与 MFC 用户界面相比功能又显得很弱，本书不再讨论，仅讨论 MFC 用户界面）。

5.1 acedCommand 函数和结果缓冲区

5.1.1 说明

在 ObjectARX 编程中，可以使用 acedCommand 或 acedCmd 函数来执行 AutoCAD 内部的命令，这与在 AutoCAD 中直接执行相应命令的效果是一样的（甚至在命令窗口中都能找到执行命令的痕迹），本节通过创建圆、对图形进行缩放来说明这两个函数的使用。

结果缓冲区作为一种特殊的数据类型，在某些特定的场合仍有不可替代的作用，因此在实例中通过创建和访问结果缓冲区的内容来帮助读者更号地理解结果缓冲区的结构。

本节所得到的 ARX 文件向 AutoCAD 注册三个命令，AddCircle1和 AddCircle2命令能够在图形窗口中创建圆，EntInfo 命令能够在命令窗口中显示所选择的圆的参数。

5.1.2 思路

1. acedCommand函数

acedCommand 函数的定义为：

```
int acedCommand(int rtype, ... unnamed);
```

该函数的参数个数是可变的，并且参数成对出现。参数对中第一个参数表示参数的类型（参考ARX帮助系统中如图5.1所示的列表），第二个表示其实际的数据。参数表的最后一个参数必须是0或者RTNONE（使用RTNONE更好一些）。

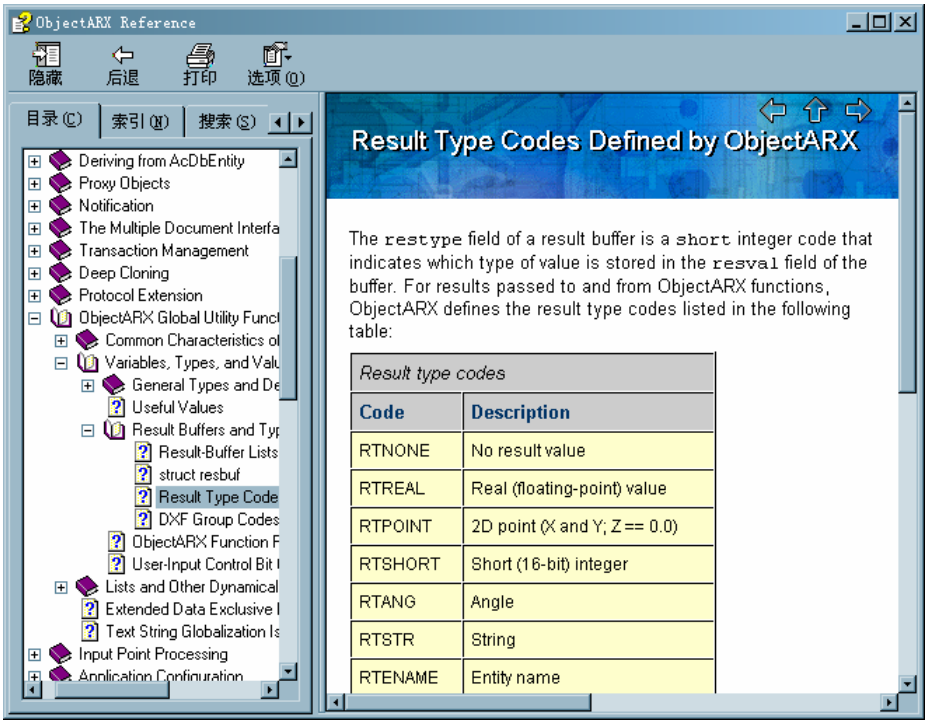

图5.1　可以在 acedCommand 函数中使用的参数类型

下面的代码用于在 AutoCAD 中创建一个圆心为（0，0）、半径为10的圆：

```
acedCommand(RTSTR, "Circle",      // 命令
            RTSTR, “0,0,0”,       // 圆心
            RTSTR, “10”,          // 半径
            RTNONE);              // 结束命令
```

在 AddCircle1命令的实现函数中，使用变量值作为 acedCommand 函数的参数，所实现的功能与上面的代码完全一样。

2. acedCmd函数

acedCmd 函数的定义为：

```
int acedCmd(const struct resbuf * rbp);
```

该函数的参数是一个 resbuf 类型的指针，这里需要的结果缓冲区可以由 acutBuildList 函数生成。由于 acedCommand 函数实质上也是为要执行的命令构造了一个 resbuf 结构，因此 acedCmd 函数和 acedCommand 函数完全能够实现相同的功能，在 AddCircle2命令的实现函数中演示了 acedCmd 函数的用法。

3. 结果缓冲区（resbuf）

结果缓冲区（resbuf）是 ObjectARX 中定义的一个结构体，其定义为：

```
struct resbuf {
    struct resbuf *rbnext; // 连接列表的指针
```

```
        short restype;
        union ads_u_val resval;
    };
```

其中，联合体 ads_u_val 的定义为：

```
    union ads_u_val {
        ads_real rreal;
        ads_real rpoint[3];
        short rint; // 必须声明为short，而不是int
        char *rstring;
        long rlname[2];
        long rlong;
        struct ads_binary rbinary;
    };
```

resbuf 结构体中的 rbnext 指针可以将多个结果缓冲区连接成一个单链表（很可能你和我一样，并不是计算机专业科班出身，如果你对链表的概念感到模糊，那肯定会影响到你对结果缓冲区链表的理解，请花一个上午的时间阅读数据结构的书籍，很快你就能对此烂熟于胸），当 rbnext 等于 NULL 时表示到达了链表的末尾。在访问结果缓冲区的内容时，通常定义两个 resbuf 指针，一个指向链表的开头（以备在使用完毕后用 acutRelRb 函数释放结果缓冲区的存储空间），另一个用于遍历链表，在 EntInfo 命令的实现函数中详细演示了这种用法。

restype变量用于指定联合体ads_u_val中保存的变量的类型，其取值可以是图5.1中的列表中的任何一种。

结果缓冲区链表的结构可以用图5.2来表示。

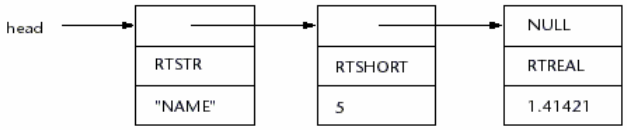

图5.2 结果缓冲区链表的结构

与结果缓冲区相关的两个全局函数：

- acutRelRb：释放结果缓冲区链表的存储空间。
- acutNewRb：创建一个新的结果缓冲区，并为其分配存储空间。使用该函数分配的存储空间必须在不用的时候用 acutRelRb 函数手工释放空间。

请务必搞明白结果缓冲区的概念和使用，不然后面学习扩展数据、字典时你就需要回来补课。

5.1.3 步骤

（1）运行 VC++ 6.0，使用 ObjectARX 向导创建一个新工程，名称为 Resbuf。注册一个新命令 AddCircle1，使用 acedCommand 函数创建一个圆，其实现函数为：

```
    void ZffCHAP5AddCircle1()
```

```
{
        // 声明ADS变量
        ads_point ptCenter = {0, 0, 0};          // 圆心
        ads_real radius = 10;                    // 半径

        // 调用acedCommand函数创建圆
        acedCommand(RTSTR, "Circle",      // 命令
                    RTPOINT, ptCenter, // 圆心
                    RTREAL, radius,          // 半径
                    RTNONE);          // 结束命令
}
```

在指定圆心时，使用（RTPOINT, ptCenter）和（RTSTR, “0,0,0”）的作用是相同的，但是前一种方式可以直接使用变量作为参数，灵活性较好。

（2）注册一个新命令 AddCircle2，使用 acedCmd 函数创建一个圆，其实现函数为：

```
void ZffCHAP5ADDCIRCLE2()
{
        struct resbuf *rb; // 结果缓冲区
        int rc = RTNORM; // 返回值

        // 创建结果缓冲区链表
        ads_point ptCenter = {30, 0, 0};
        ads_real radius = 10;
        rb = acutBuildList(RTSTR, "Circle",
                    RTPOINT, ptCenter,
                    RTREAL, radius,
                    RTNONE);

        // 创建圆
        if (rb != NULL)
        {
            rc = acedCmd(rb);
        }

        // 检验返回值
        if (rc != RTNORM)
        {
            acutPrintf("\n创建圆失败!");
        }
```

```
        acutRelRb(rb);

        // 进行缩放
        acedCommand(RTSTR, "Zoom", RTSTR, "E", RTNONE);
}
```

首先使用 acutBuildList 函数构造一个结果缓冲区，将其作为参数传递给 acedCmd 函数，这样的功能通过 acedCommand 函数同样可以实现。需要注意的是，acutBuildList 函数会构造一个结果缓冲区并自动分配内存，在不需要的时侯必须用 acutRelRb 函数释放分配的内容。

最后，使用 acedCommand 函数调用 AutoCAD 内部命令 Zoom 对图形窗口进行缩放操作。

（3）注册一个新命令 EntInfo，用于提示用户选择一个实体，在命令窗口中显示该实体的参数，其实现函数为：

```
void ZffCHAP5EntInfo()
{
        // 提示用户选择实体
        ads_name entName;
        ads_point pt;
        if (acedEntSel("\n选择实体:", entName, pt) != RTNORM)
                return;

        struct resbuf *rbEnt;     // 保存实体数据的结果缓冲区
        struct resbuf *rb;   // 用于遍历rbEnt的结果缓冲区

        // 从entName获得保存实体数据的结果缓冲区
        rbEnt = acdbEntGet(entName);
        rb = rbEnt;

        while (rb != NULL)
        {
                switch (rb->restype)
                {
                case -1:   // 图元名
                    acutPrintf("\n图元名: %x", rb->resval.rstring);
                    break;
                case 0:       // 图元类型
                    acutPrintf("\n图元类型: %s", rb->resval.rstring);
                    break;
                case 8:       // 图层
                    acutPrintf("\n图层:%s", rb->resval.rstring);
                    break;
```

```
                case 10:   // 圆心
                    acutPrintf("\n圆心:(%.2f, %.2f, %.2f)",
                            rb->resval.rpoint[X],
                            rb->resval.rpoint[Y],
                            rb->resval.rpoint[Z]);
                    break;
                case 40:   // 半径
                    acutPrintf("\n半径:%.4f", rb->resval.rreal);
                    break;
                case 210: // 圆所在平面的法向矢量
                    acutPrintf("\n平面的法向矢量:(%.2f, %.2f, %.2f)",
                            rb->resval.rpoint[X],
                            rb->resval.rpoint[Y],
                            rb->resval.rpoint[Z]);
                    break;
                default:
                    break;
                }    // switch

                rb = rb->rbnext;        // 切换到下一个节点
            }       // while

            if (rbEnt != NULL)
            {
                acutRelRb(rbEnt);
            }
        }
```

使用 acedEntSel 函数提示用户选择实体，将选择的结果保存在一个 ads_name 变量中，这在前面已经多次出现。全局函数 acdbEntGet 可以返回一个保存了实体定义数据的结果缓冲区链表，这里获得用户选择实体的结果缓冲区链表。

为了访问结果缓冲区链表rbEnt中保存的内容，定义了一个辅助的结果缓冲区链表指针rb，使用rb->restype判断当前的结果缓冲区保存数据的类型，在acutPrintf函数中用rb->resval获得其中保存的数据，将其输出到命令窗口中。其中，数据类型的代码是标准的DXF组码，可以在AutoCAD帮助系统中的acad_dev.chm文件中查看，如图5.3所示。

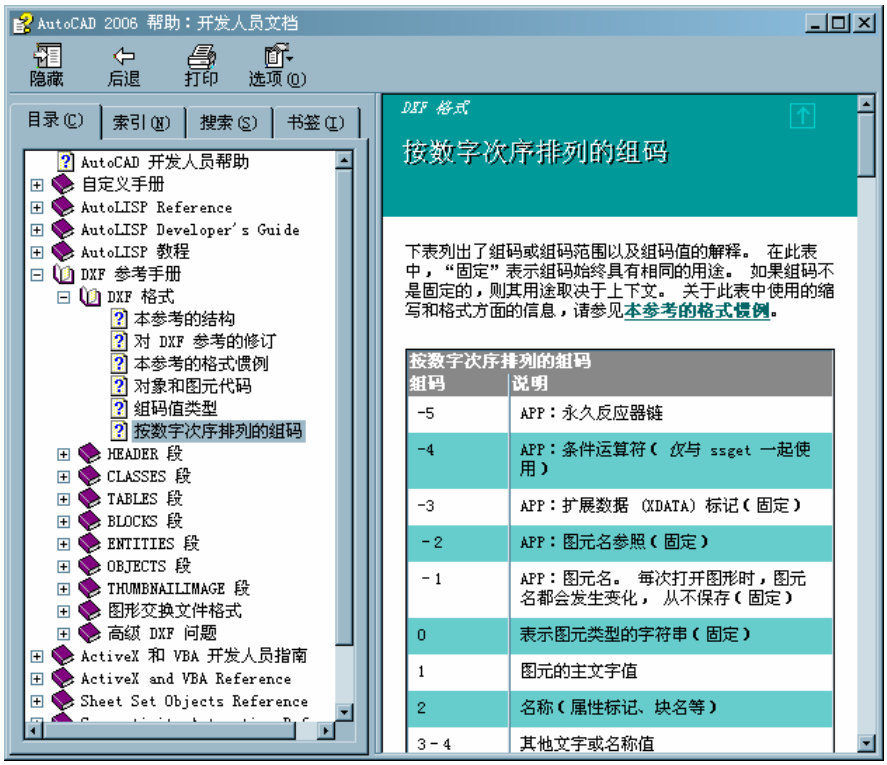

图5.3 查看 DXF 组码

每次显示结果缓冲区数据之后，使用 rb = rb->rbnext 使 rb 指向结果缓冲区链表的下一个节点。

最后，需要使用 acutRelRb 函数释放结果缓冲区链表的存储空间。试想，如果没有使用辅助的指针 rb 来遍历所有节点，而是直接使用 rbEnt 来访问，最后就无法通过 acutRelRb 函数释放结果缓冲区链表的存储空间（rbEnt 并未指向链表的头节点），造成内存泄漏。

5.1.4 效果

（1）编译连接程序，启动AutoCAD 2002，加载生成的ARX文件，执行AddCircle1和AddCircle2命令，系统会在图形窗口中创建两个圆。如果按下F2键显示命令窗口，能够发现其中包含了创建圆的AutoCAD命令提示语句，如图5.4所示。

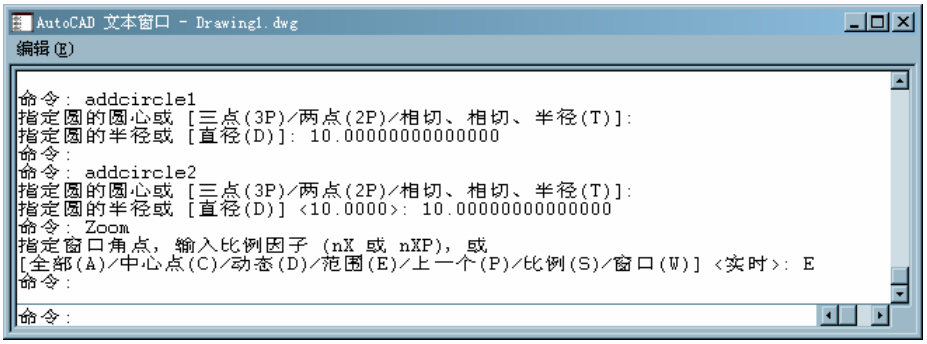

图5.4　查看命令窗口中的命令执行信息

（2）执行EntInfo命令，选择图形窗口中右侧的圆，能够在命令窗口得到如图5.5所示的结果。

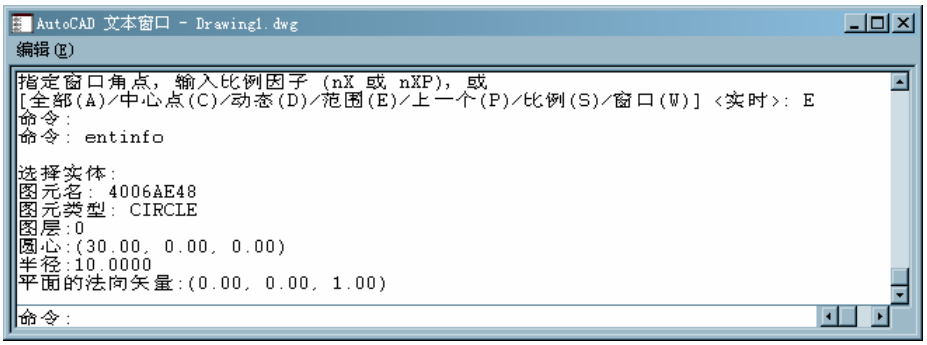

图5.5　显示实体的参数信息

5.1.5 小结

虽然 acedCommand 和 acedCmd 函数能够调用 AutoCAD 内部命令完成一些操作，但是这两个函数会降低程序的灵活性，因此在 ObjectARX 编程中不推荐使用。

前面的章节中已经使用过 acedEntSel 等函数（以 aced 开头的函数大多都是 ADSRX 函数），并且使用过它的返回值 RTNORM 来判断函数是否正确执行。实际上，大多数的 ADSRX 函数的返回值都是下面的一种：

- RTNORM：库函数成功执行。
- RTERROR：库函数执行过程中遇到一个可以捕获的错误。
- RTCAN：用户按下 Esc 键终止函数的执行。
- RTREJ：AutoCAD 拒绝了无效的请求。
- RTFAIL：与 AutoLISP 的连接失败。
- RTKWORD：用户输入了一个关键字。

5.2 和用户交互

5.2.1 说明

ObjectARX 中提供了多个提示用户输入的全局函数，包括 acedGetString、acedGetPoint、acedGetInt、acedGetKword 和 acedGetReal 等。这些方法本身的使用很简单，仅在小结部分提供参考的使用代码，本节所要讨论的主要问题是 acedGetPoint 函数和关键字的结合应用，例如在 AutoCAD 中执行 PLINE 命令时，能够得到如下的命令提示：

命令: _pline
指定起点:
当前线宽为 0.0000
指定下一个点或 [圆弧(A)/半宽(H)/长度(L)/放弃(U)/宽度(W)]:
指定下一点或 [圆弧(A)/闭合(C)/半宽(H)/长度(L)/放弃(U)/宽度(W)]:

其中的提示“指定下一个点或 [圆弧(A)/半宽(H)/长度(L)/放弃(U)/宽度(W)]:”就将提示用户输入点和关键字结合在一起。

本节的实例将要模拟 AutoCAD 中动态创建多段线的效果，并且允许用户通过命令行的关键字来改变多段线的某些特性。

5.2.2 思路

1. 动态创建多段线

动态创建多段线，最基本的要求是用户在图形窗口中按顺序拾取多个顶点，每次拾取一点都会将其添加到多段线的末尾，最终按下 Enter 键或者 Esc 键完成多段线的创建。

根据这个思路，可以编写下面的代码来完成动态创建多段线：

```
void ZffCHAP5AddPolyBasic()
{
        int index = 2;                  // 当前输入点的次数
        ads_point ptStart;              // 起点
        if (acedGetPoint(NULL, "\n输入第一点：", ptStart) != RTNORM)
            return;

        ads_point ptPrevious, ptCurrent; // 前一个参考点，当前拾取的点
        acdbPointSet(ptStart, ptPrevious);
        AcDbObjectId polyId;                    // 多段线的ID
        while (acedGetPoint(ptPrevious, "\n输入下一点：", ptCurrent) ==
RTNORM)
        {
```

```
                    if (index == 2)
                    {
                        // 创建多段线
                        AcDbPolyline *pPoly = new AcDbPolyline(2);
                        AcGePoint2d ptGe1, ptGe2;      // 两个节点
                        ptGe1[X] = ptPrevious[X];
                        ptGe1[Y] = ptPrevious[Y];
                        ptGe2[X] = ptCurrent[X];
                        ptGe2[Y] = ptCurrent[Y];
                        pPoly->addVertexAt(0, ptGe1);
                        pPoly->addVertexAt(1, ptGe2);

                        // 添加到模型空间
                        polyId = PostToModelSpace(pPoly);
                    }
                    else if (index > 2)
                    {
                        // 修改多段线，添加最后一个顶点
                        AcDbPolyline *pPoly;
                        acdbOpenObject(pPoly, polyId, AcDb::kForWrite);

                        AcGePoint2d ptGe;                // 增加的节点
                        ptGe[X] = ptCurrent[X];
                        ptGe[Y] = ptCurrent[Y];

                        pPoly->addVertexAt(index - 1, ptGe);

                        pPoly->close();
                    }

                    index++;

                    acdbPointSet(ptCurrent, ptPrevious);
                }
            }
```

上面的代码包含了最基本的动态创建多段线的语句，其算法可以描述为：

（1）拾取第一点；

（2）拾取第二点，创建多段线；

（3）如果用户没有按下 Enter 键或者 Esc 键，拾取下一点，并将拾取的点添加到多段线

的末尾；

（4）如果用户按下 Enter 键或者 Esc 键，退出程序执行，完成多段线的创建，否则反复执行步骤（3）。

2. 在acedGetPoint函数中使用关键字

要在 acedGetPoint 函数中使用关键字，必须在使用 acedGetPoint 函数之前定义关键字，并且使用 acedInitGet 函数来设置关键字，并紧跟这一句调用 acedGetPoint 函数。下面的代码演示了在 acedGetPoint 函数中使用关键字的方法：

```
void ZffCHAP5GetPoint()
{
        int rc;       // 返回值
        char kword[20];    // 关键字
        ads_point pt;
        acedInitGet(RSG_NONULL, "Keyword1 keyWord2");
        rc = acedGetPoint(NULL, "输入一个点或[Keyword1/keyWord2]:", pt);

        switch (rc)
        {
        case RTKWORD: // 输入了关键字
            if (acedGetInput(kword) != RTNORM)
                return;
            if (strcmp(kword, "Keyword1") == 0)
                acedAlert("选择的关键字是Keyword1!");
            else if (strcmp(kword, "keyWord2") == 0)
                acedAlert("选择的关键字是keyWord2!");
            break;
        case RTNORM:
            acutPrintf("输入点的坐标是(%.2f, %.2f, %.2f)",
                pt[X], pt[Y], pt[Z]);
        }       // switch
}
```

要获得 acedGetPoint 函数执行过程中用户输入的关键字，可以通过该函数的返回值来判断用户是否输入了关键字，如果返回值是 RTKWORD 就通过 acedGetInput 函数获得输入关键字的内容。

之所以两个关键字的大写字母不都是第一个，分别为 Keyword1 和 keyWord2，是为了让用户可以用不同的大写字母（K 和 W）来代表这两个关键字。

5.2.3 步骤

（1）启动 VC++ 6.0，使用 ObjectARX 向导创建一个新工程，其名称为 AddPolyDynamic。

注册一个新命令 AddPoly，该命令的实现函数被放置在 AddPolyDynamicCommands.cpp 文件中，在此文件的开头创建一个函数 GetWidth，用于获取用户输入的线宽，其实现代码为：

```
ads_real GetWidth()
{
        ads_real width = 0;
        if (acedGetReal("\n输入线宽:", &width) == RTNORM)
        {
            return width;
        }
        else
        {
            return 0;
        }
}
```

本函数没有对线宽的有效性进行检验。

（2）在 GetWidth 函数后面添加一个函数 GetColorIndex，用于提示用户输入颜色索引值，其实现代码为：

```
int GetColorIndex()
{
        int colorIndex = 0;
        if (acedGetInt("\n输入颜色索引值(0～256):", &colorIndex) !=
RTNORM)
            return 0;

        // 处理颜色索引值无效的情况
        while (colorIndex < 0 || colorIndex > 256)
        {
            acedPrompt("\n输入了无效的颜色索引.");
            if (acedGetInt("\n输入颜色索引值(0～256):", &colorIndex) !=
RTNORM)
                return 0;
        }

        return colorIndex;
}
```

这个函数的结构与 GetWidth 基本相同，不过我们为这个函数加上更完善的错误处理机制，因为颜色索引值的有效范围是 0～256。

（3）注册一个新命令 AddPoly，提示用户输入多段线的节点、线宽和颜色，完成多段线的创建，其实现代码为：

```
void ZffCHAP5AddPoly()
{
        int colorIndex = 0;             // 颜色索引值
        ads_real width = 0;                 // 多段线的线宽

        int index = 2;                  // 当前输入点的次数
        ads_point ptStart;              // 起点

        // 提示用户输入起点
        if (acedGetPoint(NULL, "\n输入第一点:", ptStart) != RTNORM)
            return;

        ads_point ptPrevious, ptCurrent; // 前一个参考点，当前拾取的点
        acdbPointSet(ptStart, ptPrevious);
        AcDbObjectId polyId;                    // 多段线的ID

        // 输入第二点
        acedInitGet(NULL, "W C O");
        int rc = acedGetPoint(ptPrevious,
            "\n输入下一点  [宽度(W)/颜色(C)]<完成(O)>:", ptCurrent);
        while (rc == RTNORM || rc == RTKWORD)
        {
            if (rc == RTKWORD)        // 如果用户输入了关键字
            {
                char kword[20];
                if (acedGetInput(kword) != RTNORM)
                    return;
                if (strcmp(kword, "W") == 0)
                {
                    width = GetWidth();
                }
                else if (strcmp(kword, "C") == 0)
                {
                    colorIndex = GetColorIndex();
                }
                else if (strcmp(kword, "O") == 0)
                {
                    return;
                }
```

```
        else
        {
            acutPrintf("\n无效的关键字.");
        }
    }
    else if (rc == RTNORM)    // 用户输入了点
    {
        if (index == 2)
        {
            // 创建多段线
            AcDbPolyline *pPoly = new AcDbPolyline(2);
            AcGePoint2d ptGe1, ptGe2;   // 两个节点
            ptGe1[X] = ptPrevious[X];
            ptGe1[Y] = ptPrevious[Y];
            ptGe2[X] = ptCurrent[X];
            ptGe2[Y] = ptCurrent[Y];
            pPoly->addVertexAt(0, ptGe1);
            pPoly->addVertexAt(1, ptGe2);

            // 修改多段线的颜色和线宽
            pPoly->setConstantWidth(width);
            pPoly->setColorIndex(colorIndex);

            // 添加到模型空间
            polyId = PostToModelSpace(pPoly);
        }
        else if (index > 2)
        {
            // 修改多段线，添加最后一个顶点
            AcDbPolyline *pPoly;
            acdbOpenObject(pPoly, polyId, AcDb::kForWrite);

            AcGePoint2d ptGe;              // 增加的节点
            ptGe[X] = ptCurrent[X];
            ptGe[Y] = ptCurrent[Y];
            pPoly->addVertexAt(index - 1, ptGe);

            // 修改多段线的颜色和线宽
            pPoly->setConstantWidth(width);
```

```
                    pPoly->setColorIndex(colorIndex);

                    pPoly->close();
                }

                index++;

                acdbPointSet(ptCurrent, ptPrevious);
            }

            // 提示用户输入新的节点
            acedInitGet(NULL, "W C O");
            rc = acedGetPoint(ptPrevious,
                "\n输入下一点 [宽度(W)/颜色(C)]<完成(O)>:", ptCurrent);
        }
    }
```

在前面已经分别对动态创建多段线和在 acedGetPoint 中使用关键字作了详细的介绍，这里的代码基本上就是将两个程序综合在一起，不再详细分析了。上面的代码中，使用了 acdbPointSet 宏将一个 ads_point 变量的值复制到另一个 ads_point 变量中，其定义为：

```
#define acdbPointSet(from, to) (*(to)= *(from), (to)[1]=(from)[1], (to)[2]=(from)[2])
```

5.2.4 效果

编译链接程序，启动 AutoCAD 2002，加载生成的 ARX 文件，执行 AddPoly 命令，按照命令提示进行操作：

```
命令: addpoly

输入第一点:
输入下一点 [宽度(W)/颜色(C)]<完成(O)>:
输入下一点 [宽度(W)/颜色(C)]<完成(O)>:w          【输入W要求改变线宽】

输入线宽:2                                        【输入2作为线宽值】

输入下一点 [宽度(W)/颜色(C)]<完成(O)>:c          【输入C要求改变颜色】

输入颜色索引值(0～256):1                         【创建红色的多段线】

输入下一点 [宽度(W)/颜色(C)]<完成(O)>:
输入下一点 [宽度(W)/颜色(C)]<完成(O)>:o          【输入O完成创建】
```

完成操作后，在图形窗口中会得到一条线宽为 2 的红色多段线。

5.2.5 小结

这里简单介绍和用户交互的其他函数。

1. 其他交互函数

acedGetInt 函数暂停程序执行，等待用户输入一个整数。该函数定义为：

```
int acedGetInt(const char * prompt, int * result);
```

prompt 用于指定显示在命令窗口中的字符串，如果不需要使用可以指定 NULL 作为该参数的值。

acedGetReal 函数暂停程序执行，等待用户输入一个实数。

acedGetString 函数暂停程序执行，等待用户输入一个字符串，该函数定义为：

```
int acedGetString(int cronly, const char * prompt, char * result);
```

其中，参数 cronly 说明字符串能否包含空格，输入 0 表示不允许字符串中包含空格，1 表示字符串可以包含空格。

acedGetDist 函数暂停程序执行，等待用户输入直线距离。

acedGetCorner 函数暂停程序执行，等待用户输入图形窗口的矩形框的对角点，一般和 acedGetPoint 函数配合使用。

acedGetAngle 函数暂停程序，等待用户输入一个相对于 ANGBASE 系统变量设置的当前值的角度。

acedInitGet、acedGetKword 和 acedGetInput 配合用于提示用户输入关键字，并根据用户输入的关键字执行不同的操作。

2. 使用acedGetFileD函数

acedGetFileD 函数使用标准的 AutoCAD 文件对话框提示用户输入一个文件名，其定义为：

```
int acedGetFileD(const char * title,
                 const char * default,
                 const char * ext,
                 int flags,
                 struct resbuf * result);
```

其中，title 指定对话框的标题；default 指定默认的文件名称；ext 指定默认的文件扩展名；flags 参数用一个位值控制对话框的样式；result 参数包含了用户选择的文件名和路径。下面的代码演示了

```
void ZffCHAP5SelectFile()
{
        const char* title = "选择图形文件";
        const char* path = "C:\\";
        struct resbuf *fileName;
```

```
        fileName = acutNewRb(RTSTR);

        if (acedGetFileD(title, path, "dwg;dxf", 0, fileName) == RTNORM)
        {
            acedAlert(fileName->resval.rstring);
        }

        acutRelRb(fileName);
    }
```

执行该命令之后，能够得到如图 5.6所示的结果。

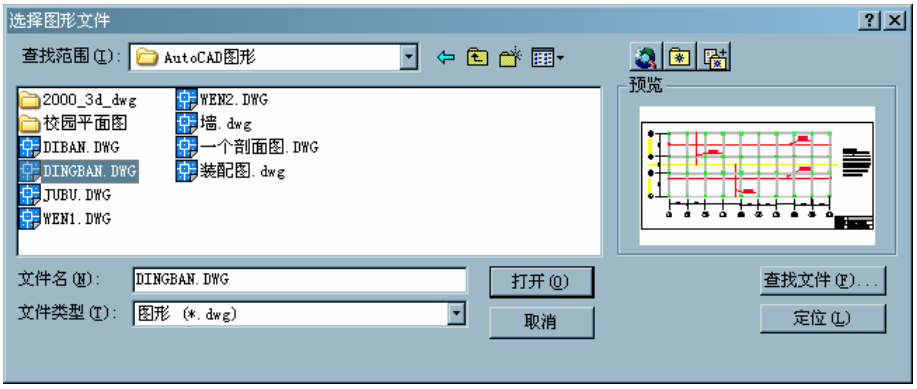

图5.6　使用标准的 AutoCAD 文件对话框

5.3 选择集

5.3.1 说明

在 ObjectARX 开发过程中，经常需要用户和 AutoCAD 之间进行交互操作，除了前面介绍的acedGetXX 系列函数之外，选择集是 AutoCAD 和用户交互操作的重要手段。与 acedEntSel 函数不同，acedEntSel 函数用于提示用户选择一个实体，选择集允许用户同时选择多个图形对象，同时提供了丰富的手段来选择符合特定条件的实体。

本节的实例详细介绍了选择集的使用，可以分为以下几个方面：

- 选择集的创建和删除。
- 选择集中对象的增加和删除。
- 对象选择的方法。
- 选择集过滤器的使用。

5.3.2 思路

1. 选择集的创建和删除

选择集是 AutoCAD 当前图形中的一组实体，通过图元名进行引用，也就是一个 ads_name 变量。创建选择集可以使用 acedSSAdd 和 acedSSGet 函数，其中 acedSSGet 函数提供了绝大多数创建选择集的方法：

- 提示用户选择实体。
- 使用 PICKFIRST 选择集（在未执行命令时用户已经选择的图形集合，也就是 AutoCAD 中的先选择、再输入命令）或者交叉（Crossing）、多变性交叉（Crossing Polygon）、栅栏（Fence）、最后一个（Last）、前一个（Previous）、窗口（Window）、多边形窗口（Window Polygon）等方式，也可以指定一个点或者一系列点来明确地限定所要选择的实体。
- 指定选择实体所要满足的一系列属性和条件来过滤当前数据库，可以和前面的选择方式配合使用。

无论使用 acedSSAdd 还是 acedSSGet 函数，都需要在程序结束之前使用 acedSSFree 函数释放选择集的内存空间。

2. 选择集中元素的增加和删除

这里所说的元素增加和删除，仅指在已经获得对象引用的情况下，使用 acedSSAdd 和 acedSSDel 函数对选择集进行元素的增加和删除。

acedSSAdd 函数定义为：

```
int acedSSAdd(const ads_name ename, const ads_name sname, ads_name result);
```

其中，ename 指定要添加到选择集的实体的图元名；sname 指定选择集的图元名；result 返回被创建或者更新的选择集。根据 ename 和 sname 的不同取值，acedSSAdd 有以下几种可能的执行结果：

- 如果 ename 和 sname 都是空指针，则创建一个未包含任何成员的选择集。
- 如果 ename 指向一个有效的实体，但 sname 是空指针，则创建一个选择集，该选择集仅包含一个成员 ename。
- 如果 ename 指向有效的实体，且 sname 指向有效的选择集，则将 ename 所指向的实体添加到 sname 所指向的选择集中。

acedSSDel 函数定义为：

```
int acedSSDel(const ads_name ename, const ads_name ss);
```

其中，ename 指定了要从选择集中删除的实体；ss 指定了所要操作的选择集。

3. 对象选择的方法

所谓对象选择的方法，就是以某种方式从图形窗口中获得满足某些条件的图形对象。这里要介绍的是使用 acedSSGet 函数所实现的选择对象的方法，该函数被定义为：

```
int acedSSGet (const char *str,
```

```
const void *pt1,
const void *pt2,
const struct resbuf *entmask,
ads_name ss);
```

其中，str 参数描述了创建选择集的方法，可以使用的参数值参见表 5-1；pt1 和 pt2 为相关的创建方式提供了点参数，如果不需要指定可以输入 NULL 作为参数值；entmask 用于指定选择实体的过滤条件；ss 则指定了要操作的选择集的图元名。

表 5-1 acedSSGet 函数的选择模式选项

值（选择模式）	说 明
NULL	单点选择（如果指定了 pt1）或者提示用户选择（如果 pt1 的值为 NULL）
#	非几何选择模式（包括 All、Last 和 Previous 选择模式）
:$	仅提供提示（Prompts supplied）
.	用户选择模式
:?	其他回调选择模式（Other callbacks）
A	全部选择
B	框选模式
C	窗交选择模式
CP	圈交选择模式（选择多边形（通过在待选对象周围指定点来定义）内部或与之相交的所有对象）
:D	允许复制选择模式（Duplicates OK）
:E	小孔中的所有实体（Everything in aperture）
F	栏选模式
G	选择编组
I	获得当前图形窗口中已经选择的实体（PickFirst 选择集）
:K	键盘回调选择模式（Keyword callbacks）
L	选择最近一次创建的可见实体
M	指定多次选择而不高亮显示对象，从而加快对复杂对象的选择过程
P	选择最近创建的选择集
:S	单一对象选择模式
W	窗口选择模式
WP	圈围选择模式
X	过滤选择模式

4. 使用选择集过滤器

在使用各种选择对象的方法时，可以使用过滤器来限定选择的对象。例如，可以指定仅

选择图层 0 上的直线对象，也可以指定仅选择蓝色的半径大于 30 的圆，等等。

如果仅使用一个过滤条件，可以使用下面的代码：

```
struct resbuf rb;
char sbuf[10];       // 存储字符串的缓冲区
ads_name ssname;

rb.restype = 0;      // 实体名
strcpy(sbuf, "CIRCLE");
rb.resval.rstring = sbuf;
rb.rbnext = NULL; // 不需要设置其他的属性

// 选择图形中所有的圆
acedSSGet("X", NULL, NULL, &rb, ssname);

acedSSFree(ssname);
```

上面的代码中虽然使用了结果缓冲区，但是仅是在栈上声明，由编译器自动管理它所使用的内存空间，不需要使用 acutRelRb 函数来手工销毁它。

如果要指定多个过滤条件，可以使用 acutBuildList 函数来构造结果缓冲区。如果要选择当前图形窗口中位于 0 层上的所有直线，就可以使用下面的方法：

```
struct resbuf *rb;// 结果缓冲区链表
ads_name ssname;

rb = acutBuildList(RTDXF0, "LINE",          // 实体类型
        8, "0",                             // 图层
        RTNONE);

// 选择图形中位于0层上的所有直线
acedSSGet("X", NULL, NULL, rb, ssname);

acutRelRb(rb);
acedSSFree(ssname);
```

在标准的 DXF 组码中，0 用于表示实体类型，但是在 acutBuildList 函数中 0 也可以用于表示结束链表，因此用 RTDXF0 来代替 0。

过滤器列表由成对的参数组成。第一个参数标识过滤器的类型（例如对象），第二个参数指定要过滤的值（例如圆）。过滤器类型是指定使用哪种过滤器的 DXF 组码，下面列出了一些最常用的过滤器类型：

- 0：对象类型（字符串），例如“Line”、“Arc”等。
- 1：图元的主文字值（字符串），例如文字、属性的字符串内容。
- 2：对象名（字符串），例如属性标记、块名等。

- 8：图层名（字符串）。
- 10：主要点（直线或文字图元的起点、圆的圆心等），用三个实数的数组来表示三维点。
- 60：对象可见性（整数），0 表示可见，1 表示不可见。
- 62：颜色编号（整数），可取 0～256 的整数值，其中 0 表示 ByBlock，256 表示 ByLayer。
- 67：模型 / 图纸空间标识符（整数），0（默认）表示在模型空间，1 表示在图纸空间。
- 1000：扩展数据中的 ASCII 字符串，最多可以包含 255 个字节。
- 1001：扩展数据的注册应用程序名，最多可以包含 31 个字节的 ASCII 字符串。
- 1003：扩展数据图层名。

关于DXF组码的完整列表，可以参考“DXF参考”中的内容（AutoCAD 2002 中文版帮助系统中的DXF参考是英文的，因此这里使用的是AutoCAD 2005 中文版的帮助系统），如图5.7所示。

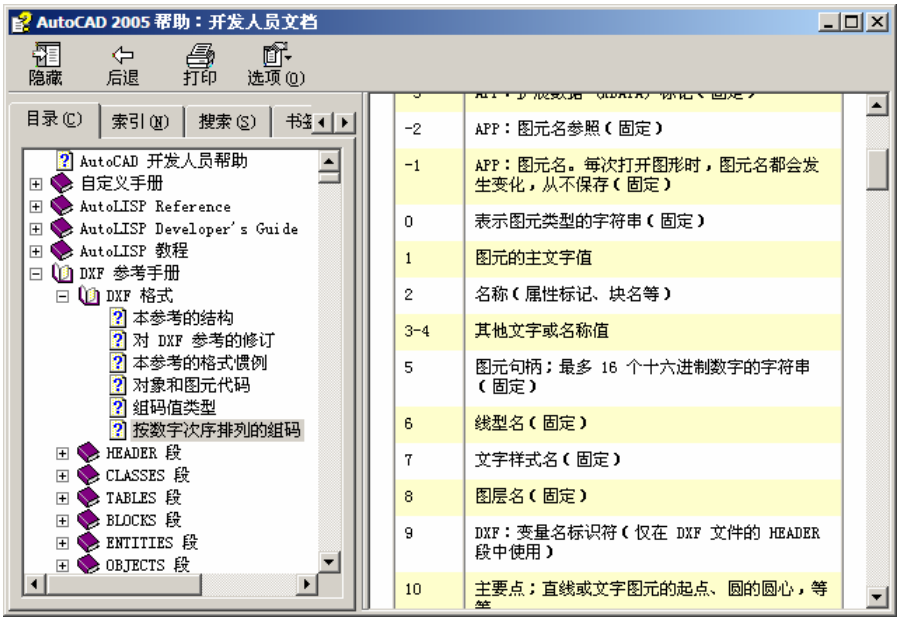

图5.7 完整的 DXF 组码参考

值得一提的是，在大部分选择模式中均可以使用选择集过滤器，下面的代码提示用户选择实体的同时使用了选择集过滤器：

```
acedSSGet(NULL, NULL, NULL, rb, ssname);
```

5.3.3 步骤

1. 选择集的创建和删除

启动 VC++ 6.0，使用 ObjectARX 向导创建一个新工程，其名称为 SelectionSet。注册一个新命令 CreateSSet，用于演示选择集的创建和删除，其实现函数为：

```
void ZffCHAP5CreateSSet()
{
        ads_name sset; // 选择集名称
        // 选择图形数据库中所有的实体
        acedSSGet("A", NULL, NULL, NULL, sset);

        // 进行其他操作

        acedSSFree(sset);
}
```

2. 对象选择的方法

（1）注册一个新命令 SelectEnt，使用多种不同的模式创建选择集，其实现函数为：

```
void ZffCHAP5SelectEnt()
{
        ads_point pt1, pt2, pt3, pt4;
        struct resbuf *pointlist;       // 结果缓冲区链表
        ads_name ssname;                // 选择集的图元名
        pt1[X] = pt1[Y] = pt1[Z] = 0.0;
        pt2[X] = pt2[Y] = 5.0; pt2[Z] = 0.0;

        // 如果已经选择到了实体，就获得当前的PICKFIRST选择集
        // 否则提示用户选择实体
        acedSSGet(NULL, NULL, NULL, NULL, ssname);

        // 如果存在，就获得当前的PickFirst选择集
        acedSSGet("I", NULL, NULL, NULL, ssname);

        // 选择最近创建的选择集
        acedSSGet("P", NULL, NULL, NULL, ssname);

        // 选择最后一次创建的可见实体
        acedSSGet("L", NULL, NULL, NULL, ssname);

        // 选择通过点(5,5)的所有实体
        acedSSGet(NULL, pt2, NULL, NULL, ssname);

        // 选择位于角点(0,0)和(5,5)组成的窗口内所有的实体
        acedSSGet("W", pt1, pt2, NULL, ssname);
```

```
    // 选择被指定的多边形包围的所有实体
    pt3[X] = 10.0; pt3[Y] = 5.0; pt3[Z] = 0.0;
    pt4[X] = 5.0; pt4[Y] = pt4[Z] = 0.0;
    pointlist = acutBuildList(RTPOINT, pt1, RTPOINT, pt2,
            RTPOINT, pt3, RTPOINT, pt4, 0);
    acedSSGet("WP", pointlist, NULL, NULL, ssname);

    // 选择与角点(0,0)和(5,5)组成的区域相交的所有实体
    acedSSGet("C", pt1, pt2, NULL, ssname);

    // 选择与指定多边形区域相交的所有实体
    acedSSGet("CP", pointlist, NULL, NULL, ssname);
    acutRelRb(pointlist);

    // 选择与选择栏相交的所有对象
    pt4[Y] = 15.0; pt4[Z] = 0.0;
    pointlist = acutBuildList(RTPOINT, pt1, RTPOINT, pt2,
            RTPOINT, pt3, RTPOINT, pt4, 0);
    acedSSGet("F", pointlist, NULL, NULL, ssname);

    acutRelRb(pointlist);
    acedSSFree(ssname);
}
```

上面的代码取自帮助系统，演示了 acedSSGet 函数的几个典型应用。使用 CP 和 WP 选择模式时，需要用 acutBuildList 函数创建一个包含多边形顶点的结果缓冲区链表，acedSSGet 函数会自动闭合顶点列表，因此不必创建一个首尾顶点相同的结果缓冲区链表。

（2）选择【Insert/New Class】菜单项，系统会弹出如图 5.8所示的【New Class】对话框，在【Name】文本框中输入CPubFunction作为类的名称，单击【OK】按钮完成类的创建。

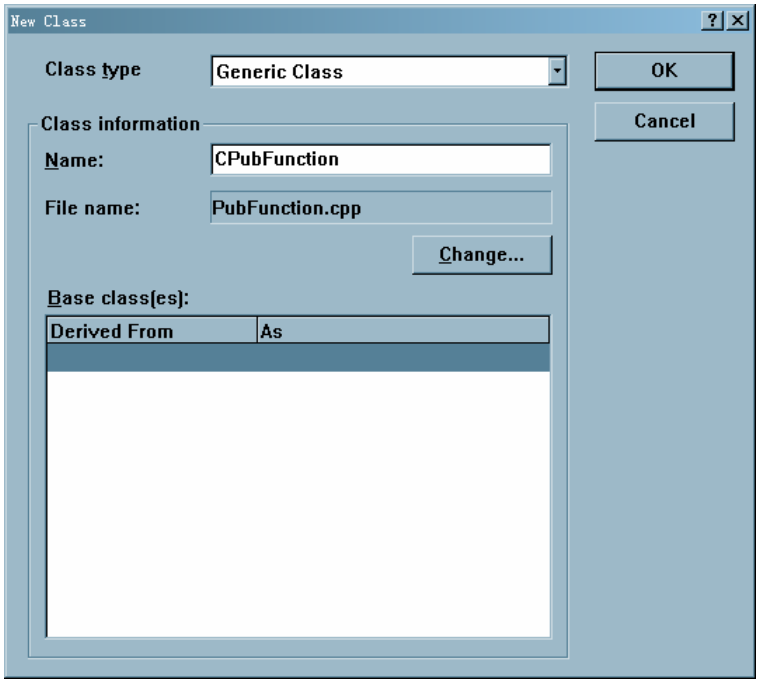

图5.8　添加一个新的类

（3）在ClassView视图中右键单击CPubFunction，从弹出的快捷菜单中选择【Add Member Function】菜单项，系统会弹出如图 5.9所示的对话框。分别输入函数的返回值类型和声明，单击【OK】按钮完成成员函数PolyToGeCurve的添加。

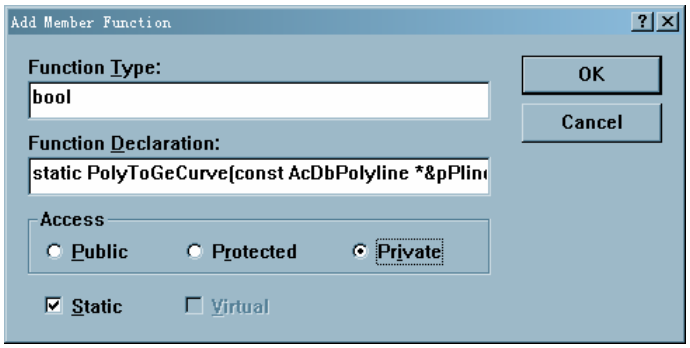

图5.9　向类中添加成员函数

将函数声明为类的私有静态函数，意味着不能从类的外部访问该函数，并且在访问该函数的时候只能通过类名来引用，例如 CPubFunction::PolyToGeCurve。

当然，如果你愿意，也可以在类的头文件和实现文件中分别输入函数的声明和实现部分，与使用对话框添加成员函数的效果是一致的。

（4）添加 PolyToGeCurve 函数的实现代码。该函数用于根据指定的多段线创建对应的几何类曲线，包含两个参数，pPline 指定已知的多段线，pGeCurve 参数输出创建的几何曲线。其实现函数为：

```
bool CPubFunction::PolyToGeCurve(const AcDbPolyline *&pPline, AcGeCurve2d
*&pGeCurve)
{
        int nSegs;                              // 多段线的段数
        AcGeLineSeg2d line, *pLine;             // 几何曲线的直线段部分
        AcGeCircArc2d arc, *pArc;               // 几何曲线的圆弧部分
        AcGeVoidPointerArray geCurves;          // 指向组成几何曲线各分段
的指针数组

        nSegs = pPline->numVerts() - 1;

        // 根据多段线创建对应的分段几何曲线
        for (int i = 0; i < nSegs; i++)
        {
            if (pPline->segType(i) == AcDbPolyline::kLine)
            {
                pPline->getLineSegAt(i, line);
                pLine = new AcGeLineSeg2d(line);
                geCurves.append(pLine);
            }
            else if (pPline->segType(i) == AcDbPolyline::kArc)
            {
                pPline->getArcSegAt(i, arc);
                pArc = new AcGeCircArc2d(arc);
                geCurves.append(pArc);
            }
        }

        // 处理闭合多段线最后一段是圆弧的情况
        if (pPline->isClosed() && pPline->segType(nSegs) ==
AcDbPolyline::kArc)
        {
            pPline->getArcSegAt(nSegs, arc);
            pArc = new AcGeCircArc2d(arc);
            pArc->setAngles(arc.startAng(), arc.endAng() -
                (arc.endAng() - arc.startAng()) / 100);
            geCurves.append(pArc);
        }
```

```
        // 根据分段的几何曲线创建对应的复合曲线
        if (geCurves.length() == 1)
        {
            pGeCurve = (AcGeCurve2d *)geCurves[0];
        }
        else
        {
            pGeCurve = new AcGeCompositeCurve2d(geCurves);
        }


        // 释放动态分配的内存
        if (geCurves.length() > 1)
        {
            for (i = 0; i < geCurves.length(); i++)
            {
                delete geCurves[i];
            }
        }

        return true;
    }
```

函数参数前面的*&表示该参数是一个指针的引用，其含义在 2.1 节已经介绍过，使用指针的引用作为参数，能够在函数内部对指针的本身的内容进行修改。

首先使用 segType 成员函数来判断多段线某一段的类型，如果是直线段就创建一个 AcGeLineSeg2d 对象，如果是圆弧段则创建一个 AcGeCircArc2d 对象，并将创建的几何类对象添加到一个对象指针数组中。

如果多段线闭合，并且最后的一段是圆弧段，那么还要将终点和起点之间的那一段曲线添加到最终的几何曲线中，这样才能保证多段线与对应的几何类曲线形状完全一致。如果最后一段是直线段，则不需要添加，其理由将在下一个函数的分析中说明。注意其中下面的这句代码：

```
pArc->setAngles(arc.startAng(), arc.endAng() -  (arc.endAng() - arc.startAng()) / 100);
```

这句代码将会使最后一段曲线的终点发生一个小的偏移（偏移后仍在原来的圆弧上，但不与原来的终点重合），其原因同样会在下一个函数的分析中说明。

再来看下面的代码：

```
        if (geCurves.length() == 1)
        {
            pGeCurve = (AcGeCurve2d *)geCurves[0];
        }
```

```
        else
        {
            pGeCurve = new AcGeCompositeCurve2d(geCurves);
        }
```

如果多段线仅包含一段曲线，直接将几何曲线的指针指向新建的几何曲线对象；否则根据对象指针数组中的元素创建一个新的复合曲线，并且为其分配相应的内存。对应地，当多段线的段数大于 1 时，函数结束前应该用 delete 操作符释放用 new 运算符分配所有的几何曲线的存储空间。

（3）在 CPubFunction 类中添加一个新的成员函数 SelectEntInPoly，用于选择指定多段线内部（或者与多段线构成的区域相交）的所有实体。该函数包含 4 个参数，pPline 指定已知的多段线，ObjectIdArray 输出选择到的所有实体的 ObjectId 数组，selectMode 指定选择模式（可以输入“CP”或者“WP”），approxEps 指定构造多段线对应几何曲线时的误差。该函数的实现代码为：

```
    bool CPubFunction::SelectEntInPoly(AcDbPolyline *pPline,
        AcDbObjectIdArray &ObjectIdArray, const char *selectMode, double
approxEps)
    {
        // 判断selectMode的有效性
        if (strcmp(selectMode, "CP") != 0 && strcmp(selectMode, "WP") != 0)
        {
            acedAlert("函数SelectEntInPline中，指定了无效的选择模式！");
            return false;
        }

        // 清除数组中所有的ObjectId
        for (int i = 0; i < ObjectIdArray.length(); i++)
        {
            ObjectIdArray.removeAt(i);
        }

        AcGeCurve2d *pGeCurve;                          // 多段线对应的几
何曲线
        Adesk::Boolean bClosed = pPline->isClosed(); // 多段线是否闭合
        if (bClosed != Adesk::kTrue)                    // 确保多段线作为选择边界
时是闭合的
        {
            pPline->setClosed(!bClosed);
        }
```

```
// 创建对应的几何类曲线
CPubFunction::PolyToGeCurve(pPline, pGeCurve);

// 获得几何曲线的样本点
AcGePoint2dArray SamplePtArray;                    // 存储曲线的样本
点
AcGeDoubleArray ParamArray;                        // 存储样本点对应
的参数值
AcGePoint2d ptStart, ptEnd;                  // 几何曲线的起点和终点
Adesk::Boolean bRet = pGeCurve->hasStartPoint(ptStart);
bRet = pGeCurve->hasEndPoint(ptEnd);
double valueSt = pGeCurve->paramOf(ptStart);
double valueEn = pGeCurve->paramOf(ptEnd);
pGeCurve->getSamplePoints(valueSt, valueEn, approxEps,
        SamplePtArray, ParamArray);

delete pGeCurve;        // 在函数PolyToGeCurve中分配了内存

// 确保样本点的起点和终点不重合
AcGeTol tol;
tol.setEqualPoint(0.01);
AcGePoint2d ptFirst = SamplePtArray[0];
AcGePoint2d ptLast = SamplePtArray[SamplePtArray.length() - 1];
if (ptFirst.isEqualTo(ptLast))
{
        SamplePtArray.removeLast();
}

// 根据样本点创建结果缓冲区链表
struct resbuf *rb;
rb = CPubFunction::BuildRbFromPtArray(SamplePtArray);

// 使用acedSSGet函数创建选择集
ads_name ssName;                  // 选择集名称
int rt = acedSSGet(selectMode, rb, NULL, NULL, ssName);
if (rt != RTNORM)
{
        acutRelRb(rb);            // 释放结果缓冲区链表
        return false;
```

```
        }

        // 将选择集中所有的对象添加到ObjectIdArray
        long length;
        acedSSLength(ssName, &length);
        for (i = 0; i < length; i++)
        {
            // 获得指定元素的ObjectId
            ads_name ent;
            acedSSName(ssName, i, ent);
            AcDbObjectId objId;
            acdbGetObjectId(objId, ent);

            // 获得指向当前元素的指针
            AcDbEntity *pEnt;
            Acad::ErrorStatus es = acdbOpenAcDbEntity(pEnt, objId,
AcDb::kForRead);

            // 选择到作为边界的多段线了，直接跳过该次循环
            if (es == Acad::eWasOpenForWrite)
            {
                continue;
            }

            ObjectIdArray.append(pEnt->objectId());

            pEnt->close();
        }

        // 释放内存
        acutRelRb(rb);                          // 释放结果缓冲区链表
        acedSSFree(ssName);                     // 删除选择集

        return true;
    }
```

首先判断 selectMode 参数的有效性，清空 ObjectIdArray 的所有元素（引用类型传递的参数，防止该参数在函数外的修改影响该函数执行的结果），然后使用 PolyToGeCurve 函数创建与指定多段线对应的几何类曲线。

此后，使用 AcGeCurve2d 类的 getSamplePoints 函数来获得多段线的样本点。为什么不直

接用数据库实体类AcDbPolyline来获得多段线的顶点来作为样本点呢？样本点是用于描述曲线形状的一组点，获得多段线的形状，可以使用两种方法：

- 等距获得样本点。输入一个固定的长度值，从起点开始，每到这个固定的长度就在曲线上取一个样本点，最后得到的样本点是等间距的。
- 按照弦切距的误差极限来获得样本点，这种方法可以控制样本点的精度。要根据一定的精度来获得几何类对象的样本点，必须深入理解几何类的getSamplePoints函数，这个函数的一个定义是：

```
void getSamplePoints(
                double fromParam,
                double toParam,
                double approxEps,
                AcGePoint3dArray& pointArray,
                AcGeDoubleArray& paramArray) const;
```

其中，fromParam是开始处的参数，toParam是结束处的参数，approxEps是弦高误差，pointArray输出曲线上位于开始参数fromParam和终止参数toParam之间的所有点的数组，paramArray输出与数组pointArray中的点相对应的参数数组[2]。这里所说的“参数”，是一个与长度有关的值，其具体含义Autodesk并未公开[3]。弦高误差的含义如图 5.10所示。

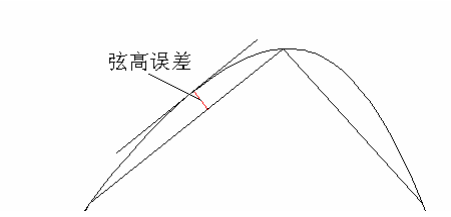

图5.10　弦高误差的含义

因此，对一条多段线取样本点，能够得到类似图 5.11所示的结果，多段线的顶点A、B、C、D仍然会是样本点。从图中可以看出，样本点很好地描述了多段线的形状，而仅由顶点组成的多边形根本无法描述带有圆弧段的多段线形状。

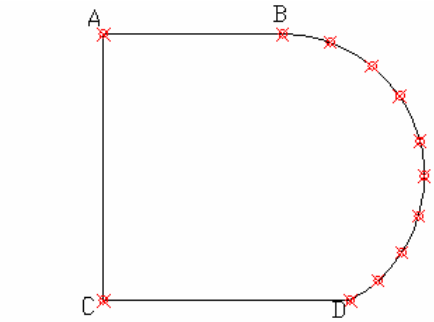

图5.11　样本点和多段线顶点

接下来的代码根据样本点创建结果缓冲区链表，在此之前，需要确保样本点数组中首尾

两点不重合，其中的原因在前面已经介绍，acedSSGet 函数会自动闭合顶点列表。创建结果缓冲区链表使用了一个自定义函数 BuildRbFromPtArray，该函数的定义将在下面的步骤中介绍。

创建选择集之后，遍历选择集，将选择到的实体添加到 ObjectId 数组中，但是有时候作为边界的多段线也被添加到选择集中，在添加 ObjectId 的时候就要将其排除。此时多段线还没有被关闭，因此使用 acdbOpenAcDbEntity 函数打开时会返回 Acad::eWasOpenForWrite，可以此作为辨别边界多段线的依据。

在函数的最后，不能忘记释放结果缓冲区链表和选择集。

（4）创建一个新的成员函数 BuildRbFromPtArray，用于根据指定的一组点创建一个结果缓冲区链表，它接受一个参数 arrPoints，该参数指定一组点，而函数返回创建的结果缓冲区链表。该函数的实现代码为：

```
struct resbuf* CPubFunction::BuildRbFromPtArray(const AcGePoint2dArray
&arrPoints)
{
        struct resbuf *retRb = NULL;
        int count = arrPoints.length();
        if (count <= 1)
        {
                acedAlert("函数BuildBbFromPtArray中，点数组包含元素个数不
足！");
                return retRb;
        }

        // 使用第一个点来构建结果缓冲区链表的头节点
        ads_point adsPt;
        adsPt[X] = arrPoints[0].x;
        adsPt[Y] = arrPoints[0].y;
        retRb = acutBuildList(RTPOINT, adsPt, RTNONE);

        struct resbuf *nextRb = retRb;          // 辅助指针

        for (int i = 1; i < count; i++)         // 注意：不考虑第一个元素，因
此i从1开始
        {
                adsPt[X] = arrPoints[i].x;
                adsPt[Y] = arrPoints[i].y;
                // 动态创建新的节点，并将其链接到原来的链表尾部
                nextRb->rbnext = acutBuildList(RTPOINT, adsPt, RTNONE);
```

```
            nextRb = nextRb->rbnext;
        }

        return retRb;
    }
```

（5）注册一个新命令 SelectEntInPoly，提示用户选择一条多段线，在命令窗口中显示与多段线形成的区域相交的实体的个数，其实现函数为：

```
    void ZffCHAP5SelectEntInPoly()
    {
        // 提示用户选择多段线
        ads_name entName;
        ads_point pt;
        if (acedEntSel("\n选择多段线:", entName, pt) != RTNORM)
            return;

        AcDbObjectId entId;
        acdbGetObjectId(entId, entName);

        // 判断选择的实体是否是多段线
        AcDbEntity *pEnt;
        acdbOpenObject(pEnt, entId, AcDb::kForWrite);
        if (pEnt->isKindOf(AcDbPolyline::desc()))
        {
            AcDbPolyline *pPoly = AcDbPolyline::cast(pEnt);

            AcDbObjectIdArray ObjectIdArray;  // 选择到的实体ID集合
            CPubFunction::SelectEntInPoly(pPoly, ObjectIdArray, "CP", 1);
            acutPrintf("\n选择到%d个实体.", ObjectIdArray.length());
        }

        pEnt->close();
    }
```

3. 使用选择集过滤器

（1）在过滤器中使用通配符。注册一个新命令 Filter1，创建一个带有通配符的过滤器，其相关代码为：

```
    void ZffCHAP5Filter1()
    {
        struct resbuf *rb; // 结果缓冲区链表
```

```
    ads_name ssname;

    rb = acutBuildList(RTDXF0, "TEXT",          // 实体类型
        8, "0,图层1",                                  // 图层
        1, "*cadhelp*",                          // 包含的字符串
        RTNONE);

    // 选择复合要求的文字
    acedSSGet("X", NULL, NULL, rb, ssname);
    long length;
    acedSSLength(ssname, &length);
    acutPrintf("\n实体数:%d", length);

    acutRelRb(rb);
    acedSSFree(ssname);
}
```

上面的代码构建的过滤器，能够获得位于图层“0”或者“图层 1”中、字符串内容包含 cadhelp 的所有文字对象。其中，“0,图层 1”中的逗号（,）是一个通配符，用于分隔两个模式，表示两种情况均可；“*mjtd*”中的星号（*）是一个通配符，用于匹配任意的字符序列，表示任何包含 cadhelp 的字符串均可。

关于AutoCAD能够识别的通配符及其含义，可以参考图 5.12。

通配符

字符	定义
# （磅值符号）	匹配任意一个数字
@ (at)	匹配任意一个字母
. （句号）	匹配任意一个非字母数字的字符
* （星号）	匹配任意的字符序列（包括空字符串），它可以用在任何搜索模式中：包括开头、中间和结尾处
? （问号）	匹配任意一个字符
~ （波浪号）	如果它是模式中的的第一个字符，则匹配除此模式以外的任意内容
[...]	匹配方括号中的任意一个字符
[~...]	匹配不在方括号中的任意一个字符
- （连字符）	用在方括号中，指定一个字符的取值范围
, （逗号）	分隔两个模式
` （单引号）	避开特殊的字符（直接读取下一个字符）

图5.12　帮助系统中对于通配符的说明

（2）在过滤器中使用逻辑运算符。注册一个新命令 Filter2，创建包含逻辑运算符的过滤器，其相关代码为（已省略与 Filter1 命令实现函数相同的部分代码，请读者自行补全）：

```
    rb = acutBuildList(-4, "<OR",                    // 逻辑运算符开始
```

```
                RTDXF0, "TEXT",              // 一个条件
                RTDXF0, "MTEXT",             // 另一个条件
                -4, "OR>",                   // 逻辑运算符结束
                RTNONE);
```

上面构建的过滤器，能够选择图形中所有的文字和多行文字。过滤器列表中的逻辑运算符用DXF组码－4 来指示。逻辑运算符是字符串但必须成对出现，运算符以小于号开始（<），以大于号结束（>）。图 5.13列出了可以在选择集过滤中使用的逻辑运算符。

选择集过滤器列表的逻辑编组运算符

开始运算符	包含的内容	结束运算符
"<AND"	一个或多个运算对象	"AND>"
"<OR"	一个或多个运算对象	"OR>"
"<XOR"	两个运算对象	"XOR>"
"<NOT"	一个运算对象	"NOT>"

图5.13　帮助系统中对于逻辑运算符的描述

（3）在过滤器中使用关系运算符。注册一个新命令 Filter3，创建包含关系运算符的过滤器，其相关代码为（已省略与 Filter1 命令实现函数相同的部分代码，请读者自行补全）：

```
rb = acutBuildList(RTDXF0, "CIRCLE",      // 实体类型
                -4, ">=",                 // 关系运算符
                40, 30,                   // 半径
                RTNONE);
```

上面构建的过滤器，能够选择图形中半径大于或等于 30 的所有圆，其中的组码 40 用于指定圆的半径。过滤器列表中的关系运算符以DXF组码－4 来指示，用字符串来表示关系运算符。图 5.14列出了AutoCAD中可以使用的关系运算符。

选择集过滤器列表的关系运算符

运算符	说明
"*"	任何情况（总为真）
"="	等于
"!="	不等于
"/="	不等于
"<>"	不等于
"<"	小于
"<="	小于或等于
">"	大于
">="	大于或等于
"&"	按位与（AND，仅限于整数组）
"&="	按位屏蔽相等（仅限于整数组）

图5.14　帮助系统中对于关系运算符的描述

（4）结合使用通配符和关系运算符，创建更为复杂的过滤器。注册一个新命令 Filter4，创建包含关系运算符和通配符的过滤器，其相关代码为：

```
rb = acutBuildList(RTDXF0, "CIRCLE",// 实体类型
                   -4, ">,>,*",          // 关系运算符和通配符
                   10, pt1,              // 圆心
                   -4, "<,<,*",          // 关系运算符和通配符
                   10, pt2,              // 圆心
                   RTNONE);
```

上面构建的过滤器，能够选择图形中圆心在 pt1 和 pt2 两点构成的矩形内的圆，其中的组码 10 用于指定圆的圆心。

（7）使用过滤器过滤扩展数据。注册一个新命令 Filter5，创建过滤扩展数据的过滤器，其相关代码为：

```
rb = acutBuildList(1001, "XData", // 扩展数据的应用程序名
                   RTNONE);
```

上面构建的过滤器，能够选择图形中所有包含“Xdata”应用程序扩展数据的图元。令人遗憾的是，除轻量多段线之外，其他的实体都不支持扩展数据内容的过滤，也就是说，如果使用下面的代码；

```
rb = acutBuildList(1000, "Road",  // 扩展数据中的ASCII字符串
                   RTNONE);
```

上面的过滤器仅能选择到图形中所有包含字符串“Road”扩展数据的轻量多段线，其他类型的实体不起作用。

讲到这里，相信你一定感觉在创建选择集过滤器时，如何获得所需要的组码是一个令人头疼的问题。如果你很着急解决这个问题，现在就可以去看本节的小结。

5.3.4 效果

编译链接程序，启动 AutoCAD 2002，加载生成的 ARX 文件，执行其中定义的命令，根据系统提示的选择到的对象数量，可以对相应命令的执行结果进行测试。

测试一些函数时发现了一些奇怪的问题。为了测试的方便，创建如图 5.15所示的图形。其中，红色的矩形两个角点分别为（0，0）和（100，100）。

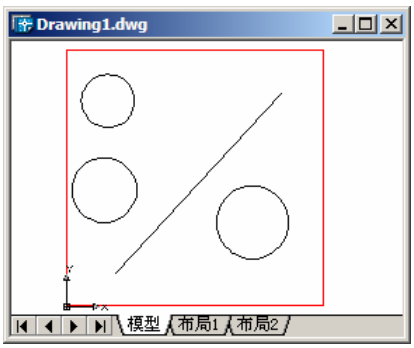

图5.15　测试所用图形

注册一个新命令 Test2，用于进行创建选择集的测试，其实现函数为：

```
void ZffCHAP5TEST2()
{
        ads_name ssname;
        ads_point pt1, pt2;
        pt1[X] = pt1[Y] = pt1[Z] = 0;
        pt2[X] = pt2[Y] = 100;
        pt2[Z] = 0;

        // 选择图形中与pt1和pt2组成的窗口相交的所有对象
        acedSSGet("C", pt1, pt2, NULL, ssname);
        long length;
        acedSSLength(ssname, &length);
        acutPrintf("\n实体数:%d", length);

        acedSSFree(ssname);
}
```

确保整个矩形都位于图形窗口内部，在AutoCAD 2002 中执行定义的命令Test2，得到的结果如图 5.16所示。这个结果容易理解，包含矩形在内，符合条件的对象是 5 个。

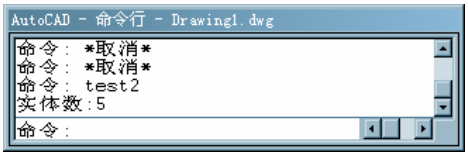

图5.16　执行结果（一）

如果平移图形，使一些实体位于图形窗口外部，如图 5.17所示。再次执行Test2 命令，能够得到如图 5.18所示的结果。

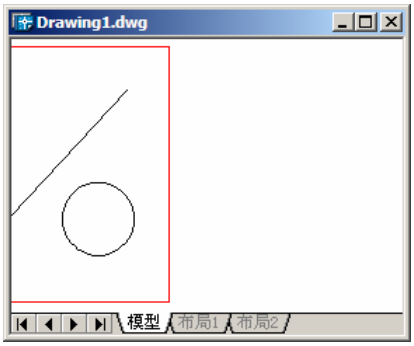

图5.17　平移图形

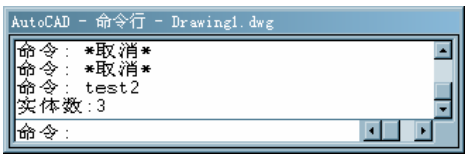

图5.18 执行结果（二）

这里的结果就让人费解了，矩形内部有 4 个对象，但是此处选择集仅获得了 3 个对象。实际上，使用 acedSSGet 函数的许多选择模式（如果使用“A”作为第一个参数，也就是使用全部选择模式不受影响）仅能获得位于图形窗口（视口）内部的实体，对于视口外的实体不起作用。从图中看，当前视口中确实包含了 3 个符合要求的实体，这个结果的含义也就明朗了。

5.3.5 小结

1. 处理创建选择集时的意外结果

由于创建选择集的一些方法仅对当前视口内的对象有效，因此可以考虑在创建选择集之前对图形进行一次全部缩放，使所有的实体显示在视口中，然后创建选择集，最后再恢复原来的视图，可以用下面的代码来描述：

```
// 保存当前视口两个角点的坐标
AcGePoint2d ptMin, ptMax;
if (!CPubFunction::GetViewPortBound(ptMin, ptMax))
{
        return;
}

// 执行全部缩放
CPubFunction::ZoomExtents();

// 执行创建选择集的操作……

// 恢复执行命令之前的视图
CPubFunction::WindowZoom(ptMin, ptMax, 1);
```

上面的代码中包含了三个自定义函数，其中，GetViewPortBound 函数用于获得当前视口两个角点的 WCS 坐标，ZoomExtents 用于对图形进行范围缩放，WindowsZoom 可以进行窗口缩放。值得一提的是，CPubFunction 类的这三个成员函数均被定义成静态（static）函数，因此调用的格式为“类名::函数名”。

GetViewPortBound 函数的实现代码为：

```
bool CPubFunction::GetViewPortBound(AcGePoint2d &ptMin, AcGePoint2d &ptMax)
{
```

```
                    // 获得当前视口的高度
                    double viewHeight;                  // 视口高度（用图形单位表示）
                    struct resbuf rbViewSize;
                    if (acedGetVar("VIEWSIZE", &rbViewSize) != RTNORM)
                    {
                        return false;
                    }
                    viewHeight = rbViewSize.resval.rreal;

                    // 获得当前视口的宽度
                    double viewWidth;               // 视口宽度（用图形单位表示）
                    struct resbuf rbScreenSize;
                    if (acedGetVar("SCREENSIZE", &rbScreenSize) != RTNORM)
                    {
                        return false;
                    }
                    // width / height = rpoint[X] / rpoint[Y]  利用高宽比计算
                    viewWidth = (rbScreenSize.resval.rpoint[X] /
rbScreenSize.resval.rpoint[X])
                        * viewHeight;

                    // 获得当前视口中心点WCS坐标
                    AcGePoint3d viewCenterPt;// 视口中心点
                    struct resbuf rbViewCtr;
                    if (acedGetVar("VIEWCTR", &rbViewCtr) != RTNORM)
                    {
                        return false;
                    }
                    struct resbuf UCS, WCS;
                    WCS.restype = RTSHORT;
                    WCS.resval.rint = 0;
                    UCS.restype = RTSHORT;
                    UCS.resval.rint = 1;

                    acedTrans(rbViewCtr.resval.rpoint, &UCS, &WCS, 0,
rbViewCtr.resval.rpoint);
                    viewCenterPt = asPnt3d(rbViewCtr.resval.rpoint);

                    // 设置视口角点坐标
```

```
                ptMin[X] = viewCenterPt[X] - viewWidth / 2;
                ptMin[Y] = viewCenterPt[Y] - viewHeight / 2;
                ptMax[X] = viewCenterPt[X] + viewWidth / 2;
                ptMax[Y] = viewCenterPt[Y] + viewHeight / 2;

                return true;
        }
```

上面的代码中，分别用 VIEWSIZE、SCREEN 和 VIEWCTR 系统变量来获得视口的高度、宽度和中心点坐标，然后再计算视口角点的坐标。

ZoomExtents 函数在 4.4 节已经介绍过，其实现代码为：

```
        void CPubFunction::ZoomExtents()
        {
                AcDbBlockTable *pBlkTbl;
                AcDbBlockTableRecord *pBlkTblRcd;
                acdbHostApplicationServices()->workingDatabase()
                        ->getBlockTable(pBlkTbl, AcDb::kForRead);
                pBlkTbl->getAt(ACDB_MODEL_SPACE, pBlkTblRcd,
AcDb::kForRead);
                pBlkTbl->close();

                // 获得当前图形中所有实体的最小包围盒
                AcDbExtents extent;
                extent.addBlockExt(pBlkTblRcd);
                pBlkTblRcd->close();

                // 计算长方形的顶点
                ads_point pt[7];
                pt[0][X] = pt[3][X] = pt[4][X] = pt[7][X] = extent.minPoint().x;
                pt[1][X] = pt[2][X] = pt[5][X] = pt[6][X] = extent.maxPoint().x;
                pt[0][Y] = pt[1][Y] = pt[4][Y] = pt[5][Y] = extent.minPoint().y;
                pt[2][Y] = pt[3][Y] = pt[6][Y] = pt[7][Y] = extent.maxPoint().y;
                pt[0][Z] = pt[1][Z] = pt[2][Z] = pt[3][Z] = extent.maxPoint().z;
                pt[4][Z] = pt[5][Z] = pt[6][Z] = pt[7][Z] = extent.minPoint().z;

                // 将长方体的所有角点转移到DCS中
                struct resbuf wcs, dcs; // 转换坐标时使用的坐标系统标记
                wcs.restype = RTSHORT;
                wcs.resval.rint = 0;
                dcs.restype = RTSHORT;
```

```
        dcs.resval.rint = 2;
        acedTrans(pt[0], &wcs, &dcs, 0, pt[0]);
        acedTrans(pt[1], &wcs, &dcs, 0, pt[1]);
        acedTrans(pt[2], &wcs, &dcs, 0, pt[2]);
        acedTrans(pt[3], &wcs, &dcs, 0, pt[3]);
        acedTrans(pt[4], &wcs, &dcs, 0, pt[4]);
        acedTrans(pt[5], &wcs, &dcs, 0, pt[5]);
        acedTrans(pt[6], &wcs, &dcs, 0, pt[6]);
        acedTrans(pt[7], &wcs, &dcs, 0, pt[7]);

        // 获得所有角点在DCS中最小的包围矩形
        double xMax = pt[0][X], xMin = pt[0][X];
        double yMax = pt[0][Y], yMin = pt[0][Y];
        for (int i = 1; i <= 7; i++)
        {
            if (pt[i][X] > xMax)
                xMax = pt[i][X];
            if (pt[i][X] < xMin)
                xMin = pt[i][X];
            if (pt[i][Y] > yMax)
                yMax = pt[i][Y];
            if (pt[i][Y] < yMin)
                yMin = pt[i][Y];
        }

        AcDbViewTableRecord view = GetCurrentView();

        // 设置视图的中心点
        view.setCenterPoint(AcGePoint2d((xMin + xMax) / 2,
            (yMin + yMax) / 2));

        // 设置视图的高度和宽度
        view.setHeight(fabs(yMax - yMin));
        view.setWidth(fabs(xMax - xMin));

        // 将视图对象设置为当前视图
        Acad::ErrorStatus es = acedSetCurrentView(&view, NULL);
    }
```

上面的代码中用到了一个自定义函数 GetCurrentView，用于获得当前视图，其实现代码

为：

```
AcDbViewTableRecord CPubFunction::GetCurrentView()
{
        AcDbViewTableRecord view;
        struct resbuf rb;
        struct resbuf wcs, ucs, dcs;  // 转换坐标时使用的坐标系统标记

        wcs.restype = RTSHORT;
        wcs.resval.rint = 0;
        ucs.restype = RTSHORT;
        ucs.resval.rint = 1;
        dcs.restype = RTSHORT;
        dcs.resval.rint = 2;

        // 获得当前视口的"查看"模式
        acedGetVar("VIEWMODE", &rb);
        view.setPerspectiveEnabled(rb.resval.rint & 1);
        view.setFrontClipEnabled(rb.resval.rint & 2);
        view.setBackClipEnabled(rb.resval.rint & 4);
        view.setFrontClipAtEye(!(rb.resval.rint & 16));

        // 当前视口中视图的中心点（UCS坐标）
        acedGetVar("VIEWCTR", &rb);
        acedTrans(rb.resval.rpoint, &ucs, &dcs, 0, rb.resval.rpoint);
        view.setCenterPoint(AcGePoint2d(rb.resval.rpoint[X],
                rb.resval.rpoint[Y]));

        // 当前视口透视图中的镜头焦距长度（单位为毫米）
        acedGetVar("LENSLENGTH", &rb);
        view.setLensLength(rb.resval.rreal);

        // 当前视口中目标点的位置（以 UCS 坐标表示）
        acedGetVar("TARGET", &rb);
        acedTrans(rb.resval.rpoint, &ucs, &wcs, 0, rb.resval.rpoint);
        view.setTarget(AcGePoint3d(rb.resval.rpoint[X],
                rb.resval.rpoint[Y], rb.resval.rpoint[Z]));

        // 当前视口的观察方向（UCS）
        acedGetVar("VIEWDIR", &rb);
```

```
acedTrans(rb.resval.rpoint, &ucs, &wcs, 1, rb.resval.rpoint);
view.setViewDirection(AcGeVector3d(rb.resval.rpoint[X],
        rb.resval.rpoint[Y], rb.resval.rpoint[Z]));

// 当前视口的视图高度（图形单位）
acedGetVar("VIEWSIZE", &rb);
view.setHeight(rb.resval.rreal);
double height = rb.resval.rreal;

// 以像素为单位的当前视口的大小（X 和 Y 值）
acedGetVar("SCREENSIZE", &rb);
view.setWidth(rb.resval.rpoint[X] / rb.resval.rpoint[Y] * height);

// 当前视口的视图扭转角
acedGetVar("VIEWTWIST", &rb);
view.setViewTwist(rb.resval.rreal);

// 将模型选项卡或最后一个布局选项卡置为当前
acedGetVar("TILEMODE", &rb);
int tileMode = rb.resval.rint;
// 设置当前视口的标识码
acedGetVar("CVPORT", &rb);
int cvport = rb.resval.rint;

// 是否是模型空间的视图
bool paperspace = ((tileMode == 0) && (cvport == 1)) ? true : false;
view.setIsPaperspaceView(paperspace);

if (!paperspace)
{
        // 当前视口中前向剪裁平面到目标平面的偏移量
        acedGetVar("FRONTZ", &rb);
        view.setFrontClipDistance(rb.resval.rreal);

        // 获得当前视口后向剪裁平面到目标平面的偏移值
        acedGetVar("BACKZ", &rb);
        view.setBackClipDistance(rb.resval.rreal);
}
else
```

```
        {
            view.setFrontClipDistance(0.0);
            view.setBackClipDistance(0.0);
        }

        return view;
    }
```

函数 WindowZoom 用于执行窗口缩放，其实现代码为：

```
    Acad::ErrorStatus CPubFunction::WindowZoom(const AcGePoint2d &ptMin,
        const AcGePoint2d &ptMax, double scale)
    {
        AcDbViewTableRecord view;
        AcGePoint2d ptCenter2d((ptMin[X] + ptMax[X]) / 2,
            (ptMin[Y] + ptMax[Y]) / 2);

        view.setCenterPoint(ptCenter2d);
        view.setWidth((ptMax[X] - ptMin[X]) / scale);
        view.setHeight((ptMax[Y] - ptMin[Y]) / scale);

        Acad::ErrorStatus es = acedSetCurrentView(&view, NULL);

        return es;
    }
```

需要注意的是，这里所介绍的几个函数仅能适用于在 WCS 坐标系的 XOY 平面上操作的情形（处理二维图形时），对于三维空间中任何平面是不能使用的，读者可以在此基础上进行完善，使其能够支持三维空间中的操作。

2. 快速获得需要的组码

使用选择集的过程中，最灵活的就是过滤器的构建，通过构建不同类型的过滤器，可以使用选择集获得符合多种条件的实体。但是，过滤器和 DXF 组码是紧紧联系在一起的，不了解组码知识，要灵活创建过滤器非常困难。

实际上，你需要掌握的仅仅是本节列出的常用组码，其他的组码都可以通过下面的函数定义的命令 EntInfo 来获得（当然，也可以直接在命令行输入(entget(car(entsel)))并按下 Enter 键）：

```
    void ZffCHAP5EntInfo()
    {
        acDocManager->sendStringToExecute(acDocManager->curDocumen
t(),
            "(entget(car(entsel))) ");    // 字符串的末尾包含一个空字符
```

```
}
```

其中，acDocManager 是一个用#define 语句定义的宏，等效于全局函数 acDocManagerPtr，此函数返回指向文档管理器的指针。curDocument 函数返回当前活动的图形文档，sendStringToExecute 函数将一个字符串发送到指定文档的命令行，在字符串的末尾加上一个空格就可以执行该字符串，该函数的定义为：

```
virtual Acad::ErrorStatus sendStringToExecute(
                AcApDocument* pAcTargetDocument,
                const char * pszExecute,
                bool bActivate = true,
                bool bWrapUpInactiveDoc = false,
                bool bEchoString = true) = 0;
```

其中，pAcTargetDocument 指定要发送字符串的目标文档；pszExecute 指定要发送的字符串内容；bActivate 说明是否将目标文档设置为当前活动文档；bWrapUpInactiveDoc 指出是否把当前活动文档排入队列，待切换文档发生 OnIdle 消息时完成；bEchoString 指出是否在命令行显示发送的字符串。

前面多次以圆作例子，那么现在就来看看如何使用EntInfo命令来获得圆心、半径的组码。在图形窗口中创建一个圆心在（50，50，0）、半径为 30 的圆，然后执行EntInfo命令，在图形窗口中选择创建的圆，能够在命令窗口中得到如图 5.19所示的结果。

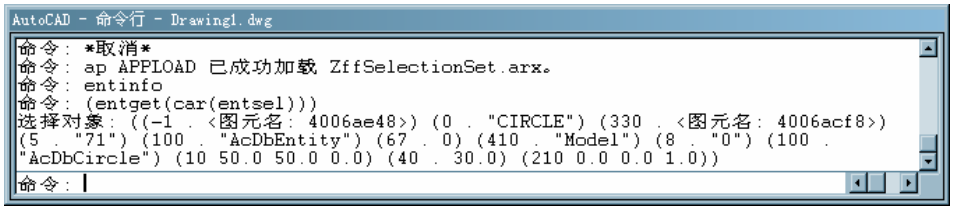

图5.19　查看圆的 DXF 组码

组码和其对应的参数值成对出现，形如（组码. 参数值）的格式。从图中可以获得几个常用的组码，例如对象类型（0. “CIRCLE”）的组码 0，所在空间（67. 0）的组码 67，图层（8. “0”）的组码 8。由于我们已经知道圆的圆心和半径，自然就很容易找到圆心的组码 10 和半径的组码 40。

OK，现在可以去构建自己想要的过滤器了。

第6章 扩展数据、扩展记录和对象字典

在编程中，某些时候不可避免地要向图形中添加一些用户数据，例如，将一条直线解释为输电线、道路中心线，或者其他类型的对象，也可能要将当前图形的编号随图形一起保存起来。

要向图形中的实体追加一些数据，可以使用扩展数据或者扩展记录；要向图形本身追加一些数据，则可以使用命名对象字典。

6.1 扩展数据

6.1.1 说明

曾经应邀写过一个对象解释的程序，就是对于同一种类型的实体，可能将其作为管线、道路中心线和建筑物轮廓线等。为了实现这个目的，就可以在实体上追加扩展数据，要获得某个实体类型的时候，就可以读取其扩展数据得到类型信息。

本节演示了向实体追加扩展数据和显示一个实体的扩展数据，并且加入了容错的处理：向已经包含扩展数据的实体添加扩展数据，程序会自动退出；要求显示不包含任何扩展数据的实体的扩展数据，程序同样会退出。

6.1.2 思路

扩展数据能被添加到任何实体上，由一个结果缓冲区链表组成，并且随 AutoCAD 图形一起保存（AutoCAD 不会使用扩展数据）。在许多情况下，扩展数据是向实体追加用户数据的一个有效途径，但是每个实体上所附加的扩展数据不能超过16K。

AcDbObject 类的 setXData 函数用于设置一个对象的扩展数据，其定义为：

```
virtual Acad::ErrorStatus
AcDbObject::setXData(const resbuf* xdata);
```

AcDbObject 类的 xData 函数用于获取一个对象的扩展数据，其定义为：

```
virtual resbuf*
AcDbObject::xData(const char* regappName = NULL) const;
```

任何一个应用程序都能将扩展数据附加到实体上，因此所有的扩展数据都需要一个惟一的应用程序名称，该名称不超过31个字符。为了注册一个应用程序，可以使用全局函数 acdbRegApp。

6.1.3 步骤

（1）在 VC++ 6.0中，使用 ObjectARX 向导创建一个新的项目，命名为 Xdata。注册一个命令 AddXData，用于向实体追加指定的扩展数据，其实现函数为：

```
void ZffCHAP5AddXData()
{
        // 提示用户选择所要添加扩展数据的图形对象
        ads_name en;
        ads_point pt;

        if (acedEntSel("\n选择所要添加扩展数据的实体：", en, pt) != RTNORM)
                return;

        AcDbObjectId entId;
        Acad::ErrorStatus es = acdbGetObjectId(entId, en);

        // 扩展数据的内容
        struct resbuf* pRb;
        char appName[] = {"XData"};
        char typeName[] = {"道路中心线"};

        // 注册应用程序名称
        acdbRegApp("XData");

        // 创建结果缓冲区链表
        pRb = acutBuildList(AcDb::kDxfRegAppName, appName,          //
应用程序名称
                AcDb::kDxfXdAsciiString, typeName,                  //
字符串
                AcDb::kDxfXdInteger32, 2,
        // 整数
                AcDb::kDxfXdReal, 3.14,                             //
实数
                AcDb::kDxfXdWorldXCoord, pt,                        // 点坐
标值
                RTNONE);

        // 为选择的实体添加扩展数据
        AcDbEntity *pEnt;
        acdbOpenAcDbEntity(pEnt, entId, AcDb::kForWrite);
```

```
        struct resbuf *pTemp;
        pTemp = pEnt->xData("XData");
        if (pTemp != NULL)          // 如果已经包含扩展数据，就不再添加新的
扩展数据
        {
            acutRelRb(pTemp);
            acutPrintf("\n所选择的实体已经包含扩展数据！");
        }
        else
        {
            pEnt->setXData(pRb);
        }

        pEnt->close();
        acutRelRb(pRb);
    }
```

在构建存储扩展数据的结果缓冲区时，除了应用程序名称对应的数据类型为AcDb::kDxfRegAppName之外，其他的数据类型前缀均为AcDb::kDxfXd，所有的与扩展数据有关的数据类型均带有这个前缀。

在函数结束之前，记得删除结果缓冲区。

（2）注册一个ViewXData命令，用于查看指定的实体的扩展数据，其实现函数为：

```
    void ZffCHAP5ViewXData()
    {
        // 提示用户选择所要查看扩展数据的图形对象
        ads_name en;
        ads_point pt;

        if (acedEntSel("\n选择所要查看扩展数据的实体：", en, pt) != RTNORM)
            return;

        AcDbObjectId entId;
        Acad::ErrorStatus es = acdbGetObjectId(entId, en);

        // 打开图形对象，查看是否包含扩展数据
        AcDbEntity *pEnt;
        acdbOpenAcDbEntity(pEnt, entId, AcDb::kForRead);
        struct resbuf *pRb;
        pRb = pEnt->xData("XData");
```

```
                pEnt->close();

                if (pRb != NULL)
                {
                    // 在命令行显示所有的扩展数据
                    struct resbuf *pTemp;
                    pTemp = pRb;

                    // 首先要跳过应用程序的名称这一项
                    pTemp = pTemp->rbnext;
                    acutPrintf("\n字符串类型的扩展数据是：%s",
pTemp->resval.rstring);

                    pTemp = pTemp->rbnext;
                    acutPrintf("\n整数类型的扩展数据是：%d", pTemp->resval.rint);

                    pTemp = pTemp->rbnext;
                    acutPrintf("\n实数类型的扩展数据是：%.2f",
pTemp->resval.rreal);

                    pTemp = pTemp->rbnext;
                    acutPrintf("\n点坐标类型的扩展数据是：(%.2f, %.2f, %.2f)",
                        pTemp->resval.rpoint[X], pTemp->resval.rpoint[Y],
                        pTemp->resval.rpoint[Z]);

                    acutRelRb(pRb);
                }
                else
                {
                    acutPrintf("\n所选择的实体不包含任何的扩展数据！");
                }
        }
```

使用 AcDbObject 类的 xData 函数能够获得一个结果缓冲区链表，该实体的所有扩展数据都保存在该链表中，因此可以通过遍历结果缓冲区的方法获得扩展数据。

6.1.4 效果

（1）编译运行程序，在 AutoCAD 2002中，使用 LINE 命令创建两条直线。执行 AddXData 命令，选择其中的一条直线，为其添加扩展数据。

（2）再次执行 AddXData 命令，仍然选择已经添加扩展数据的那条直线，系统会在命令

行给出提示：所选择的实体已经包含扩展数据！

（3）执行 ViewXData 命令，选择未添加扩展数据的直线，系统会在命令窗口给出提示：所选择的实体不包含任何的扩展数据！

（4）再次执行 ViewXData 命令，选择已经添加扩展数据的直线，系统会在命令窗口提示：

命令: VIEWXDATA
选择所要查看扩展数据的实体：
字符串类型的扩展数据是：道路中心线
整数类型的扩展数据是：2
实数类型的扩展数据是：3.14
点坐标类型的扩展数据是：(561.77, 302.57, 0.00)

6.1.5 小结

学习本节内容之后，读者应该掌握下面的要点：

- 向实体追加各种类型的扩展数据。
- 判断对象是否已经包含扩展数据。
- 遍历结果缓冲区链表的方法。

6.2 扩展字典和有名对象字典

6.2.1 说明

扩展记录与扩展数据类似，但是其数据存储量和能够存储的数据类型都要多于扩展数据。扩展记录可以保存在实体的扩展字典或有名对象字典中。

扩展字典隶属于特定的实体，每个实体只能包含一个扩展字典，它为实体保存自定义数据提供了一种途径，实际上，Autodesk 也建议开发者用扩展字典和扩展记录来代替传统的扩展数据。有名对象字典直接保存在图形数据库中，不与特定的实体有关，因此可用于保存与实体无关的设计参数。

字典与符号表类似，其中包含一个惟一的字符串关键字索引和对象 ID 号，通过关键字来访问字典中保存的内容。

本节的实例分别在实体和图形中保存与上节实例相同的数据，让读者对比扩展数据和扩展记录使用的异同，更好地理解这两种保存数据的机制。

6.2.2 思路

1. 访问实体的扩展字典

扩展字典与特定的实体关联，但是一个实体在默认情况下不包含扩展字典，如果要利用扩展字典保存与实体关联的数据，可以使用 createExtensionDictionary 函数为实体建立扩展字典，如果实体已经包含扩展字典，该函数的调用不会产生任何影响。

创建扩展字典之后，就可以使用 extensionDictionary 函数获得实体的扩展字典，AcDbDictionary 对象的 setAt 函数可以为字典添加一个元素，该元素既可以是 AcDbXrecord，也可以是其他类型的对象。如果添加了 AcDbXrecord（扩展记录），就可以使用 acutBuildList 函数构建一个保存数据的结果缓冲区链表，然后使用 setFromRbChain 函数将结果缓冲区链表添加到扩展及记录中，这样就使用扩展记录保存了数据。

如果要访问实体扩展字典中保存的扩展记录，可以使用 extensionDictionary 函数获得实体的扩展字典，然后通过字典的 getAt 函数得到指定的元素（扩展记录），使用 AcDbXrecord 类的 rbChain 函数得到保存数据的结果缓冲区链表，并且遍历该链表获得保存的数据。

2. 访问有名对象字典

AutoCAD 每个图形数据库中都包含一个有名对象字典，默认情况下该字典中包含了组、多线样式、布局和打印等信息。例如，用户在 AutoCAD 创建一个组，就会有一个代表改组的元素被添加到组字典中。

如果需要在有名对象字典中保存自己的数据，一般可以在有名对象字典中添加一个根字典，然后再向根字典中添加新的字典，进而在新字典中保存数据。这样的好处是不会与有名对象字典的基本字典相混淆。

使用 AcDbDatabase 对象的 getNamedObjectsDictionary 函数可以获得图形的有名对象字典（根字典），可以通过 setAt 函数向根字典添加一个字典，或者通过 getAt 函数获得其中的一个字典。获得字典之后，向字典中保存数据的方法与扩展字典完全一致。

6.2.3 步骤

（1）启动 VC++ 6.0，使用 ObjectARX 向导创建一个名为 Xrecord 的新工程。注册一个新命令 AddXRecord，提示用户选择一个实体，并将一些附加的数据保存到该实体的扩展字典中，其实现函数为：

```
void ZffCHAP5AddXRecord()
{
        // 提示用户选择所要添加扩展记录的图形对象
        ads_name en;
        ads_point pt;

        if (acedEntSel("\n选择所要添加扩展记录的实体：", en, pt) != RTNORM)
            return;
```

```
    AcDbObjectId entId;          // 要添加扩展记录的实体ID
    Acad::ErrorStatus es = acdbGetObjectId(entId, en);

    AcDbXrecord *pXrec = new AcDbXrecord;
    AcDbObject *pObj;
    AcDbObjectId dictObjId, xRecObjId;
    AcDbDictionary *pDict;

    // 要在扩展记录中保存的字符串
    char entType[] = {"直线"};
    struct resbuf* pRb;

    // 向实体中添加扩展字典
    acdbOpenObject(pObj, entId, AcDb::kForWrite);
    pObj->createExtensionDictionary();
    dictObjId = pObj->extensionDictionary();
    pObj->close();

    // 向扩展字典中添加一条记录
    acdbOpenObject(pDict, dictObjId, AcDb::kForWrite);
    pDict->setAt("XRecord", pXrec, xRecObjId);
    pDict->close();

    // 设置扩展记录的内容
    pRb = acutBuildList(AcDb::kDxfText, entType,
                        AcDb::kDxfInt32, 12,
                        AcDb::kDxfReal, 3.14,
                        AcDb::kDxfXCoord, pt,
                        RTNONE);
    pXrec->setFromRbChain(*pRb);
    pXrec->close();

    acutRelRb(pRb);
}
```

在上面的代码中，值得注意的是 AcDbXrecord 类的成员函数 setFromRbChain 的定义：

```
Acad::ErrorStatus setFromRbChain(
    const resbuf& pRb,
    AcDbDatabase* auxDb = NULL);
```

因此，传递的第一个参数必须是 pRb 所指向的结果缓冲区对象，也就是*pRb。

（2）注册一个新命令 ViewXRecord，提示用户选择图形对象，如果选中的实体包含扩展字典，就查看扩展字典中“Xrecord”元素所保存的数据，其实现函数为：

```
void ZffCHAP5VIEWXRECORD()
{
        // 提示用户选择所要查看扩展记录的图形对象
        ads_name en;
        ads_point pt;

        if (acedEntSel("\n选择所要查看扩展记录的实体：", en, pt) != RTNORM)
                return;

        AcDbObjectId entId;
        Acad::ErrorStatus es = acdbGetObjectId(entId, en);

        // 打开图形对象，获得实体扩展字典的ObjectId
        AcDbEntity *pEnt;
        acdbOpenAcDbEntity(pEnt, entId, AcDb::kForRead);
        AcDbObjectId dictObjId = pEnt->extensionDictionary();
        pEnt->close();

        // 查看实体是否包含扩展字典
        if (dictObjId == AcDbObjectId::kNull)
        {
                acutPrintf("\n所选择的实体不包含扩展字典!");
                return;
        }

        // 打开扩展字典，获得与关键字“XRecord”关联的扩展记录
        AcDbDictionary *pDict;
        AcDbXrecord *pXrec;
        acdbOpenObject(pDict, dictObjId, AcDb::kForRead);
        pDict->getAt("XRecord", (AcDbObject*&)pXrec, AcDb::kForRead);
        pDict->close();

        // 获得扩展记录的数据链表并关闭扩展数据对象
        struct resbuf *pRb;
        pXrec->rbChain(&pRb);
        pXrec->close();
```

```
        if (pRb != NULL)
        {
            // 在命令行显示扩展记录内容
            struct resbuf *pTemp;
            pTemp = pRb;

            acutPrintf("\n字符串类型的扩展数据是：%s",
pTemp->resval.rstring);

            pTemp = pTemp->rbnext;
            acutPrintf("\n整数类型的扩展数据是：%d", pTemp->resval.rint);

            pTemp = pTemp->rbnext;
            acutPrintf("\n实数类型的扩展数据是：%.2f",
pTemp->resval.rreal);

            pTemp = pTemp->rbnext;
            acutPrintf("\n点坐标类型的扩展数据是：(%.2f, %.2f, %.2f)",
                pTemp->resval.rpoint[X], pTemp->resval.rpoint[Y],
                pTemp->resval.rpoint[Z]);

            acutRelRb(pRb);
        }
    }
```

需要注意的是，AcDbXrecord 类的 rbChain 函数定义为：

```
Acad::ErrorStatus rbChain(
        resbuf** ppRb,
        AcDbDatabase* auxDb = NULL) const;
```

第一个参数是一个指针的指针，帮助系统中已经说明要输入一个指向结果缓冲区的指针的地址（Input the address of a pointer to resbuf structure），因此应该输入&pRb 作为参数值。

（3）注册一个新命令 AddNameDict，向当前的图形数据库中添加一个字典，并在其中保存自定义数据，其实现函数为：

```
void ZffCHAP5AddNameDict()
{
        // 要在扩展记录中保存的字符串
        char entType[] = {"直线"};
        struct resbuf *pRb;
```

```
                    // 获得有名对象字典，向其中添加指定的字典项
                    AcDbDictionary *pNameObjDict, *pDict;
                    acdbHostApplicationServices()->workingDatabase()
                            ->getNamedObjectsDictionary(pNameObjDict,
AcDb::kForWrite);
                    // 检查所要添加的字典项是否已经存在
                    AcDbObjectId dictObjId;
                    if (pNameObjDict->getAt("MyDict", (AcDbObject*&)pDict,
                            AcDb::kForWrite) == Acad::eKeyNotFound)
                    {
                            pDict = new AcDbDictionary;
                            pNameObjDict->setAt("MyDict", pDict, dictObjId);
                            pDict->close();
                    }
                    pNameObjDict->close();

                    // 向新建的字典中添加一个扩展记录
                    AcDbObjectId xrecObjId;
                    AcDbXrecord *pXrec = new AcDbXrecord;
                    acdbOpenObject(pDict, dictObjId, AcDb::kForWrite);
                    pDict->setAt("XRecord", pXrec, xrecObjId);
                    pDict->close();

                    // 设置扩展记录的内容
                    ads_point pt;
                    pt[X] = 100;
                    pt[Y] = 100;
                    pt[Z] = 0;
                    pRb = acutBuildList(AcDb::kDxfText, entType,
                            AcDb::kDxfInt32, 12,
                            AcDb::kDxfReal, 3.14,
                            AcDb::kDxfXCoord, pt,
                            RTNONE);
                    pXrec->setFromRbChain(*pRb);
                    pXrec->close();

                    acutRelRb(pRb);
            }
```

从上面的代码可以看出，在有名对象字典中保存数据与在扩展字典中保存数据相比，仅

仅是获得字典的方法有所不同。

（4）注册一个新命令 ViewNameDict，检查当前图形中是否包含指定的用户字典，并在命令窗口中显示字典中保存的自定义数据，其实现函数为：

```
void ZffCHAP5ViewNameDict()
{
        // 获得对象有名字典中指定的字典项
        AcDbDictionary *pNameObjDict, *pDict;
        Acad::ErrorStatus es;
        acdbHostApplicationServices()->workingDatabase()
                ->getNamedObjectsDictionary(pNameObjDict,
AcDb::kForRead);
        es = pNameObjDict->getAt("MyDict", (AcDbObject*&)pDict,
AcDb::kForRead);
        pNameObjDict->close();

        // 如果不存在指定的字典项，退出程序
        if (es == Acad::eKeyNotFound)
                return;

        // 获得指定的对象字典
        AcDbXrecord *pXrec;
        pDict->getAt("XRecord", (AcDbObject*&)pXrec, AcDb::kForRead);
        pDict->close();

        // 获得扩展记录的数据链表并关闭扩展数据对象
        struct resbuf *pRb;
        pXrec->rbChain(&pRb);
        pXrec->close();

        if (pRb != NULL)
        {
                // 在命令行显示扩展记录内容
                struct resbuf *pTemp;
                pTemp = pRb;

                acutPrintf("\n字符串类型的扩展数据是：%s",
pTemp->resval.rstring);

                pTemp = pTemp->rbnext;
```

```
                    acutPrintf("\n整数类型的扩展数据是：%d", pTemp->resval.rint);

                    pTemp = pTemp->rbnext;
                    acutPrintf("\n实数类型的扩展数据是：%.2f",
pTemp->resval.rreal);

                    pTemp = pTemp->rbnext;
                    acutPrintf("\n点坐标类型的扩展数据是：(%.2f, %.2f, %.2f)",
                        pTemp->resval.rpoint[X], pTemp->resval.rpoint[Y],
                        pTemp->resval.rpoint[Z]);

                    acutRelRb(pRb);
                }
        }
```

6.2.4 效果

（1）编译链接程序，启动 AutoCAD 2002，加载生成的 ARX 文件。执行 AddXRecord 命令，在图形窗口中选择一个实体，即可为其添加扩展字典并在其中保存自定义数据。

（2）执行ViewXRecord命令，在图形窗口中选择已经添加了扩展字典的实体，在命令窗口中能够得到如图6.1所示的结果。

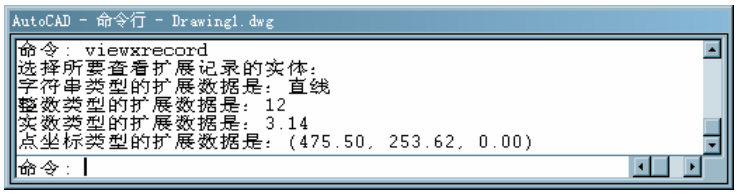

图6.1　查看扩展字典中保存的数据

（3）执行AddNameDict命令向图形的有名对象字典中添加用户字典，并且在其中保存自定义数据。执行ViewNameDict命令查看用户字典中保存的数据，能够在命令窗口中得到如图6.2所示的结果。

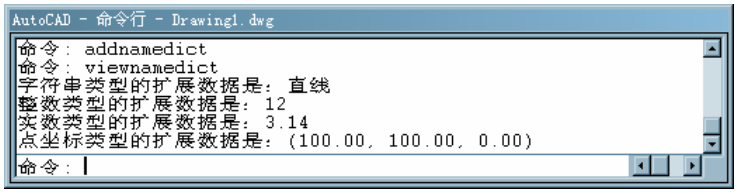

图6.2　查看有名对象字典中保存的数据

更有意思的是，即使在执行ViewXRecord和ViewNameDict命令之前保存并关闭了图形，再次打开图形仍然可以查询到这些数据。实际上，扩展记录所保存的数据已经被保存在图形

数据库中，可以使用前面介绍的数据库监视器来查看有名对象字典和扩展字典中保存数据的结构，如图6.3所示。

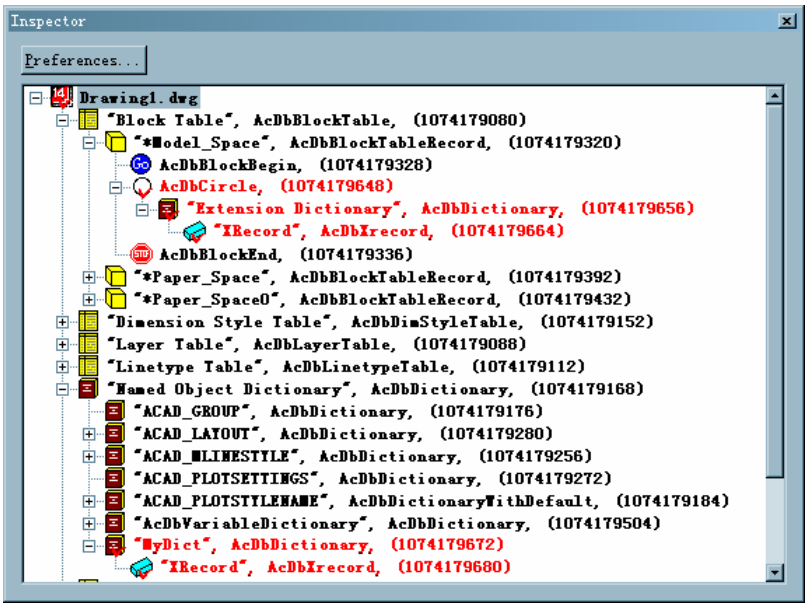

图6.3　使用数据库监视器查看扩展记录

6.2.5 小结

学习本节内容之后，读者需要掌握下面的几个知识点：

- 使用扩展字典来保存对象的附加数据。
- 使用对象命名字典来保存与对象无关联的数据。

扩展记录是 AutoCAD 设计用来代替扩展数据的，因此在向对象附加数据的时候，应优先考虑使用扩展记录。

6.3 组字典

6.3.1 说明

本节的实例演示编组的创建和删除。在 AutoCAD 中，可以使用 Group 命令来创建和删除编组，但是在 ObjectARX 中就没有直接的函数来创建和删除组，必须通过组的底层实现来操作。

6.3.2 思路

前面已经介绍，编组的信息保存在图形数据库的有名对象字典中，在 ObjectARX 中对编组的操作实际上就是对有名对象字典中组字典的操作，可以通过 AcDbDatabase 对象的 getGroupDictionary 函数获得组字典的指针。

组字典在有名对象字典的根字典中的关键字是“ACAD_GROUP”，因此还可以通过有名对象字典的 getAt 函数来获得组字典的指针，参考下面的代码：

```
AcDbDictionary *pNameDict;
acdbHostApplicationServices()->workingDatabase()
->getNamedObjectsDictionary(pNameDict, AcDb::kForRead);
pNameDict->getAt("ACAD_GROUP", (AcDbObject*&)pGroupDict,
    AcDb::kForWrite);
pNameDict->close();
```

当然一般情况下都不会用上面的代码来获得组字典，提供这段代码只是为了让读者更好地理解组字典与根字典的关系。

6.3.3 步骤

（1）在 VC++ 6.0中，使用 ObjectARX 向导创建一个新工程，其名称为 Group。注册一个新命令 AddGroup，提示用户选择几个实体，将其创建成一个编组，实现函数为：

```
void ZffCHAP6AddGroup()
{
    // 提示用户选择多个实体
    ads_name sset;
    acutPrintf("\n选择要成组的实体:");
    if (acedSSGet(NULL, NULL, NULL, NULL, sset) != RTNORM)
        return;

    long length;
    acedSSLength(sset, &length);
    AcDbObjectIdArray objIds;
    for (long i = 0; i < length; i++)
    {
        // 获得指定元素的ObjectId
        ads_name ent;
        acedSSName(sset, i, ent);
        AcDbObjectId objId;
        acdbGetObjectId(objId, ent);

        // 获得指向当前元素的指针
```

```
            AcDbEntity *pEnt;
            acdbOpenObject(pEnt, objId, AcDb::kForRead);
            objIds.append(pEnt->objectId());

            pEnt->close();
        }

        acedSSFree(sset);

        // 创建组
        char groupName[] = {"MyGroup"};
        CreateGroup(objIds, groupName);
    }
```

其中，CreateGroup 是一个自定义函数，能够根据指定的 ObjectId 数组创建编组，其实现代码为：

```
    void CreateGroup(AcDbObjectIdArray& objIds, char* pGroupName)
    {
        AcDbGroup *pGroup = new AcDbGroup(pGroupName);
        for (int i = 0; i < objIds.length(); i++) {
            pGroup->append(objIds[i]);
        }

        // 将组添加到有名对象字典的组字典中
        AcDbDictionary *pGroupDict;
        acdbHostApplicationServices()->workingDatabase()
            ->getGroupDictionary(pGroupDict, AcDb::kForWrite);

        AcDbObjectId pGroupId;
        pGroupDict->setAt(pGroupName, pGroup, pGroupId);
        pGroupDict->close();
        pGroup->close();
    }
```

（2）注册一个新命令 DelGroup，用于删除图形中的编组“MyGroup”，其实现函数为：

```
    void ZffCHAP6DelGroup()
    {
            // 获得组字典
        AcDbDictionary *pGroupDict;
        acdbHostApplicationServices()->workingDatabase()
            ->getGroupDictionary(pGroupDict, AcDb::kForWrite);
```

```
        if (pGroupDict->has("MyGroup"))
                {
                        pGroupDict->remove("MyGroup");
                }
        pGroupDict->close();
}
```

6.3.4 效果

（1）编译连接程序，运行 AutoCAD 2002，加载生成的 ARX 文件，执行 AddGroup 命令，在图形窗口中选择若干个实体并按下 Enter 键，就能完成组的创建。

（2）在命令窗口执行Group命令，系统会弹出如图6.4所示的对话框，其中显示了创建的编组MyGroup。

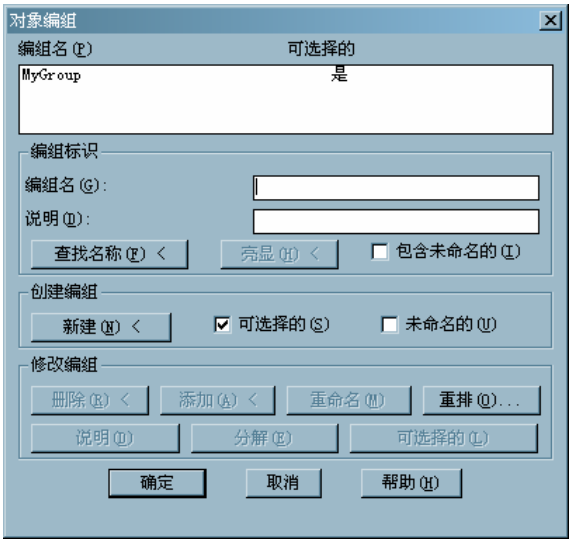

图6.4　查看生成的编组

（3）执行 DelGroup 命令，就能从图形中删除编组 MyGroup。当然，删除编组的时候，组成编组的实体不会从图形中删除。再次执行 Group 命令，【对象编组】对话框中就不再包含 MyGroup 编组。

6.3.5 小结

组字典是有名对象字典的一个字典，它的使用方法与用户字典类似，可以使用数据库监视器来了解组字典在有名对象字典中的保存结构。

6.4 多线样式字典

6.4.1 说明

多线在用于创建一组直线时特别有用，例如绘制建筑平面图时的墙线和轴线就可以用一条多线来表示。在创建程序时，某些情况下就有必要创建新的多线样式，本节的实例演示多线样式的创建和删除。

6.4.2 思路

多线样式在图形数据库中同样是通过有名对象字典保存，其对应的是有名对象字典中的“ACAD_MLINESTYLE”字典。除了 AcDbMlineStyle 对象的创建，使用有名对象字典保存多线样式的方法与创建组类似，因此这里把注意力放在 AcDbMlineStyle 对象的创建方面。

要创建一个 AcDbMlineStyle 对象（多线样式），至少需要下面的四个步骤：

（1）创建新的对象，并且用指针指向分配的内存空间：

```
AcDbMlineStyle *pMlStyle = new AcDbMlineStyle;
```

（2）使用 initMlineStyle 函数初始化多线样式。

（3）使用 setName 函数设置多线样式的名称。

（4）使用 addElement 函数添加新的元素，该函数被定义为：

```
Acad::ErrorStatus addElement(
                int& index,
                double offset,
                const AcCmColor & color,
                AcDbObjectId linetypeId,
                bool checkIfReferenced = true);
```

其中，index 返回从0开始的元素索引；offset 指定添加的元素的偏移量；color 指定元素的颜色；linetypeId 指定添加元素的线型；checkIfReferenced 在帮助文档中并未说明其含义，可以不必考虑。

此外，还可以使用 AcDbMlineStyle 类的其他成员函数对多线样式进行修改，例如 setElement 函数用于修改元素的样式，setFillColor 函数用于设置多线内部的填充颜色，等等。

6.4.3 步骤

（1）启动 VC++ 6.0，使用 ObjectARX 向导创建一个新工程，其名称为 MlineStyle。注册一个新命令 AddMlStyle，用于添加一个新的多线样式，其实现函数为：

```
void ZffCHAP6AddMlStyle()
{
                // 加载线型（两种方法）
                Acad::ErrorStatus es;
```

```
        es = acdbHostApplicationServices()->workingDatabase()
                ->loadLineTypeFile("CENTER", "acadiso.lin");
        es = acdbLoadLineTypeFile("HIDDEN", "acadiso.lin",
                acdbHostApplicationServices()->workingDatabase());

        // 创建新的AcDbMlineStyle对象
        AcDbMlineStyle *pMlStyle = new AcDbMlineStyle;
        pMlStyle->initMlineStyle();
        pMlStyle->setName("NewStyle");

        int index;          // 多线样式中的元素索引
        AcCmColor color; // 颜色
        AcDbObjectId linetypeId;    // 线型的ID

        // 添加第一个元素（红色的中心线）
        color.setColorIndex(1);       // 红色
        GetLinetypeId("CENTER", linetypeId);
        pMlStyle->addElement(index, 0, color, linetypeId);

        // 添加第二个元素（蓝色的虚线）
        color.setColorIndex(5);       // 蓝色
        GetLinetypeId("HIDDEN", linetypeId);
        pMlStyle->addElement(index, 0.5, color, linetypeId);

        // 添加第三个元素（蓝色的虚线）
        pMlStyle->addElement(index, -0.5, color, linetypeId);

        // 将多线样式添加到多线样式字典中
        AcDbDictionary *pDict;
        acdbHostApplicationServices()->workingDatabase()
                ->getMLStyleDictionary(pDict, AcDb::kForWrite);
        AcDbObjectId mlStyleId;
        es = pDict->setAt("NewStyle", pMlStyle, mlStyleId);
        pDict->close();
        pMlStyle->close();
    }
```

由于在向多线样式添加元素时需要指定其线型，因此首先要加载用到的两个线型，使用全局函数 acdbLoadLineTypeFile 或者 AcDbDatabase 类的 loadLineTypeFile 函数均可以完成线型的加载。

GetLinetypeId 是一个自定义函数，能够根据输入的线型名称获得其 ObjectId，其实现代码为：

```
Acad::ErrorStatus GetLinetypeId(const char *linetype,
                                                AcDbObjectId &linetypeId)
{
        AcDbLinetypeTable *pLtpTbl;
        acdbHostApplicationServices()->workingDatabase()
            ->getLinetypeTable(pLtpTbl, AcDb::kForRead);

        if (!pLtpTbl->has(linetype))
        {
            pLtpTbl->close();
            return Acad::eBadLinetypeName;
        }

        pLtpTbl->getAt(linetype, linetypeId);
        pLtpTbl->close();

        return Acad::eOk;
}
```

ObjectARX 中大多函数的返回值都是 Acad::ErrorStatus 或者 bool 类型，这是为了在调试过程中便于找到错误发生的位置（在调试过程中非常方便），在编写自定义函数中应尽量遵循这一规则。

（2）注册一个新命令 DelMlStyle，用于删除 AddMlStyle 命令创建的多线样式，其实现函数为：

```
void ZffCHAP6DelMlStyle()
{
        // 获得多线样式字典
    AcDbDictionary *pDict;
    acdbHostApplicationServices()->workingDatabase()
        ->getGroupDictionary(pDict, AcDb::kForWrite);

    if (pDict->has("NewStyle"))
        {
            pDict->remove("NewStyle");
        }
    pDict->close();
}
```

（3）为了给程序的使用者带来方便，还可以在 InitApplication 函数中加入命令的提示，

其相关代码为：

```
void InitApplication()
{
        // NOTE: DO NOT edit the following lines.
        //{{AFX_ARX_INIT
        AddCommand("ZFFCHAP6", "ADDMLSTYLE", "ADDMLSTYLE",
ACRX_CMD_TRANSPARENT | ACRX_CMD_USEPICKSET, ZffCHAP6AddMlStyle);
        AddCommand("ZFFCHAP6", "DELMLSTYLE", "DELMLSTYLE",
ACRX_CMD_TRANSPARENT | ACRX_CMD_USEPICKSET, ZffCHAP6DelMlStyle);
        //}}AFX_ARX_INIT

        // 在命令窗口中提示注册命令
        acutPrintf("\n注册AddMlStyle命令,用于创建新的多线样式");
        acutPrintf("\n注册DelMlStyle命令,用于删除新的多线样式");
}
```

6.4.4 效果

（1）编译链接程序，启动AutoCAD 2002，加载生成的ARX文件，能在AutoCAD命令窗口中得到如图6.5所示的结果。

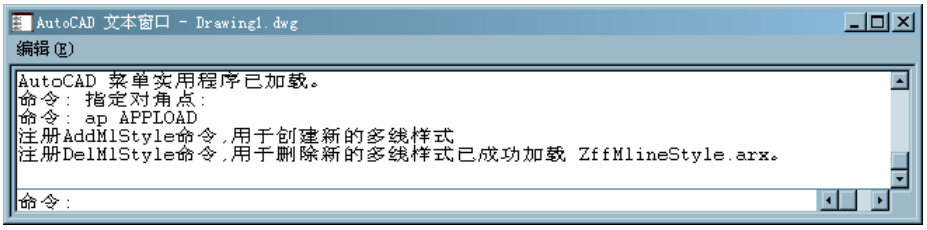

图6.5　显示手工添加的命令提示

（2）执行 AddMlStyle 命令，向当前图形中添加一个名称为“NewStyle”的多线样式。选择【绘图 / 多线】菜单项，按照命令提示进行操作：

```
命令: _mline
当前设置: 对正 = 上，比例 = 20.00，样式 = STANDARD
指定起点或 [对正(J)/比例(S)/样式(ST)]:  st【输入ST并按下Enter键】

输入多线样式名或 [?]:  ?                【查询当前图形中包含的多线样式】

已加载的多线样式:

        名称              说明
---------- ------------------
```

NEWSTYLE

STANDARD

输入多线样式名或 [?]: newstyle 【设置当前要使用的多线样式】

当前设置: 对正 = 上，比例 = 20.00，样式 = NEWSTYLE
指定起点或 [对正(J)/比例(S)/样式(ST)]: 【指定起点】
指定下一点:
指定下一点或 [放弃(U)]:
指定下一点或 [闭合(C)/放弃(U)]: c 【闭合创建的多线】

完成操作后，得到如图6.6所示的结果。

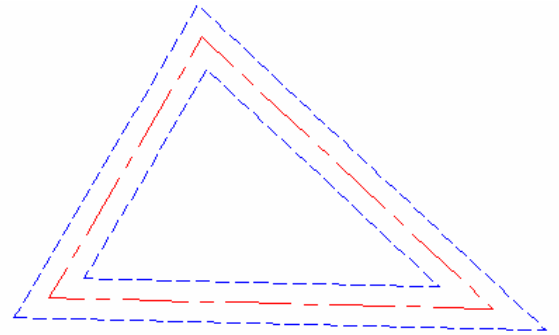

图6.6 使用新样式的多线

（3）执行 DelMlStyle 命令，可以删除前面创建的“NewStyle”多线样式。

6.4.5 小结

学习本节内容之后，读者应该掌握下面的几个知识点：

- 从文件中加载线型。
- 创建和删除多线样式。
- 在加载程序时提供命令的帮助。

第7章 操作图形数据库

AcDbDatabase 类代表 AutoCAD 图形文件，每个 AcDbDatabase 对象都包含许多系统变量、符号表、符号表记录、实体和组成图形的其他对象。

AcDbDatabase 类提供了如下的几类成员函数：

- 访问所有符号表。
- 读写 DWG 文件。
- 获得和设置数据库的特性。
- 执行多种数据库层的操作，例如写块和深层克隆。
- 获得和设置系统变量。

一定要将图形数据库（AcDbDatabase）和文档（AcApDocument）区分开，前者代表了 AutoCAD 图形文件，后者仅仅是为了实现 MDI（多文档用户界面）而提供的一个接口而已。

每个打开的图形都有一个关联的 AcApDocument 对象，AcApDocument 对象包含了一些信息，例如文件名称、MFC 文档对象、当前数据库和当前图形的保存格式等。除了这些方面之外，在其他的任何情况下，请忘记 AcApDocument。

本章要介绍的是与 AcDbDatabase 有关的操作，例如创建图形数据库和访问图形数据库内容，通过写块、插入数据库实现两个图形数据库的内容传递，长事务处理实现在位编辑，以及读写图形的摘要信息等。

7.1 创建和访问图形数据库

7.1.1 说明

本节的实例创建一个新的图形数据库，添加两个圆并将其保存在 AutoCAD 的安装目录下（以帮助系统中的 createDwg 函数为基础改写而成）；局部加载该图形文件，显示加载的所有实体的类名称（例如 AcDbLine、AcDbCircle 等）。

需要注意的是，本节所介绍的实例均不能在图形窗口中显示创建或打开的图形数据库，如果要新建或打开一个图形文件并在图形窗口中显示，应该查看第 11 章（多文档界面）中的相关内容。

7.1.2 思路

1. 图形数据库的基本操作

可以使用下面的语句新建一个图形数据库：

```
AcDbDatabase *pDb = new AcDbDatabase();
```

AcDbDatabase 类的构造函数为：

```
AcDbDatabase(bool buildDefaultDrawing = true, bool noDocument = false);
```

其中，buildDefaultDrawing 指出是否创建一个空的图形数据库，也就是是否包含图形数据库的初始内容（默认的符号表、命名对象字典和一组系统变量）；noDocument 指出新建的图形数据库是否与当前文档相关联。

saveAs 函数用于保存图形数据库，在指定文件名称时必须包含 dwg 扩展名。

readDwgFile 函数将一个已经存在的图形文件的内容读入到当前图形数据库，但是调用该函数的图形数据库必须用下面的语句来创建：

```
AcDbDatabase *pDb = new AcDbDatabase(Adesk::kFalse);
```

2. 获得AutoCAD的安装路径

获得 AutoCAD 的安装路径可以使用三种方法：

- 读取注册表HKEY_LOCAL_MACHINE\ SOFTWARE\ Autodesk\ AutoCAD\ R15.0\ ACAD-1:804 项中AcadLocation键的键值，如图 7.1所示。其中，对应于不同的AutoCAD版本，R15.0\ ACAD-1:804 的内容可能有所不同。

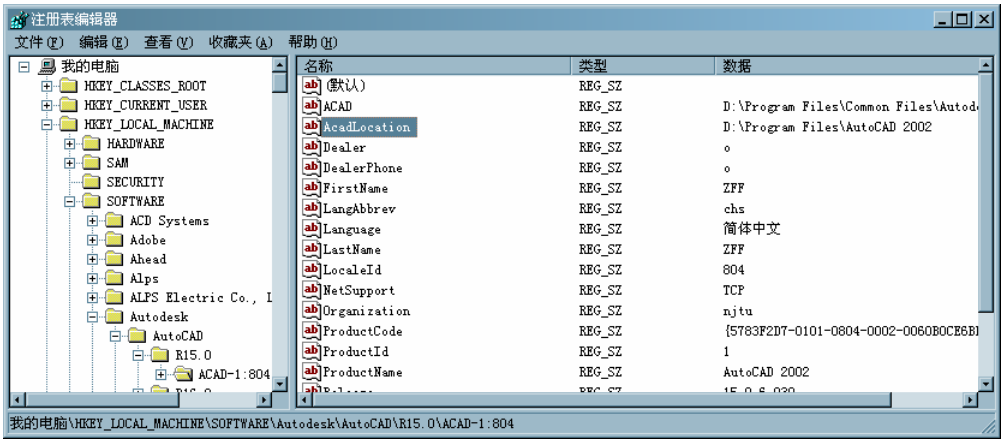

图7.1 注册表中保存的 AutoCAD 安装路径

- 使用 COM 技术调用 ActiveX 模型中相应的属性，如果在 VBA 中描述应该是下面的形式：

```
Sub GetAcadPath()
    MsgBox Application.FullName
End Sub
```

- 使用 Windows API 函数 GetModuleFileName，例如下面的代码可以在 CString 类型的变量 acadPath 中保存 AutoCAD 的安装路径（实际上准确说应该是 acad.exe 文件的路径）：

```
DWORD dwRet
= ::GetModuleFileName(acedGetAcadWinApp()->m_hInstance,
        acadPath.GetBuffer(_MAX_PATH) , _MAX_PATH);
acadPath.ReleaseBuffer();
```

其中，_MAX_PATH 是 Windows 标准库中定义的一个常量，保存 Windows 的最大路径长度。

提示：选择 Windows 开始菜单中的【运行】菜单项，在弹出的【运行】窗口中输入 regedit 然后单击【确定】按钮即可打开注册表编辑器。

3. 局部加载

要实现图形的局部加载，应该在调用 readDwgFile 函数之后执行 AcDbDatabase 类的 applyPartialOpenFilters 函数，最后还要执行 closeInput 函数。

applyPartialOpenFilters 函数的定义为：

```
Acad::ErrorStatus applyPartialOpenFilters(
        const AcDbSpatialFilter* pSpatialFilter,
        const AcDbLayerFilter* pLayerFilter);
```

其中，pSpatialFilter 指定了模型空间中的一个三维区域进行空间过滤，pLayerFilter 则指定了所要进行过滤的图层。

7.1.3 步骤

（1）启动VC++ 6.0，使用ObjectARX向导创建一个新工程，其名称为CreateDatabase。在创建工程的过程中，注意要选择【Use MFC】（使用MFC）选项，如图 7.2所示。

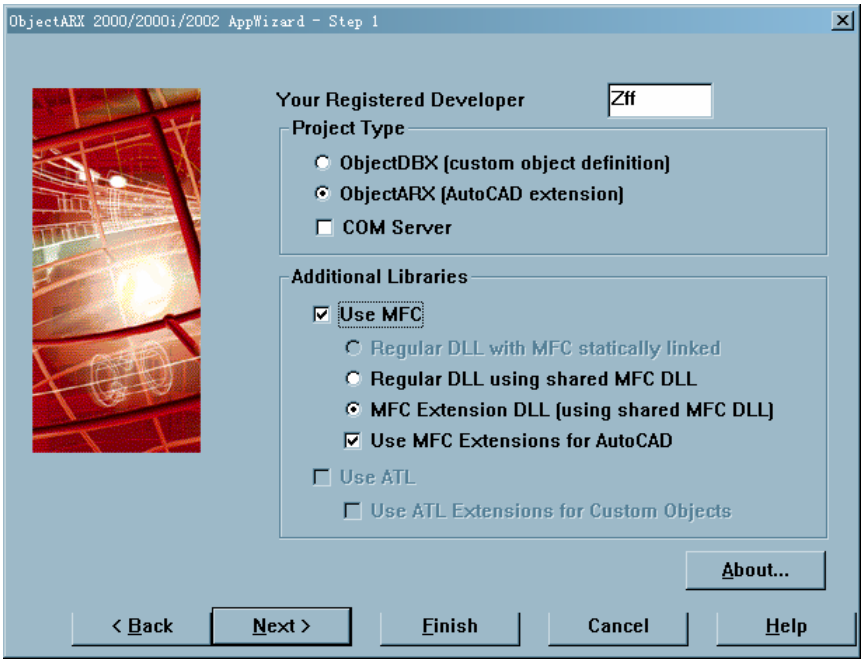

图7.2 在向导中选择“使用 MFC”的选项

（2）注册一个新命令 CreateDwg，用于创建一个新的图形文件，并保存在 AutoCAD 的安装路径中，其实现函数为：

```
void ZffCHAP7CreateDwg()
{
    // 创建新的图形数据库，分配内存空间
    AcDbDatabase *pDb = new AcDbDatabase();

    AcDbBlockTable *pBlkTbl;
    pDb->getSymbolTable(pBlkTbl, AcDb::kForRead);

    AcDbBlockTableRecord *pBlkTblRcd;
    pBlkTbl->getAt(ACDB_MODEL_SPACE, pBlkTblRcd,
        AcDb::kForWrite);
    pBlkTbl->close();

    // 创建两个圆
    AcDbCircle *pCir1 = new AcDbCircle(AcGePoint3d(1,1,1),
                        AcGeVector3d(0,0,1), 1.0);
    AcDbCircle *pCir2 = new AcDbCircle(AcGePoint3d(4,4,4),
                        AcGeVector3d(0,0,1), 2.0);
    pBlkTblRcd->appendAcDbEntity(pCir1);
    pCir1->close();
```

```
        pBlkTblRcd->appendAcDbEntity(pCir2);
        pCir2->close();
        pBlkTblRcd->close();

        CString acadPath;
        GetAcadPath(acadPath);      // 获得acad.exe的位置
        // 去掉路径最后的"acad.exe"字符串，得到AutoCAD安装路径
        acadPath = acadPath.Left(acadPath.GetLength() - 8);
        CString filePath = acadPath + "test.dwg";
        // 使用saveAs成员函数时，必须指定包含dwg扩展名的文件名称
        pDb->saveAs(filePath.GetBuffer(0));
        filePath.ReleaseBuffer();
        delete pDb;             // pDb不是数据库的常驻对象，必须手工销毁
    }
```

上面的代码中使用了 MFC 的 CString 类，其 GetLength 函数用于获得其中包含的字符串的长度。使用 CString 类连接字符串特别方便，可以直接使用+运算符与字符串连接，返回一个新的 CString 类对象。如果你愿意，当然可以使用“acadPath = acadPath + "test.dwg";”，这里为了加强变量的可读性使用了 filePath 来作为文件的名称。

AcDbDatabase 类的 saveAs 函数接受一个 const char*类型的变量（字符串常量），可以直接使用 CString 类的 GetBuffer(0)来获得一个指向保存在 CString 对象中的字符指针，以此作为 saveAs 函数的参数。每次执行完毕 GetBuffer 函数之后，应尽快使用 ReleaseBuffer 函数来释放分配的缓冲区。

代码中的 GetAcadPath 是一个自定义函数，用于获得当前运行的 AutoCAD 程序的 acad.exe 的位置，其实现代码为：

```
    bool GetAcadPath(CString &acadPath)
    {
            DWORD dwRet
= ::GetModuleFileName(acedGetAcadWinApp()->m_hInstance,
                acadPath.GetBuffer(_MAX_PATH) , _MAX_PATH);
            acadPath.ReleaseBuffer();

            if (dwRet == 0)
            {
                return false;
            }
            else
            {
                return true;
```

```
        }
    }
```

其中，DWORD 是 Win32 编程中的一个基本数据类型，它实际上是一个 32 位的无符号整型变量。

（3）注册一个新的命令 ReadDwg，读取 CreateDwg 命令中创建的 test.dwg 文件，在命令窗口中显示图形数据库的模型空间块表记录中所有实体的实体名，其实现函数为：

```
void ZffCHAP7ReadDwg()
{
    // 使用kFalse作为构造函数的参数，创建一个空的图形数据库
    // 这样保证图形数据库仅仅包含读入的内容
    AcDbDatabase *pDb = new AcDbDatabase(Adesk::kFalse);

    // AcDbDatabase::readDwgFile()函数可以自动添加dwg扩展名
    CString acadPath;
    GetAcadPath(acadPath);
    // 去掉路径最后的"acad.exe"字符串
    acadPath = acadPath.Left(acadPath.GetLength() - 8);
    CString filePath = acadPath + "test.dwg";
    pDb->readDwgFile(filePath.GetBuffer(0));
    filePath.ReleaseBuffer();

    // 执行局部加载
    bool bRet = PartialOpenDatabase(pDb);

    // 打开模型空间块表记录
    AcDbBlockTable *pBlkTbl;
    pDb->getSymbolTable(pBlkTbl, AcDb::kForRead);

    AcDbBlockTableRecord *pBlkTblRcd;
    pBlkTbl->getAt(ACDB_MODEL_SPACE, pBlkTblRcd,
        AcDb::kForRead);
    pBlkTbl->close();

    AcDbBlockTableRecordIterator *pBlkTblRcdItr;
    pBlkTblRcd->newIterator(pBlkTblRcdItr);

    // 遍历模型空间块表记录，打印所有的图形
    AcDbEntity *pEnt;
    for (pBlkTblRcdItr->start(); !pBlkTblRcdItr->done();
```

```
        pBlkTblRcdItr->step())
        {
            pBlkTblRcdItr->getEntity(pEnt,
                AcDb::kForRead);
            acutPrintf("类名称: %s\n",
                (pEnt->isA())->name());
            pEnt->close();   // 注意关闭实体
        }
        pBlkTblRcd->close();

        delete pBlkTblRcdItr;
        delete pDb;
    }
```

除了使用自定义函数 PartialOpenDatabase 来执行局部加载之外，其他的代码前面都已经出现过，因此把注意力放在 PartialOpenDatabase 函数上：

```
    bool PartialOpenDatabase(AcDbDatabase *pDb)
    {
        if (pDb == NULL)
                    return false;

        // 指定限制窗口的两个角点
        ads_point pt1, pt2;
        pt1[X] = 0.0;
        pt1[Y] = 0.0;
        pt1[Z] = 0.0;
        pt2[X] = 100.0;
        pt2[Y] = 100.0;
        pt2[Z] = 0.0;

        // 获得当前视图的方向
        AcGeVector3d normal;      // 视图法线方向
        struct resbuf rb;
        acedGetVar("VIEWDIR", &rb);
        normal[0] = rb.resval.rpoint[0];
        normal[1] = rb.resval.rpoint[1];
        normal[2] = rb.resval.rpoint[2];
        normal.normalize();

        // 将窗口角点从WCS转换到ECS
```

```
struct resbuf rbFrom, rbTo;
rbFrom.restype = RTSHORT;
rbFrom.resval.rint = 0;        // WCS
rbTo.restype = RT3DPOINT;
rbTo.resval.rpoint[0] = normal[0];
rbTo.resval.rpoint[1] = normal[1];
rbTo.resval.rpoint[2] = normal[2];
acedTrans(pt1, &rbFrom, &rbTo, FALSE, pt1);
acedTrans(pt2, &rbFrom, &rbTo, FALSE, pt2);

// 创建空间过滤器
AcGePoint2dArray array;
array.append(AcGePoint2d(pt1[0], pt1[1]));
array.append(AcGePoint2d(pt2[0], pt2[1]));
AcDbSpatialFilter spatialFilter;
spatialFilter.setDefinition(
      array,
      normal,
      pt1[Z],
      ACDB_INFINITE_XCLIP_DEPTH,
      -ACDB_INFINITE_XCLIP_DEPTH,
      Adesk::kTrue);

// 创建图层过滤器
AcDbLayerFilter layerFilter;
layerFilter.add("Circle");
layerFilter.add("Line");

// 对图形数据库应用局部加载
Acad::ErrorStatus es;
es = pDb->applyPartialOpenFilters(&spatialFilter, &layerFilter);

if ((es == Acad::eOk) && pDb->isPartiallyOpened())
{
                  pDb->closeInput();
                  return true;
}
else
{
```

```
                return false;
            }
        }
```

首先创建两个坐标分别为（0，0，0）和（100，100，0）的点作为限制窗口的角点，VIEWDIR 系统变量保存了当前视口的观察方向，用一个 AcGeVector3d 对象来保存视口的观察方向。

定义空间过滤器必须使用 ECS 来指定限制窗口的角点坐标，因此必须使用 acedTrans 全局函数将 pt1 和 pt2 从 WCS 转换到 ECS，并使用一个 AcGePoint2dArray 对象来保存这两个角点的坐标。

定义空间过滤器需要使用 AcDbSpatialFilter 类的 setDefinition 函数，该函数被定义为：

```
Acad::ErrorStatus setDefinition(
                const AcGePoint2dArray& pts,
                const AcGeVector3d& normal,
                double elevation,
                double frontClip,
                double backClip,
                Adesk::Boolean enabled);
```

其中，pts 指定了平面上的边界，如果是两个角点就用一个矩形来作为边界，如果是多个角点就用多边形来作为边界；normal 指定正的拉伸方向；elevation 指定标高值，和 normal 一起定义 pts 所在的平面；frontClip 指定正拉伸方向上的前向剪切距离；backClip 指定正向拉伸方向上的后向剪切距离；enabled 指定剪切的体积有效，或者使用整个三维空间。

让我们看看 PartialOpenDatabase 函数中如何使用 setDefinition 函数来创建空间过滤器：

```
spatialFilter.setDefinition(
            array,
            normal,
            pt1[Z],
            ACDB_INFINITE_XCLIP_DEPTH,
            -ACDB_INFINITE_XCLIP_DEPTH,
            Adesk::kTrue);
```

array、normal 和 pt[Z]共同定义了限制窗口的大小和在三维空间中的具体位置，而 ACDB_INFINITE_XCLIP_DEPTH 则指定了前向和后向剪切距离，该常量在 ObjectARX 中被定义为 1.0e+300，因此构建的空间过滤器是一个以 pt1、pt2 构成的矩形为截面的无限高长方体。

相比之下，构建图层过滤器就很简单了，只需要使用 AcDbLayerFilter 函数将需要加载的图层名称添加到过滤器中即可。

7.1.4 效果

（1）编译链接程序，启动 AutoCAD 2002，加载生成的 ARX 文件。执行 CreateDwg 命令，然后选择【文件 / 打开】菜单项，打开生成的 test.dwg 文件，执行 ZOOM 命令并选择 E 选项，观察图形中已经创建的两个圆。

（2）选择【格式 / 图层】菜单项，打开图层特性管理器，创建两个新图层Circle和Line，并将其颜色修改为红色和蓝色，如图 7.3所示。

图7.3 创建新图层

（3）删除图形中的两个圆，使用Rectangle命令创建一个角点为（0，0）和（100，100）的矩形，然后分别使用Line和Circle命令创建若干个圆和直线，确保有一条直线位于矩形的外部，并分别将两条直线放置在Line层上，将一个圆放置在Circle层上，如图 7.4所示。保存并关闭test.dwg图形。

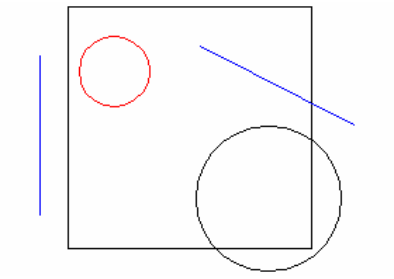

图7.4 创建测试实体

（4）执行ReadDwg命令，能够在命令窗口得到如图 7.5所示的结果。

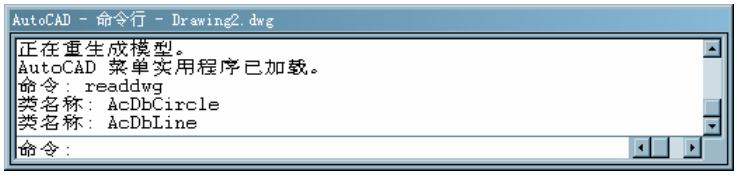

图7.5　显示图形数据库中所有实体的类名称

这个结果与预期是一致的，因为局部加载包含了两个过滤条件：（1）在矩形内部；（2）在图层 Circle 或 Line 上。满足条件的实体仅有两个，因此被加载到图形数据库中的实体仅有两个。

7.1.5 小结

学习本节的内容之后，读者需要掌握下面的知识点：

- 获得 AutoCAD 安装路径。
- 创建新的图形数据库和读取已经存在的图形文件。
- 局部加载的实现。
- CString 类型变量的使用。

7.2 在图形数据库之间传递数据

7.2.1 说明

Windows 程序的一个重要特征是对多文档界面的支持，而多文档应用程序允许在不同的文档之间交换数据。在 AutoCAD 中，确实可以通过复制和粘贴在不同的图形之间交换数据。

从 ObjectARX 编程的角度来看，这个问题就没有这么直观了，不同的图形数据库之间是独立的，如何在两者之间传递数据呢？本节的实例使用 AcDbDatabase 类的 wblock 和 insert 成员函数，并且通过一个临时图形数据库作为中介，实现将一个图形数据库的块定义复制到另外一个图形数据库中。

7.2.2 思路

1. wblock函数

AcDbDatabase 类的 wblock 函数用于将当前图形数据库的对象导出到一个新的图形数据库中，它包含了三个不同的定义形式：

```
Acad::ErrorStatus wblock(
                AcDbDatabase*& pOutputDb,
                const AcDbObjectIdArray& outObjIds,
                const AcGePoint3d& basePoint);
Acad::ErrorStatus wblock(AcDbDatabase*& pOutputDb);
Acad::ErrorStatus wblock(AcDbDatabase*& pOutputDb, AcDbObjectId blockId);
```

这三种定义形式与 AutoCAD 中 WBLOCK 命令的三种功能一一对应。第一种形式能够将

保存在 outObjIds 中所有的实体（必须是 AcDbEntity 的派生类的对象）导出到一个新的图形数据库；第二种形式将当前图形中所有的内容导出到新的图形数据库中；第三种形式能将当前图形中指定的一个块表记录的内容导出到一个新的图形数据库。

需要注意的是，wblock 会创建一个新的图形数据库并分配内存，在使用结束后，编程者必须手工销毁新的图形数据库。

2. insert函数

insert 函数用于将一个外部图形数据库的内容导入到当前的图形数据库，它同样包含三种定义形式：

```
Acad::ErrorStatus insert(
                AcDbObjectId& blockId,
                const char* pBlockName,
                AcDbDatabase* pDb,
                bool preserveSourceDatabase = true);
Acad::ErrorStatus insert(
                const AcGeMatrix3d& xform,
                AcDbDatabase* pDb,
                bool preserveSourceDatabase = true);
Acad::ErrorStatus insert(
                AcDbObjectId& blockId,
                const char* pSourceBlockName,
                const char* pDestinationBlockName,
                AcDbDatabase* pDb,
                bool preserveSourceDatabase = true);
```

第一种形式将外部图形数据库的内容作为一个块表记录保存在当前的图形数据库中；第二种形式将外部图形数据库的内容插入到当前图形数据库的模型空间；第三种形式将外部图形数据库的内容作为一个新的块表记录 pDestinationBlockName 添加到当前图形数据库中，并且将当前图形数据库中原有的块表记录 pSourceBlockName 的所有内容添加到新的块表记录 pDestinationBlockName 中。

3. 将一个图形数据库的块定义传递到另一个图形数据库中

利用 wblock 函数可以将源数据库中的内容导出到一个临时数据库，然后使用 insert 函数将临时数据库的内容导入到目标数据库中，即可实现两个图形数据库的数据传递。

如果要在两个图形数据库之间传递多个实体非常简单，可以使用 wblock 函数的第一种形式将多个实体导出到临时数据库，再使用 insert 函数的第二种形式导入临时数据库即可。

如果要将一个图形数据库的块定义传递到另一个图形数据库，就必须使用 wblock 函数的第一种形式导出一个对应的块参照实体到临时图形数据库，这样临时图形数据库中包含了块参照和其对应的块定义。再将临时图形数据库其导入到目标图形数据库中，目标数据库中亦包含了块参照和其对应的块定义，删除块参照即可完成块定义的传递。

7.2.3 步骤

启动 VC++ 6.0，使用 ObjectARX 向导创建一个新工程，其名称为 Wblock。注册一个命令 ImportBlkDef，用于将一个外部图形文件中包含的块定义复制到当前图形中来，其实现函数为：

```
void ZffCHAP7ImportBlkDef()
{
        // 提示用户选择图形文件
        AcDbDatabase pExternalDb(Adesk::kFalse);    // 外部图形数据库
    struct resbuf *rb;
        rb = acutNewRb(RTSTR);
    if (RTNORM != acedGetFileD("选择图形文件名称", NULL, "dwg", 0,
            rb))
    {
            acutRelRb(rb);          // 意外退出时要释放结果缓冲区
        return;
    }
    if (Acad::eOk != pExternalDb.readDwgFile(rb->resval.rstring))
    {
        acedAlert("读取DWG文件失败!");
            acutRelRb(rb);          // 意外退出时要释放结果缓冲区
        return;
    }
        acutRelRb(rb);

        // 获得名称为Blk的块表记录
    AcDbBlockTable* pBlkTbl;
    if (Acad::eOk != pExternalDb.getBlockTable(pBlkTbl, AcDb::kForRead))
    {
        acedAlert( "获得块表失败!" );
        return;
    }
    AcDbBlockTableRecord* pBlkTblRcd;
    Acad::ErrorStatus es = pBlkTbl->getAt(_T("Blk"), pBlkTblRcd, AcDb::kForRead);
    pBlkTbl->close();
    if (Acad::eOk != es) {
        acedAlert("获得指定的块表记录失败!");
        return;
    }
```

```
        // 创建块参照遍历器
        AcDbBlockReferenceIdIterator *pItr;
        if (Acad::eOk != pBlkTblRcd->newBlockReferenceIdIterator(pItr))
        {
            acedAlert("创建遍历器失败!");
    pBlkTblRcd->close();
    return;
        }

        // 找到图形中的第一个复合要求的块参照，将其添加到ObjectId数组中
        AcDbObjectIdArray list;      // 导出到临时图形数据库的实体数组
        for (pItr->start(); !pItr->done(); pItr->step())
        {
            AcDbObjectId blkRefId;
            if (Acad::eOk == pItr->getBlockReferenceId(blkRefId))
            {
                list.append(blkRefId);
                break;
            }
        }
        delete pItr;
        pBlkTblRcd->close();

if (list.isEmpty()) {
    acedAlert("实体数组中未包含任何实体!");
    return;
}

AcDbDatabase *pTempDb;          // 临时图形数据库
        // 将list数组中包含的实体输出到一个临时图形数据库中
if (Acad::eOk != pExternalDb.wblock( pTempDb, list, AcGePoint3d::kOrigin ))
{
    acedAlert("wblock操作失败!");
    return;
}
        // 将临时数据库的内容插入到当前图形数据库
if (Acad::eOk != acdbHostApplicationServices()->workingDatabase()
    ->insert(AcGeMatrix3d::kIdentity, pTempDb))
    acedAlert("insert操作失败!");
```

```
        delete pTempDb;

                // 如果不需要保留块参照，将模型空间中的最后一个对象删除即可
                ads_name lastEnt;
                if (acdbEntLast(lastEnt) != RTNORM)
                {
                        acedAlert("获得模型空间最后一个实体失败!");
                        return;
                }
                AcDbObjectId entId;
                es = acdbGetObjectId(entId, lastEnt);
                AcDbEntity *pEnt;
                es = acdbOpenAcDbEntity(pEnt, entId, AcDb::kForWrite);
                pEnt->erase();
                pEnt->close();
}
```

上面的代码中，需要注意的是，list 数组中仅包含了一个“Blk”块表记录对应的块参照。此外，用到了两个特殊的变量 AcGePoint3d::kOrigin 和 AcGeMatrix3d::kIdentity，前者返回一个三个坐标值均为 0 的点，后者返回一个 4 阶（不是 3 阶，具体请参照 AcGeMatrix3d 类的定义）单位矩阵。

7.2.4 效果

（1）编译链接程序，启动 AutoCAD 2002，加载生成的 ARX 文件。创建一个新图形，在其中定义一个块 Blk，并在该图形中插入一个块 Blk 的参照，将该图形保存为 test.dwg，关闭该图形。

（2）新建一个图形，执行ImportBlkDef命令，系统会弹出如图 7.6所示的【选择图形文件名称】对话框。选择上一步保存的test.dwg文件，单击【打开】按钮。

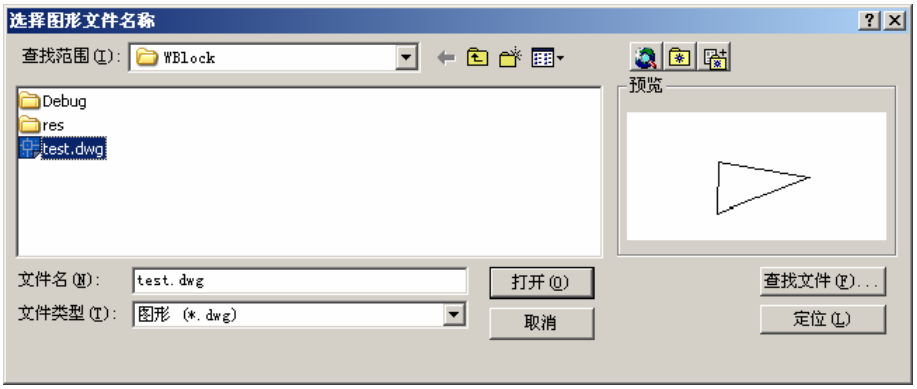

图7.6　选择图形文件

（3）执行Insert命令，系统会弹出如图 7.7所示的【插入】对话框，在【名称】列表中已经包含了Blk，表示当前图形中有一个名称为Blk的块定义。

图7.7　查看当前图形中包含的块定义

7.2.5 小结

学习本节内容之后，读者应该掌握下面的知识点：

- 使用 wblock 函数导出图形数据库的内容。
- 使用 insert 函数导入外部图形数据库的内容。
- 两个全局函数 acdbEntLast 和 acdbGetObjectId 的使用。
- AcGePoint3d::kOrigin 和 AcGeMatrix3d::kIdentity 的含义和使用。

7.3 长事务处理

7.3.1 说明

长事务用于实现 AutoCAD 的引用编辑功能，能够为应用程序检出要编辑的实体，编辑之后再将其保存回原来的位置。使用长事务处理，系统会使用编辑后的对象替换原来的对象。

长事务处理一般用于三种情况：

- 引用编辑同一个图形内的普通块定义。
- 引用编辑图形中的外部引用。
- 引用编辑与当前图形无关的、临时图形数据库。

所谓长事务处理，实际上是允许用户在当前图形数据库的模型空间中以可视（要编辑的实体被复制到当前图形数据库并显示在图形窗口中）的形式进行编辑，编辑完成之后，能将编辑的结果保存回原来的位置。

ObjectARX 开发包 docsamps\longtrans 文件夹中的实例演示了引用编辑临时图形数据库，

本节的实例则演示引用编辑当前图形数据库中的一个块定义。

7.3.2 思路

执行长事务处理，大致需要下面的步骤：

（1）创建一个 AcDbObjectIdArray 对象，将所要引用编辑的实体的 ObjectId 添加到该数组中；

（2）使用 checkOut 函数将选中的实体复制到当前图形数据库的模型空间；

（3）获得一个指向长事务对象的指针，为其创建一个工作集遍历器，遍历工作集中的所有实体并对其进行编辑；

（4）使用 checkIn 函数将引用编辑的实体保存回原来的位置。

7.3.3 步骤

启动 VC++ 6.0，使用 ObjectARX 向导创建一个新工程，其名称为 LongTransaction。注册一个新命令 BlockEdit，用于引用编辑当前图形中的块“Blk”，将其中所有的圆修改为红色，其实现函数为：

```
void ZffCHAP7BlockEdit()
{
        // 获得指定的块表记录
        AcDbBlockTable *pBlkTbl;
        acdbHostApplicationServices()->workingDatabase()
                ->getBlockTable(pBlkTbl, AcDb::kForRead);
        AcDbBlockTableRecord *pBlkTblRcd;// Blk块表记录
        pBlkTbl->getAt("Blk", pBlkTblRcd, AcDb::kForWrite);

        AcDbObjectIdArray objIdArray;          // 要修改的实体的数组
        AcDbBlockTableRecordIterator *pItr;        // 块表记录遍历器
        pBlkTblRcd->newIterator(pItr);

        // 遍历块表记录，将所有的圆添加到objIdArray中
        for (pItr->start(); !pItr->done(); pItr->step())
        {
                AcDbEntity *pEnt;
                pItr->getEntity(pEnt, AcDb::kForRead);

                if (pEnt->isKindOf(AcDbCircle::desc()))
                {
                    objIdArray.append(pEnt->objectId( ));
                }
```

```
            pEnt->close();
        }
        delete pItr;
        pBlkTblRcd->close();

        // 获得当前图形模型空间块表记录
        AcDbBlockTableRecord *pModelSpaceRcd;
        pBlkTbl->getAt(ACDB_MODEL_SPACE, pModelSpaceRcd,
AcDb::kForWrite);
        pBlkTbl->close();

        AcDbObjectId id = pModelSpaceRcd->objectId();    // 模型空间的
ObjectId
        pModelSpaceRcd->close();

        // 创建长事务
        AcDbObjectId transId;        // 长事务对象的ObjectId
        AcDbIdMapping errorMap;      // 源实体和克隆实体之间的ObjectId
映象
        // 将选中的实体取出到当前图形数据库的模型空间
        acapLongTransactionManagerPtr()->checkOut(transId, objIdArray,
            id, errorMap);

        // 将这些实体的颜色修改为红色
        AcDbObject *pObj;                // 指向长事务的指针
        if (acdbOpenObject(pObj, transId, AcDb::kForRead) == Acad::eOk)
        {
            // 获得指向长事务的指针
            AcDbLongTransaction *pLongTrans =
                AcDbLongTransaction::cast(pObj);

            if (pLongTrans != NULL)
            {
                // 创建一个工作集遍历器
                AcDbLongTransWorkSetIterator *pWorkSetIter;
                pLongTrans->newWorkSetIterator(pWorkSetIter);

                // 遍历工作集中的所有实体，修改其颜色
                for (pWorkSetIter->start(); !pWorkSetIter->done();
```

```
                    pWorkSetIter->step())
                {
                    AcDbEntity *pEnt;
                    acdbOpenObject(pEnt, pWorkSetIter->objectId(),
                      AcDb::kForWrite);
                    pEnt->setColorIndex(1);
                    pEnt->close();
                }
                delete pWorkSetIter;
            }
            pObj->close();
        }

        // 暂停观察程序中的变化
        char strInput[100];
        acedGetString(0, "\n按下Enter键将修改的实体保存回原来的位置...",
strInput);

        // 将实体存储回原来的位置
        acapLongTransactionManagerPtr()->checkIn(transId, errorMap);
    }
```

其中，执行长事务处理的代码已经加粗。

7.3.4 效果

（1）编译链接程序，启动 AutoCAD 2002，加载生成的 ARX 文件。创建一个新图形，在其中创建若干个图形对象，其中包括两个圆，然后将创建的实体定义为一个块 Blk。如果图形中包含 Blk 块的参照，就将其从图形中删除。

（2）执行BlockEdit命令，系统会将添加到ObjectIdArray中的实体复制并显示在图形窗口中，如图 7.8所示。

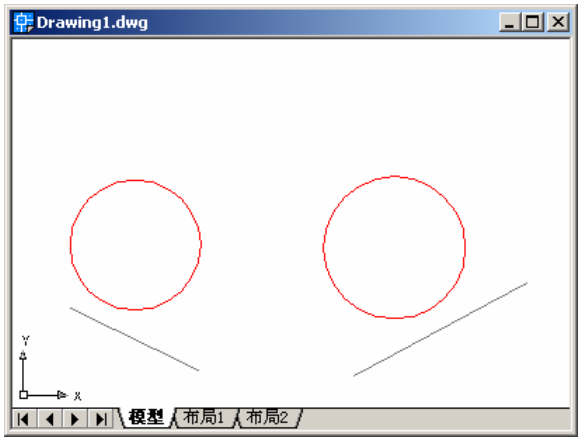

图7.8 图形窗口中显示引用编辑的实体

（3）按下 Enter 键，完成引用编辑，引用编辑的实体从图形窗口中消失，并被保存到原来的位置，完成引用编辑实体的更新。

7.3.5 小结

关于长事务的处理，在 AutoCAD 中提供了块定义和外部引用图形的引用编辑，读者可以根据长事务处理的基本方法，编制复合自己要求的处理程序。

7.4 保存图形摘要信息

7.4.1 说明

AutoCAD 的摘要信息对于图形的检索非常有帮助，除了在 AutoCAD 中选择【文件 / 图形属性】菜单项进行设置之外，还可以在 ObjectARX 中访问这些信息。本节的实例介绍在 ObjectARX 中保存图形摘要信息的方法。

7.4.2 思路

AcDbDatabaseSummaryInfo 类被用来封装图形的摘要信息，其中包括了读取和设置图形摘要信息的方法。该类是一个抽象类，必须通过全局函数 acdbGetSummaryInfo 来获得，修改图形摘要信息之后，要使用全局函数 acdbPutSummaryInfo 将其保存导图形数据库中。

图形摘要信息保存在有名对象字典的“DWGPROPS”字典中，要判断图形是否包含摘要信息，就可以查看有名对象字典的根字典中是否存在关键字为“DWGPROPS”的字典。

7.4.3 步骤

启动 VC++ 6.0，使用 ObjectARX 向导创建一个名为 SummaryInfo 的新工程。注册一个命令 SaveSummaryInfo，用于在当前图形中保存摘要信息，其实现代码为：

```
void ZffCHAP7SaveSummaryInfo()
{
        // 必须确保加载了acsiobj.arx
        if (!acrxDynamicLinker->loadModule("acsiobj.arx", 0))
        {
                acedAlert("未加载必要的SummaryInfo对象!");
                return;
        }

        // 判断当前图形是否已经包含摘要信息
        if (HasSummaryInfo())
                return;

        AcDbDatabase *pDb;
        pDb = acdbHostApplicationServices()->workingDatabase();

        // 获得图形的摘要信息
        AcDbDatabaseSummaryInfo *pInfo;
        Acad::ErrorStatus es;
        es = acdbGetSummaryInfo(pDb, pInfo);

        pInfo->setAuthor("Afanto");
        pInfo->setComments("The drawing is used for the ARX book.");
        pInfo->addCustomSummaryInfo("Size", "A4");
        pInfo->addCustomSummaryInfo("Language", "Chinese");
        pInfo->setHyperlinkBase("http://www.cadhelp.net");
        pInfo->setKeywords("ObjectARX");
        pInfo->setLastSavedBy("cadhelp");
        pInfo->setRevisionNumber("Version 1.0");
        pInfo->setSubject("Development");
        pInfo->setTitle("Development of ObjectARX");

        // 保存摘要信息
        es = acdbPutSummaryInfo(pInfo);
}
```

其中，HasSummaryInfo 是一个自定义函数，用于判断当前图形是否已经保存了摘要信息，

其实现代码为：

```
bool HasSummaryInfo()
{
        AcDbDictionary *pDict;
        acdbHostApplicationServices()->workingDatabase()
                ->getNamedObjectsDictionary(pDict, AcDb::kForRead);

        // 摘要信息保存在有名对象字典的"DWGPROPS"字典中
        if (!pDict->has("DWGPROPS"))
        {
                pDict->close();
                return false;
        }
        pDict->close();
        return true;
}
```

7.4.4 效果

编译链接程序，启动AutoCAD 2002，加载生成的ARX文件。执行SaveSummaryInfo命令，在当前图形中保存摘要信息。选择【文件 / 图形属性】菜单项，系统会弹出图形属性对话框，在其中的【摘要】、【统计信息】和【自定义】三个选项卡中能够显示已经保存的摘要信息，如图 7.9所示。

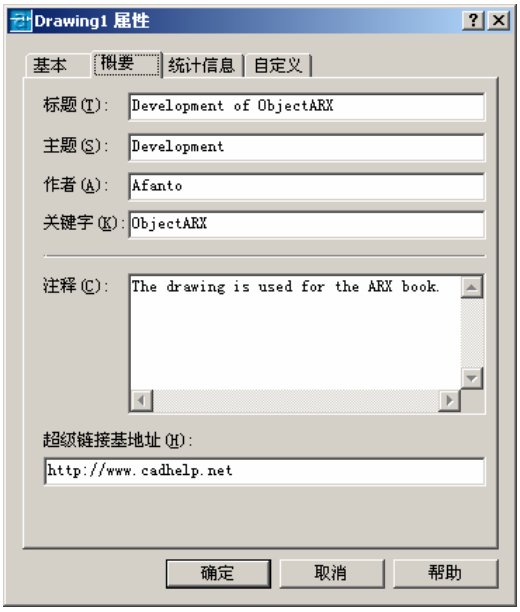

图7.9 显示图形的摘要信息

7.4.5 小结

如果要读取图形中保存的摘要信息，同样可以使用 acdbGetSummaryInfo 函数获得图形的摘要信息，读取 AcDbDatabaseSummaryInfo 中的内容，然后使用 acdbPutSummaryInfo 函数将其保存。

作为一个练习，读者可以自行编写从当前图形中读取摘要信息的程序。

第8章 在 ObjectARX 中使用 MFC

ObjectARX 以动态链接库的形式运行，能够在其中使用 MFC 的资源，实际上这是编程人员选择 ObjectARX 作为开发工具的一个重要原因。ObjectARX 也提供了一组基于 MFC 的类，使用这些类开发出来的界面风格能与 AutoCAD 自身的风格保持一致。

值得一提的是，本章所有的实例都必须使用 ObjectARX 向导创建，并且要选择“使用 MFC”的选项。

8.1 模态对话框

8.1.1 实例说明

在 ObjectARX 中可以直接使用 MFC 的对话框，也可以使用 ObjectARX 中的 AcUi 类库来构建对话框。前者使用简单，与普通的 MFC 用户界面构建完全相同，后者要使用 ObjectARX 提供的类库，可以创建与 AutoCAD 风格完全一致的用户界面。

本节首先使用基于 MFC 的对话框演示在 ObjectARX 应用程序中创建模态对话框的一般过程，然后使用基于 ObjectARX 的对话框演示创建与 AutoCAD 风格相同的用户界面，以及隐藏模态对话框和 AutoCAD 进行交互的方法。

8.1.2 编程思路

在 ObjectARX 应用程序中显示一个非模态对话框非常简单，只需要创建一个对话框，并将其与 CDialog 或其派生类相关联，调用 CDialog 类的 DoModal 函数即可。

在 ObjectARX 中，以 AcUi 开头的类用于创建与 AutoCAD 风格一致的用户界面，这组类从 MFC 中继承，除了具有基类的特性之外，还引入了一些在 AutoCAD 中特有的特性。使用 ObjectARX 嵌入工具栏创建基于 ObjectARX 的对话框，能够节省大量的手工编码工作量，因此本节的程序多用向导完成。

8.1.3 步骤

1. 直接使用MFC

（1）启动VC++ 6.0，使用ObjectARX向导创建一个新工程ModalDialog，注意要选择“使用MFC”的选项。选择【Insert/Resource】菜单项，系统会弹出【Insert Resource】对话框，

如图 8.1所示。选择【Dialog】选项，单击【New】按钮向当前工程添加一个新的对话框。

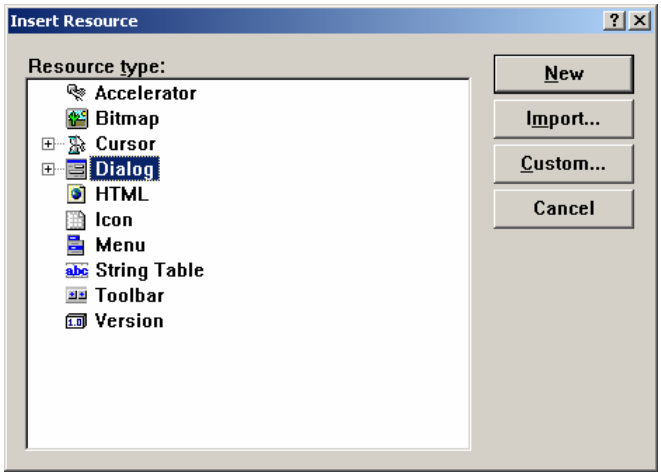

图8.1　添加对话框资源

（2）在对话框的资源编辑窗口中，选择新添加的对话框，直接按下Enter键，系统会弹出如图 8.2所示的对话框。在【ID】组合框中输入新的ID值“IDD_MFC_MODAL”，在【Caption】文本框中输入“MFC模态对话框”，单击【Font】按钮修改对话框的字体为“宋体”，字体大小设置为 9。

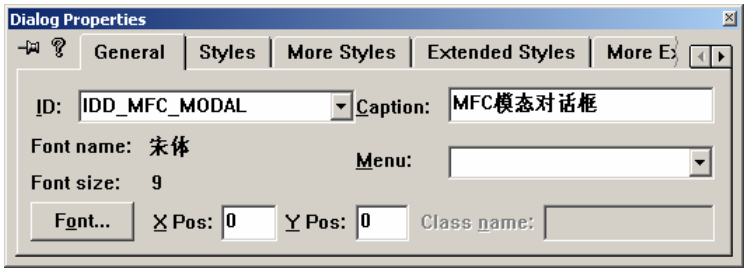

图8.2　修改 ID、字体和标题栏

（3）在【Dialog Properties】对话框中，单击左上角的【Keep Visible】按钮，使该对话框始终显示在最上层。在对话框设计界面中单击【OK】按钮，在【Dialog Properties】对话框中将其Caption属性修改为“确定”。使用相同的方法将【Cancel】按钮的Caption属性修改为“取消”，如图 8.3所示。关闭【Dialog Properties】对话框。

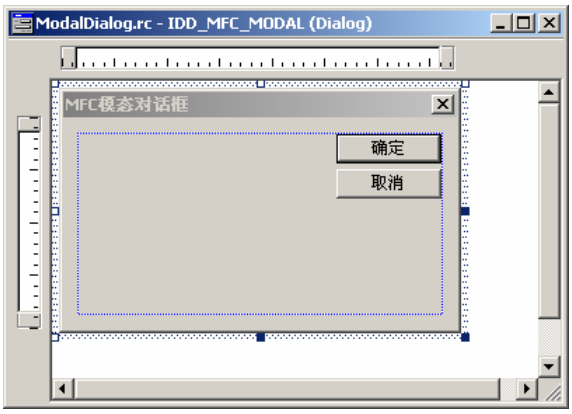

图8.3 修改按钮的标题

（4）选择【View/ClassWizard】菜单项，或者直接按下快捷键Ctrl+W，系统会弹出如图8.4所示的对话框，提示用户为新的对话框资源（IDD_MFC_MODAL）添加新类或者指定对应的类。

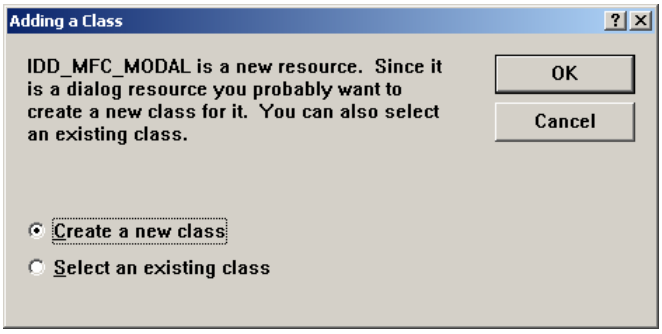

图8.4 提示用户添加新类

（5）单击【OK】按钮，系统会弹出如图 8.5所示的【New Class】对话框。输入CMfcDialog作为新类的名称，单击【OK 】按钮完成新类的创建。系统会自动在工程中添加两个文件MfcDialog.h和MfcDialog.cpp，分别用于保存CMfcDialog类的定义和实现代码。

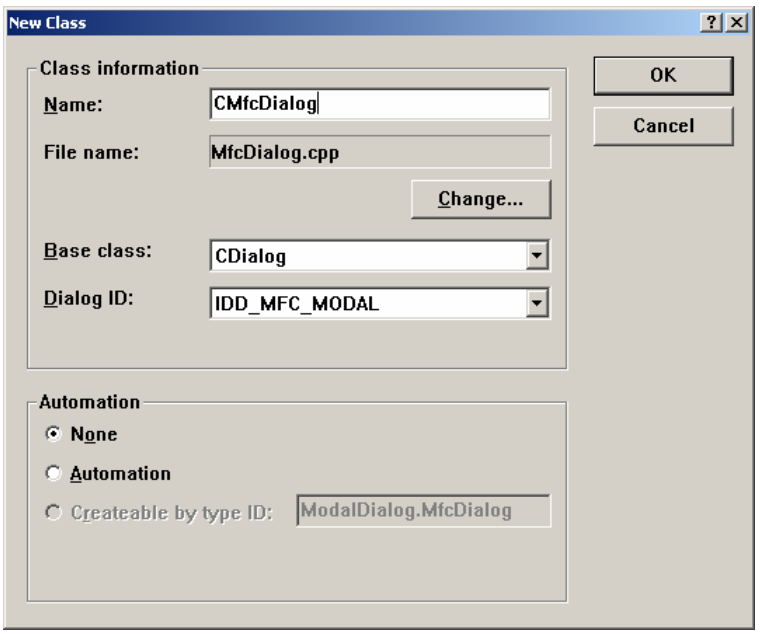

图8.5　指定新类的名称和位置

（6）添加新类之后，在【MFC ClassWizard】对话框的【Message Maps】选项卡中，就出现了CMfcDialog类及对话框中所要含的资源，如图 8.6所示。

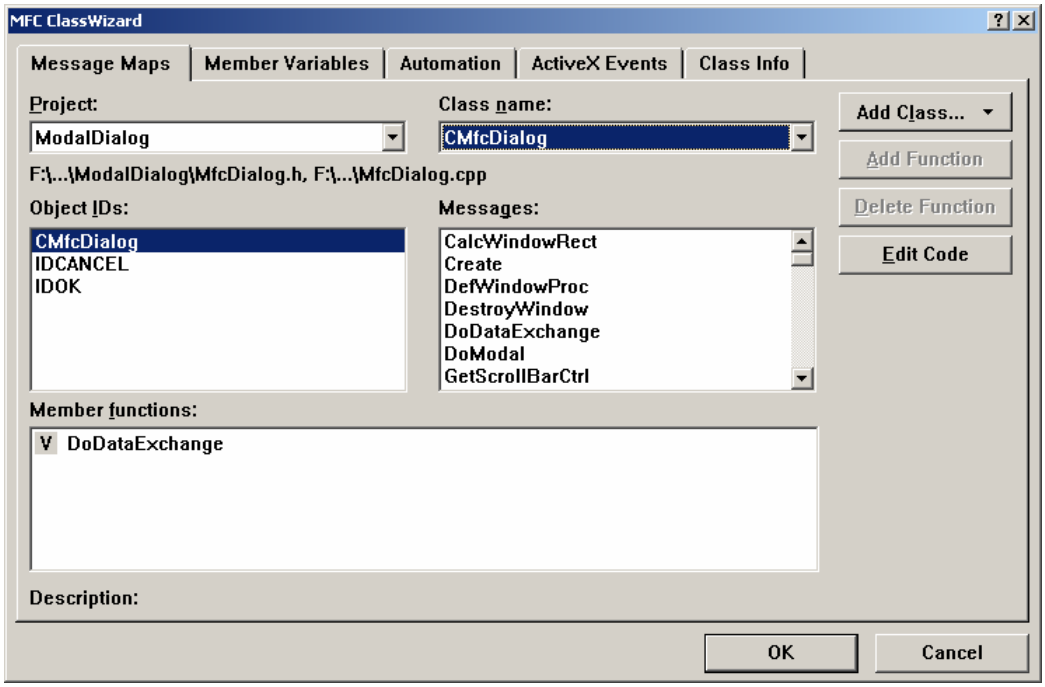

图8.6　显示添加的新类

（7）注册一个新命令 MfcModal，用于显示新创建的对话框，其实现代码为：

```
void ZffCHAP12MfcModal()
```

```
{
        // 显示MFC的模态对话框
        CMfcDialog theDialog;
        if (theDialog.DoModal() == IDOK)
        {
            AfxMessageBox(_T("关闭对话框！"));
        }
}
```

（8）在 ZffCHAP12MfcModal 函数所在的文件 ModalDialogCommands.cpp 中，添加对 CMfcDialog 类的包含：

```
#include "MfcDialog.h"
```

此外，还需要在 CMfcDialog 类的定义文件 MfcDialog.h 中添加下面的语句：

```
#include "Resource.h"
```

这样，程序才能被正确编译。

2. 使用ObjectARX中基于MFC的类库

（1）插入一个新的对话框资源，分别将其 ID 修改为“IDD_ARX_MODAL”，标题修改为“使用 ObjectARX 提供的 MFC 支持”，字体修改为宋体，字体大小为 9。删除对话框中的 OK 和 Cancel 按钮。

（2）在对话框中添加 2 个按钮、4 个标签和 4 个文本框，其在窗体中的位置如图 8.7所示。其中，2 个按钮的ID分别设置为IDC_BUTTON_POINT和IDC_BUTTON_ANGLE，4 个文本框的ID分别设置为IDC_EDIT_XPT、IDC_EDIT_YPT、IDC_EDIT_ZPT和IDC_EDIT_ANGLE。

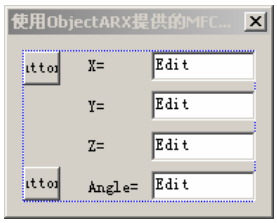

图8.7 设计窗体

分别选择 2 个按钮，在属性对话框的【Styles】选项卡中选中【Owner draw】复选框，如图 8.8所示。

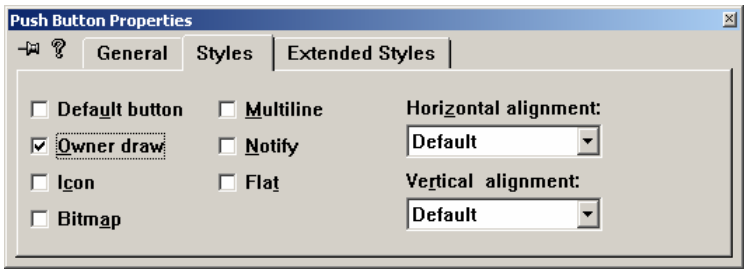

图8.8　设置按钮的 Owner draw 属性

（3）在对话框设计界面中选择新建的对话框，单击ObjectARX嵌入工具栏的“ObjectARX MFC Support”按钮，系统会弹出如图 8.9所示的对话框。在【Name】文本框中输入CArxDialog，选择CAcUiDialog作为新类的基类，单击【Create Class】按钮创建CArxDialog类，然后单击【OK】按钮关闭对话框。

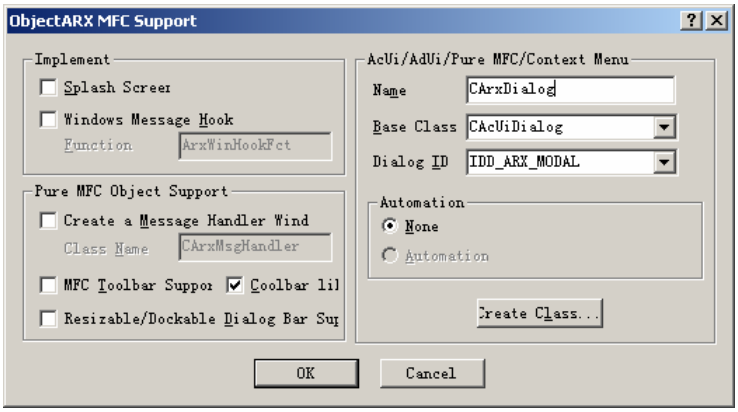

图8.9　使用 ObjectARX 嵌入工具创建对话框的类

（4）在对话框设计界面中选择新建的对话框，按下Ctrl+W快捷键，系统会弹出【MFC ClassWizard】对话框，切换到【Member Variables】选项卡。双击控件列表中的IDC_BUTTON_ANGLE选项，系统会弹出如图 8.10所示的【Add Member Variable】对话框，从【Category】列表框中选择AutoCAD Control选项，从【Variable type】列表框中选择CAcUiPickButton选项，输入m_btnAngle作为成员变量的名称，单击【OK】按钮完成成员变量的添加。

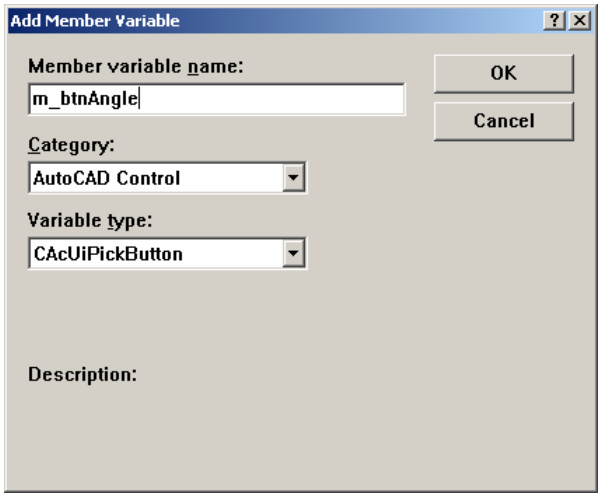

图8.10 为按钮添加成员变量

完成操作后，会在 CArxDialog 类中创建一个成员变量 m_btnAngle，该成员变量与对话框中的角度按钮相关联。

（5）使用相同的方法，分别为对话框中的 2 个按钮和 4 个文本框创建对应的成员变量，如图 8.11所示。

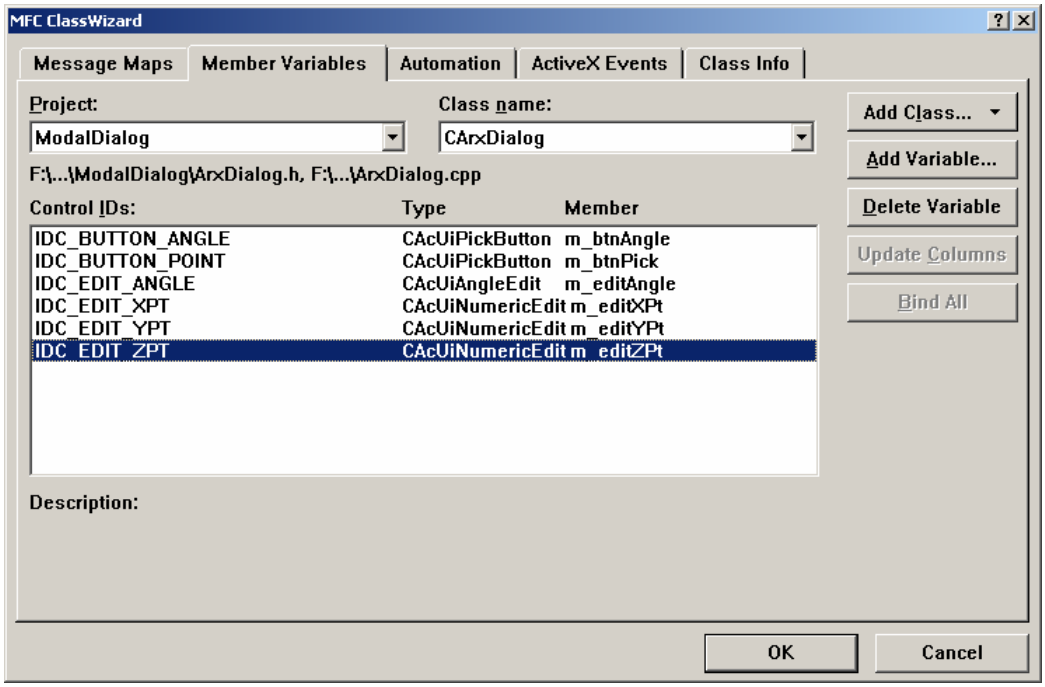

图8.11 为对话框所有控件添加成员变量

（6）切换到【Message Maps】选项卡，在对象ID列表中选择CArxDialog选项，然后双击消息列表中的WM_INITDIALOG选项，为对话框的初始化事件添加处理函数OnInitDialog，关闭对话框的事件添加处理函数OnClose。此外，分别为两个按钮的单击事件添加处理函数

OnButtonAngle 和 OnButtonPoint，为 4 个文本框的失去焦点事件分别添加处理函数 OnKillfoucsEditAngle、OnKillfoucsEditXPt、OnKillfoucsEditYPt和OnKillfoucsEditZPt，如图 8.12所示。单击【OK】按钮关闭【MFC ClassWizard】对话框。

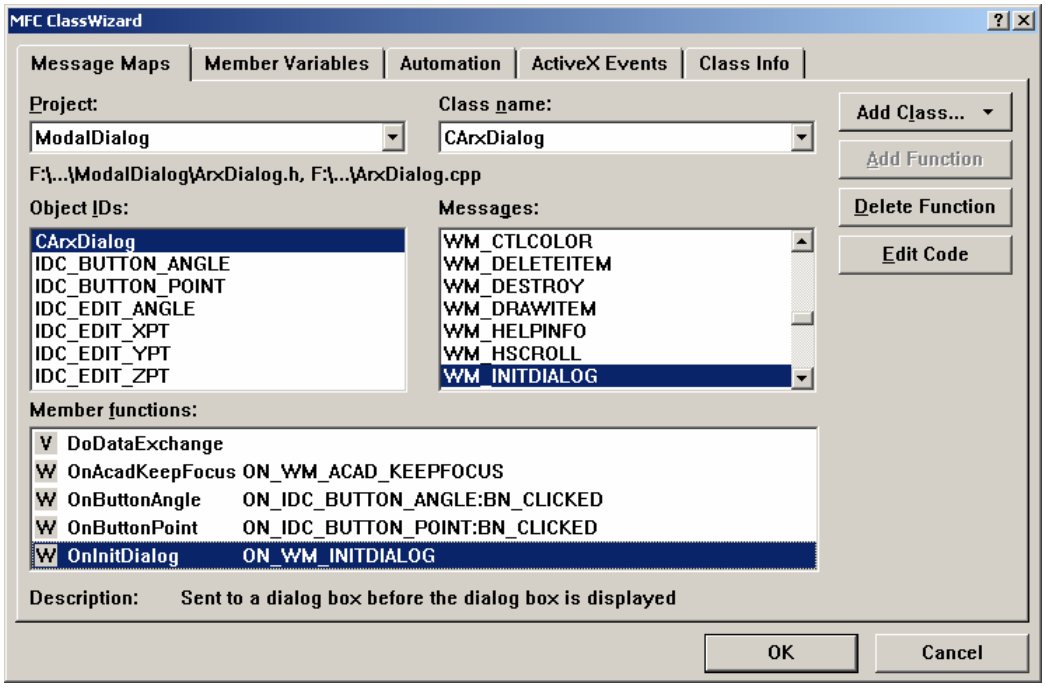

图8.12　为对话框添加消息处理函数

（7）在 CArxDialog 类中添加 4 个公有成员变量，用于作为用户在图形窗口中输入的点坐标、角度与文本框成员变量之间的“沟通工具”：

```
CString m_strAngle;
CString m_strZPt;
CString m_strYPt;
CString m_strXPt;
```

（8）在对话框初始化事件的处理函数 OnInitDialog 中，设置文本框控件的有效值范围和默认值，并且为按钮加载默认的位图，其相关代码为：

```
BOOL CArxDialog::OnInitDialog()
{
    CAcUiDialog::OnInitDialog();

    // 设置输入点的范围
    m_editXPt.SetRange(-100.0, 100.0);
    m_editYPt.SetRange(-100.0, 100.0);
    m_editZPt.SetRange(-100.0, 100.0);
```

```
        // 设置角度的输入范围
        m_editAngle.SetRange(0.0, 90.0);

        // 加载默认的位图
        m_btnPick.AutoLoad();
        m_btnAngle.AutoLoad();

        // 设置文本框的默认值
        m_strAngle = "0.0";
        m_strXPt = "0.0";
        m_strYPt = "0.0";
        m_strZPt = "0.0";

        // 显示初始点的坐标和角度值
        DisplayPoint();
        DisplayAngle();

        return TRUE;  // return TRUE unless you set the focus to a control
                      // EXCEPTION: OCX Property Pages should return
FALSE
    }
```

SetRange 函数用于设置文本框内容的有效值范围，可以使用 Validate 函数来检查内容的有效性。AutoLoad 函数用于初始化自画按钮，可以显示为按钮预设的位图。

DisplayPoint 和 DisplayAngle 是两个自定义函数，分别用于在对话框中显示点的坐标和角度值，其实现代码为：

```
    void CArxDialog::DisplayPoint()
    {
        // 在对话框中显示点的坐标
        m_editXPt.SetWindowText(m_strXPt);
        m_editXPt.Convert();  // 更新控件和其关联的成员变量
        m_editYPt.SetWindowText(m_strYPt);
        m_editYPt.Convert();
        m_editZPt.SetWindowText(m_strZPt);
        m_editZPt.Convert();
    }
    void CArxDialog::DisplayAngle()
    {
        // 在对话框中显示角度值
        m_editAngle.SetWindowText(m_strAngle);
```

```
        m_editAngle.Convert();
}
```

SetWindowText 函数是 CacUiEdit 类的基类 CWindow（MFC 中的一个类）的一个成员函数，用于修改文本框的内容。Convert 函数则用于更新控件和其关联的成员变量，使两者的值保持一致。

（9）在“选择点”按钮的单击事件中，隐藏对话框，提示用户在图形窗口中选择一个点，然后重新显示该对话框，其处理函数为：

```
void CArxDialog::OnButtonPoint()
{
        // 隐藏对话框把控制权交给AutoCAD
        BeginEditorCommand();

        // 提示用户输入一个点
        ads_point pt;
        if (acedGetPoint(NULL, "\n输入一个点:", pt) == RTNORM)
        {
                // 如果点有效，继续执行
                CompleteEditorCommand();
                m_strXPt.Format("%.2f", pt[X]);
                m_strYPt.Format("%.2f", pt[Y]);
                m_strZPt.Format("%.2f", pt[Z]);

                // 显示点的坐标
                DisplayPoint();
        }
        else
        {
                // 否则取消命令（包括对话框）
                CancelEditorCommand();
        }
}
```

BeginEditorCommand 函数用于将控制权（焦点）交给 AutoCAD，一般用于开始一个交互操作；CompleteEditorCommand 函数用于从一个在 AutoCAD 中完成的交互命令返回到应用程序；CancelEditorCommand 函数用于从一个在 AutoCAD 中被取消的交互命令返回到应用程序。这三个函数组合使用，能够在模态对话框中实现用户和 AutoCAD 的交互操作。

（10）在“选择角度”按钮的单击事件中，隐藏对话框，提示用户在图形窗口中输入一个角度值，然后重新显示该对话框，其实现函数为：

```
void CArxDialog::OnButtonAngle()
{
```

```
                    // 隐藏对话框把控制权交给AutoCAD
            BeginEditorCommand();

            // 将当前选择的点的位置作为基点
            ads_point pt;
            acdbDisToF(m_strXPt, -1, &pt[X]);
            acdbDisToF(m_strYPt, -1, &pt[Y]);
            acdbDisToF(m_strZPt, -1, &pt[Z]);

            // 提示用户输入一点
                    double angle;
                    const double PI = 4 * atan(1);
            if (acedGetAngle(pt, "\n输入角度:", &angle) == RTNORM)
                    {
                // 如果正确获得角度，返回对话框
                CompleteEditorCommand();
                // 将角度值转换为弧度值
                m_strAngle.Format("%.2f", angle * (180.0/PI));

                            // 显示角度值
                DisplayAngle();
            }
                    else
                    {
                // 否则取消命令（包括对话框）
                CancelEditorCommand();
            }
        }
```

acdbDisToF 函数根据指定的单位将一个字符串形式保存的实数转换成一个实数，其定义为：

```
int acdbDisToF(
            const char * str,
            int unit,
            ads_real * v);
```

其中，str 保存了要转换的字符串；unit 指定使用的单位，如果设置为-1，会使用 AutoCAD 图形中 LUNITS 系统变量的值作为使用的单位；v 返回转换结果。

（11）如果用户直接在对话框的文本框中修改其内容，而不是使用“选择点”或“选择角度”按钮来指定，那么在文本框内容被修改之后应该改变与其关联的成员变量，可在文本框失去焦点的事件中进行处理：

```
void CArxDialog::OnKillfocusEditAngle()
{
        // 获得并更新用户输入的值
        m_editAngle.Convert();
        m_editAngle.GetWindowText(m_strAngle);
}

void CArxDialog::OnKillfocusEditXpt()
{
        // 获得并更新用户输入的值
        m_editXPt.Convert();
        m_editXPt.GetWindowText(m_strXPt);
}

void CArxDialog::OnKillfocusEditYpt()
{
        // 获得并更新用户输入的值
        m_editYPt.Convert();
        m_editYPt.GetWindowText(m_strYPt);
}

void CArxDialog::OnKillfocusEditZpt()
{
        // 获得并更新用户输入的值
        m_editZPt.Convert();
        m_editZPt.GetWindowText(m_strZPt);
}
```

Convert 函数用于更新控件的内容和其对应的成员变量；GetWindowText 函数能够获得文本框中的内容，将其保存在对应的中间变量（例如 m_strXPt）中。

（12）为了模拟使用用户输入的点和角度，在对话框关闭的时候将其输出到 AutoCAD 的命令窗口，可以在对话框关闭的事件中进行处理，其实现函数为：

```
void CArxDialog::OnClose()
{
        // 在AutoCAD命令窗口输出选择点的结果
        double x = atof(m_strXPt);
        double y = atof(m_strYPt);
        double z = atof(m_strZPt);
        acutPrintf("\n选择的点坐标为(%.2f, %.2f, %.2f).", x, y, z);
```

```
        CAcUiDialog::OnClose();
    }
```

atof 函数用于将一个字符串常量转换成实数，而 CString 类型的变量可以直接用于代替 const char*类型作为函数的参数。

（13）注册一个新命令 ArxModal，用于以模态方式显示对话框，其实现函数为：

```
void ZffCHAP12ArxModal()
{
        // 防止资源冲突
        CAcModuleResourceOverride resOverride;

        // 显示ObjectARX的模态对话框
        CArxDialog theDialog;
        theDialog.DoModal();
}
```

如果同时加载了多个包含对话框资源的 ObjectARX 应用程序，很可能会导致资源的冲突而使一些程序无法正确运行（对话框无法正确弹出，系统提示“AutoCAD 无法打开对话框”），解决的方法就是在显示对话框之前使用“CAcModuleResourceOverride resOverride;”语句。

（14）最后，在 ZffCHAP12ArxModal 函数所在的文件开头处添加对 CArxDialog 类的包含：

```
#include “ArxDialog.h”
```

8.1.4 实例效果

（1）编译链接程序，启动AutoCAD 2002，加载生成的ARX文件。执行MfcModal命令，系统会弹出如图 8.13所示的对话框。

图8.13 基于 MFC 的对话框

（2）单击【确定】按钮，系统会弹出如图 8.14所示的提示对话框。

图8.14　提示对话框关闭

（3）执行ArxModal命令，系统会弹出如图 8.15所示的对话框。单击“选择点”按钮，系统在命令窗口提示“输入一个点:”，选择一点之后返回到对话框中，并且在文本框中显示选择点的坐标值。单击“选择角度”按钮则可以在图形窗口中指定角度值。

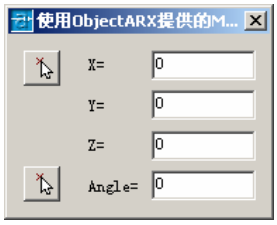

图8.15　基于 ObjectARX 的对话框

（4）关闭对话框之后，系统会在命令窗口输出下面的提示：

选择的点坐标为(517.14, 233.77, 0.00).

8.1.5 小结

学习本节之后，读者应该掌握下面的知识点：

- 模态显示对话框的一般步骤。
- 处理对话框事件的方法。
- 使用 AutoCAD 风格的控件。
- 隐藏对话框，和 AutoCAD 进行交互。
- 将 CString 类型的变量转化为 double 类型。
- 用户手工修改文本框内容之后，更新成员变量的方法。初学者一个常犯的错误是没有处理文本框被修改之后的事件，结果导致用户手工修改文本框内容之后，与其关联的成员变量并没有被更新，最终计算的结果错误。
- 防止资源冲突。

8.2 非模态对话框

8.2.1 实例说明

显示模态对话框之后，在关闭该对话框之前，用户不能对应用程序的其他界面进行操作；而非模态对话框则允许用户在该对话框和其他的界面之间自由切换焦点，这在某些情况下非常有用。

本节的程序能够在AutoCAD中显示一个非模态对话框，在该对话框未关闭的情况下，用户仍然可以执行AutoCAD的其他功能，如图8.16所示。

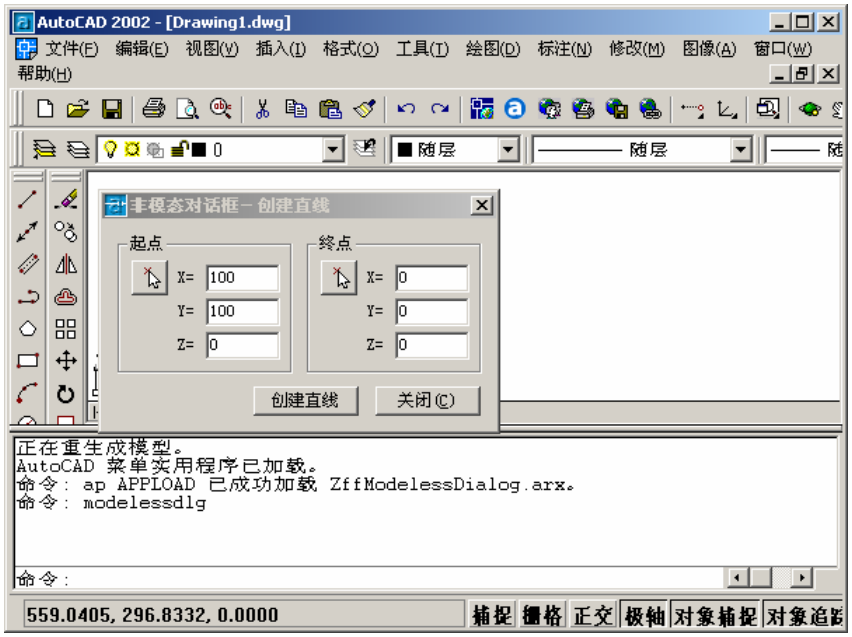

图8.16 显示非模态对话框

8.2.2 编程思路

在 ObjectARX 中，显示一个非模态对话框比显示模态对话框要复杂得多，因为对话框关闭的时间并不能在编程时确定，因此对话框的内存必须在堆上分配，销毁对话框的操作自然也需要由程序员来完成。

简单来说，显示一个非模态对话框需要下面的代码：

```
pDialog = new CModelessDlg(acedGetAcadFrame());
pDialog->Create(IDD_DIALOG_MODELESS);
pDialog->ShowWindow(SW_SHOW);
```

首先使用 new 关键字创建新的对象并分配内存，然后使用 Create 函数完成对话框的创建（显示模态对话框时，DoModal 函数内部封装了创建对话框的过程），最后使用 ShowWindow 函数确保对话框显示出来。

销毁对话框需要关闭对话框和释放对话框的内存空间，要分两部分完成，首先在关闭对话框的事件中调用 DestroyWindow 函数销毁对话框，然后需要重写对话框的 PostNcDestroy 函数，在该函数内部释放对话框所分配到的内存空间。

在非模态对话框中修改图形数据库内容时，必须在操作之前锁定文档，而在操作完成之后解锁文档。之所以这样做，是为了避免在操作图形数据库期间用户切换了图形数据库。

8.2.3 步骤

（1）启动VC++ 6.0，使用ObjectARX创建一个新工程，其名称为ModelessDialog，注意要使用MFC。插入一个对话框资源，修改其标题和字体，在其中添加如图8.17所示的控件。

图8.17　设计对话框界面

对话框的 ID 设置为 IDD_DIALOG_MODELESS，两个“拾取点”按钮的 ID 设置为 IDC_BUTTON_PICK_ST 和 IDC_BUTTON_PICK_EN，并且在属性对话框的【Styles】选项卡中选中【Owner draw】复选框；两个命令按钮的 ID 设置为 IDD_BUTTON_LINE 和 IDC_BUTTON_CLOSE，标题设置为“创建直线”和“关闭(&C)”；六个文本框的 ID 设置为 IDC_EDIT_STX、IDC_EDIT_STY、IDC_EDIT_STZ、IDC_EDIT_ENX、IDC_EDIT_ENY 和 IDC_EDIT_ENZ。

（2）单击 ObjectARX 嵌入工具栏的“ObjectARX MFC Support”按钮，在弹出的对话框中为对话框资源创建一个派生于 CAcUiDialog 的类 CModelessDlg。

（3）按下Ctrl+W显示【MFC ClassWizard】对话框，切换到【Member Variables】选项卡，双击控件ID列表中的IDC_EDIT_ENX选项，系统会弹出如图8.18所示的对话框。从【Category】列表中选择Value，从【Variable type】列表中选择CString，输入m_strEnX作为成员变量的名称，单击【OK】按钮，为文本框添加一个关联的成员变量m_strEnX。

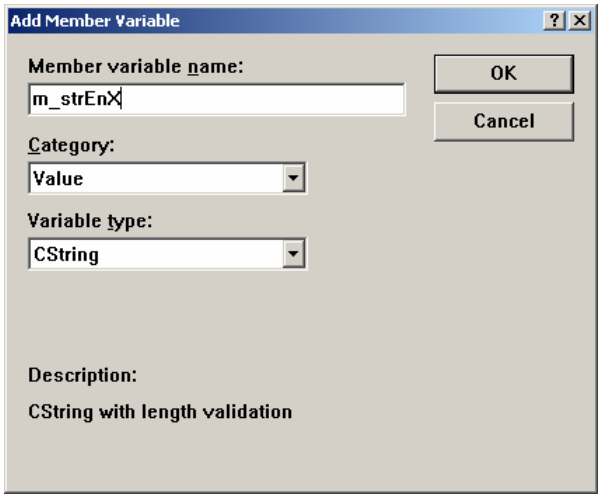

图8.18　添加 CString 类型的成员变量

（4）使用相同的方法，为其他的5个文本框分别添加关联的成员变量，完成后得到如图8.19所示的结果。

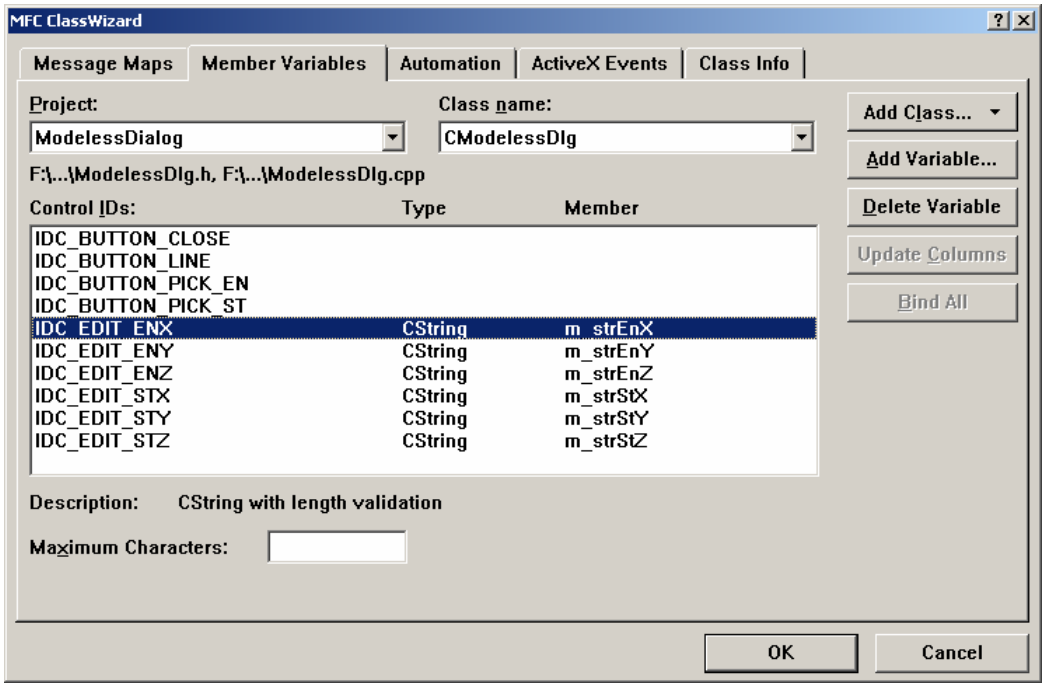

图8.19 为文本框添加成员变量

（5）使用上节介绍的方法，为两个拾取点按钮添加关联的成员变量，成员变量的类型为 CacUiPickButton。如果对添加的方法印象不太深刻，就重新温习上一节的相关内容。

（6）在【MFC ClassWizard】对话框中，切换到【Message Maps】选项卡，为四个按钮分别添加单击事件的处理函数。添加的方法是，在【Object IDs】列表中选择所要添加消息处理的按钮的 ID，然后双击右侧【Messages】列表中的【BN_CLICKED】选项，并在弹出的对话框中单击【OK】按钮，完成按钮单击事件处理函数的添加。

（7）为对话框的初始化和关闭事件添加处理函数，并对PostNcDestroy函数进行重写（OverWrite，注意区别于OverLoad，后者是重载）。其方法是，在左侧的【Object IDs】列表中选择CModelessDlg类，然后在右侧的【Messages】列表中分别双击【WM_CLOSE】和【WM_INITIDIALOG】选项，完成处理函数的添加。此时，成员函数列表中已经包含了添加的消息处理函数，如图8.20所示。

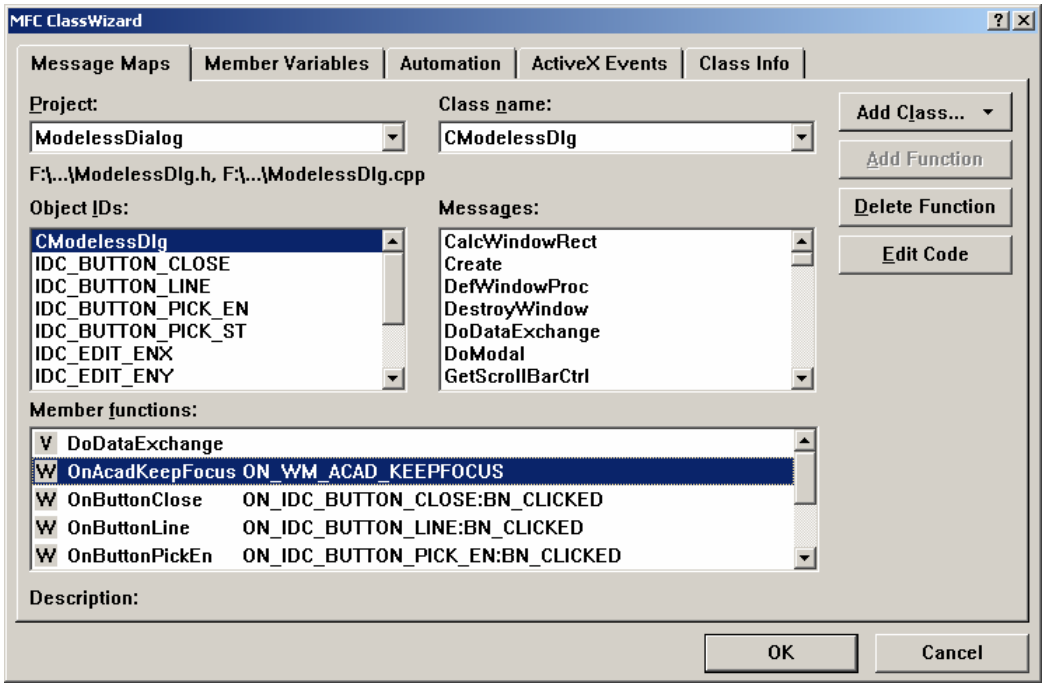

图8.20　添加消息处理函数的结果

从图形可以看出，ObjectARX 的向导已经自动为我们添加了一个消息处理函数 OnAcadKeepFocus，该函数的作用将在后面介绍。

（8）在 ModelessDialog.cpp 文件（入口点函数 acrxEntryPoint 所在的文件）中添加一个全局变量 pDialog，并且添加对 ModelessDlg.h 文件的包含。注意，全局变量 pDialog 的定义可以紧跟着_hdllInstance 变量的定义：

```
#include "ModelessDlg.h"

HINSTANCE _hdllInstance =NULL ;
CModelessDlg *pDialog = NULL;        // 非模态显示的窗体
```

（9）使用 ObjectARX 嵌入工具栏的工具注册一个新命令 ModelessDlg，并在命令实现函数所在的文件中添加对 ModelessDlg.h 文件的包含和全局变量 pDialog 的声明（注意，不是定义，只是声明该变量已经在其他的文件中定义），最终该文件的内容为：

```
#include "StdAfx.h"
#include "StdArx.h"
#include "ModelessDlg.h"

extern CModelessDlg *pDialog;

// This is command 'MODELESSDLG'
void ZffCHAP12ModelessDlg()
{
```

```
        CAcModuleResourceOverride resOverride;

        // 以非模态方式启动对话框
        if (pDialog == NULL)
        {
            pDialog = new CModelessDlg(acedGetAcadFrame());
            pDialog->Create(IDD_DIALOG_MODELESS);
            pDialog->ShowWindow(SW_SHOW);
        }
        else
        {
            pDialog->ShowWindow(SW_SHOW);
        }
    }
```

上面的代码完成了非模态对话框的创建和显示。CAcModuleResourceOverride resOverride; 语句是为了防止资源冲突，应该在所有的对话框显示之前执行该语句。如果 pDialog 的值为 NULL，表示对话框没有被创建，就使用 Create 函数创建对话框，并使用 ShowWindow 显示对话框。

注意一种特殊情况，如果用户在显示对话框的时候按下了 Esc 键，对话框会消失，但是并没有被销毁，而只是被隐藏了，pDialog 变量的值不会是 NULL。为了保证对话框被隐藏之后仍然能够在执行命令时显示出来，就可以在 pDialog 不为 NULL 的情况直接调用 ShowWindow 函数。

（10）在对话框初始化的处理函数中，对两个“拾取点”按钮进行初始化，加载默认的位图，其相关代码为：

```
    BOOL CModelessDlg::OnInitDialog()
    {
        CAcUiDialog::OnInitDialog();

        // “拾取点”按钮位图的加载
        m_btnPickEnd.AutoLoad();
        m_btnPickStart.AutoLoad();

        return TRUE;  // return TRUE unless you set the focus to a control
                      // EXCEPTION: OCX Property Pages should return
FALSE
    }
```

CModelessDlg 是 CAcUiDialog 的一个派生类，在派生类的成员函数中调用基类对应的成员函数，能够执行系统默认的初始化内容，节省大量的编码工作。

（11）让我们把注意力转移到非模态对话框的销毁。在“关闭”按钮的单击事件中，使

用 DestroyWindow 函数来手工销毁对话框，其实现代码为：

```
void CModelessDlg::OnButtonClose()
{
        // 销毁对话框
        DestroyWindow();
}
```

DestroyWindow 是 CWindow 类（MFC 中的一个类，CAcUiDialog 间接派生于此类）的一个成员函数，用于销毁与 CModelessDlg 类关联的对话框，但并不会释放对话框的内存空间。

（12）由于创建对话框是在堆上进行的，因此需要我们自己释放它所占用的内存空间，最合适的位置是在 PostNcDestroy 函数中，这个函数会在对话框销毁之后自动调用。可以用下面的代码完成对话框内存空间的释放：

```
void CModelessDlg::PostNcDestroy()
{
        // 释放非模态对话框的内存
        delete this;

        if (pDialog != NULL)
        {
                pDialog = NULL;
        }

        CAcUiDialog::PostNcDestroy();
}
```

由于在上面的代码中使用了全局变量 pDialog，因此必须在 ModelessDlg.cpp 文件的开头添加该变量的声明：

```
extern CModelessDlg *pDialog;
```

关于堆和栈的区别，可以从一个由 C/C++编译的程序占用的内存分类来说明：

- 栈区（stack）：由编译器自动分配和释放，用于存放函数的参数值、局部变量的值等，其操作方式类似于数据结构中的栈。
- 堆区（heap）：一般由程序员分配和释放，若程序员不释放，程序结束时可能由操作系统回收。注意它与数据结构中的堆是两回事，分配方式类似于链表。
- 全局区（静态区 static）：全局变量和静态变量的存储是放在一起的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由操作系统自动释放。
- 文字常量区：常量字符串就放在这里，程序结束后由操作系统自动释放。
- 程序代码区：存放函数体的二进制代码。

（13）如果用户单击了对话框右上角的关闭按钮，对话框仍然会被关闭，在这种情况下，仍然需要调用 DestroyWindow 函数来销毁对话框，其相关代码为：

```
void CModelessDlg::OnClose()
```

```
{
        CAcUiDialog::OnClose();

        // 销毁对话框
        DestroyWindow();
}
```

（14）可能在退出 AutoCAD 的时候对话框仍然没有被关闭，因此最好在程序卸载的时候销毁对话框并释放内存空间，可以在 ModelessDialog.cpp 文件中创建一个全局函数 CloseDialog，其实现代码为：

```
BOOL CloseDialog()
{
        if (pDialog == NULL)
        {
            return TRUE;
        }

        BOOL bRet = pDialog->DestroyWindow();

        if (bRet)
        {
            pDialog = NULL;
        }

        return bRet;
}
```

该函数检测对话框是否已经关闭，如果没有关闭就销毁它。

可以在 UnloadApplication 函数中添加对 CloseWindow 函数的调用，其相关代码为：

```
void UnloadApplication()
{
        // NOTE: DO NOT edit the following lines.
        //{{AFX_ARX_EXIT
        acedRegCmds->removeGroup("ZFFCHAP12");
        //}}AFX_ARX_EXIT

        // TODO: clean up your application
        CloseDialog();      // 关闭非模态对话框
}
```

（15）非模态对话框中的两个“拾取点”按钮分别用于拾取要创建的直线的起点和终点，其原理与模态对话框中拾取点完全相同，相关的代码为：

```
void CModelessDlg::OnButtonPickEn()
{
        // 隐藏对话框把控制权交给AutoCAD
        BeginEditorCommand();

        // 提示用户输入一个点
        ads_point pt;
        if (acedGetPoint(NULL, "\n输入一个点:", pt) == RTNORM)
        {
            // 如果点有效，继续执行
            CompleteEditorCommand();
            m_strEnX.Format("%.2f", pt[X]);
            m_strEnY.Format("%.2f", pt[Y]);
            m_strEnZ.Format("%.2f", pt[Z]);
        }
        else
        {
            // 否则取消命令（包括对话框）
            CancelEditorCommand();
        }

        // 用成员变量的值更新文本框显示的内容
        UpdateData(FALSE);
}

void CModelessDlg::OnButtonPickSt()
{
        // 隐藏对话框把控制权交给AutoCAD
        BeginEditorCommand();

        // 提示用户输入一个点
        ads_point pt;
        if (acedGetPoint(NULL, "\n输入一个点:", pt) == RTNORM)
        {
            // 如果点有效，继续执行
            CompleteEditorCommand();
            m_strStX.Format("%.2f", pt[X]);
            m_strStY.Format("%.2f", pt[Y]);
            m_strStZ.Format("%.2f", pt[Z]);
```

```
        }
        else
        {
            // 否则取消命令（包括对话框）
            CancelEditorCommand();
        }

        // 用成员变量的值更新文本框显示的内容
        UpdateData(FALSE);
}
```

（16）对话框中“创建直线”按钮用于向模型空间添加一条直线，其实现代码为：

```
void CModelessDlg::OnButtonLine()
{
        // 获得起点和终点的坐标
        AcGePoint3d ptStart(atof(m_strStX), atof(m_strStY), atof(m_strStZ));
        AcGePoint3d ptEnd(atof(m_strEnX), atof(m_strEnY), atof(m_strEnZ));

        // 锁定文档
        acDocManager->lockDocument(curDoc());

        AcDbLine *pLine = new AcDbLine(ptStart, ptEnd);

        AcDbBlockTable *pBlockTable;
        acdbHostApplicationServices()->workingDatabase()
            ->getBlockTable(pBlockTable, AcDb::kForRead);

        AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);

        AcDbObjectId lineId;
    pBlockTableRecord->appendAcDbEntity(lineId, pLine);

        pBlockTable->close();
    pBlockTableRecord->close();
    pLine->close();

        // 解锁文档
        acDocManager->unlockDocument(curDoc());
```

```
}
```

与模态对话框中创建直线相比，仅仅多了两个操作，就是在创建直线之前使用下面的语句锁定文档：

```
acDocManager->lockDocument(curDoc());
```

在创建直线之后，使用下面的语句解锁文档：

```
acDocManager->unlockDocument(curDoc());
```

之所以需要在非模态对话框操纵图形数据库时锁定文档，是为了避免在修改图形数据库期间用户切换了图形数据库（非模态对话框显示期间用户可以自由切换图形数据库）。

acDocManager 是一个全局函数，能够返回一个文档管理器的指针。curDoc 是一个全局函数，能够返回当前的图形文档。

8.2.4 实例效果

（1）编译运行程序，启动AutoCAD 2002，加载生成的ARX文件。执行ModelessDlg命令，系统会弹出如图8.21所示的对话框。

图8.21　默认的对话框

（2）单击拾取点按钮，可以在图形窗口中拾取起点和终点，然后单击【创建直线】按钮，就能在图形窗口中完成直线的创建。

（3）单击【关闭】按钮或者右上角的【╳】按钮，均可以关闭当前显示的非模态对话框。

8.2.5 小结

学习本节内容之后，读者需要掌握下面的知识点：

- 创建和显示非模态对话框的方法，以及与显示模态对话框的区别。
- 销毁非模态对话框的方法，注意按下 Esc 键的处理。
- 在非模态对话框中修改图形数据库的方法。

8.3 可停靠窗体

8.3.1 说明

AutoCAD 提供的属性窗口、设计中心等窗口，都是可停靠的窗体，本节我们使用 ObjectARX 推荐的方式来实现这种效果。

8.3.2 思路

ObjectARX 提供了 CAcUiDockControlBar 类来简化可停靠窗体的实现，使用这个类只需要通过下面的几个步骤就可以实现可停靠的窗体：

（1）创建一个对话框，用非模态的方式来实现对话框的按钮消息（也就意味着操作图形数据库时需要进行文档锁定）；

（2）添加一个 CAcUiDockControlBar 类的一个派生类，在派生类中进行对话框的创建工作，将对话框作为派生类的一个成员，实际上是将对话框嵌入到可停靠的窗体中；

（3）在应用程序加载或者命令执行的时候动态创建可停靠的窗体，并且在应用程序卸载的时候清理资源。

8.3.3 步骤

（1）启动VC++ 6.0，使用ObjectARX创建一个新工程，其名称为DockControlBar，注意要使用MFC。插入一个对话框资源，修改其标题和字体，在其中添加如图8.22所示的控件。

图8.22 对话框资源

对话框的 ID 设置为 IDD_DIALOG，字体设置为宋体9号（与 AutoCAD 自身保持一致）。按钮的 ID 设置为 IDC_LINE，标题设置为“绘制直线(&D)”。

（2）为了将对话框能够作为子窗体嵌入到其他窗体中，需要对资源进行一些特殊的设置，首先在对话框的属性窗口中，切换到【Styles】选项卡，调整对话框样式和边框的设置，如图8.23所示。

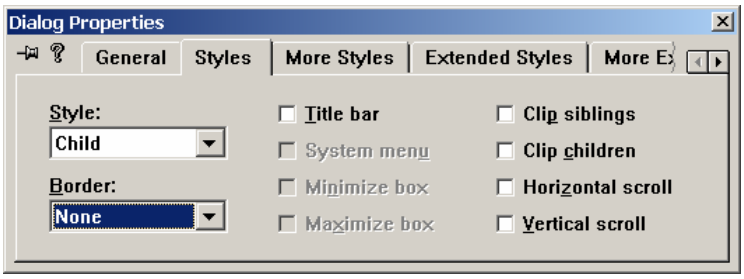

图8.23　修改样式和边框

（3）切换到【More Styles】选项卡，选中【Visible】复选框，如所示。这样当这个窗体被创建之后直接会显示出来，这与调用对话框的 ShowWindow 函数作用是一致的。

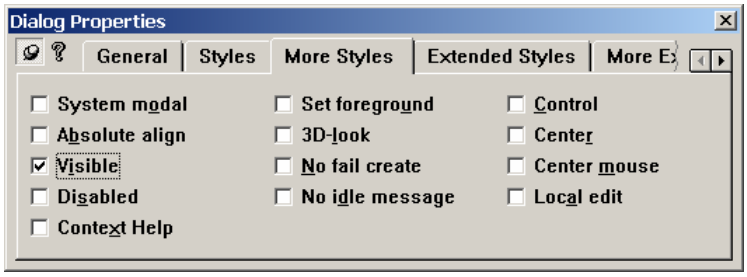

图8.24　选中【Visible】复选框

（4）单击ObjectARX嵌入工具栏的“ObjectARX MFC Support”按钮，系统弹出如图8.25所示的对话框。输入CChildDlg作为类的名称，选择CAcUiDialog为基类，IDD_DIALOG为要创建的对话框的资源，单击【Create Class】按钮来添加对话框对应的类。

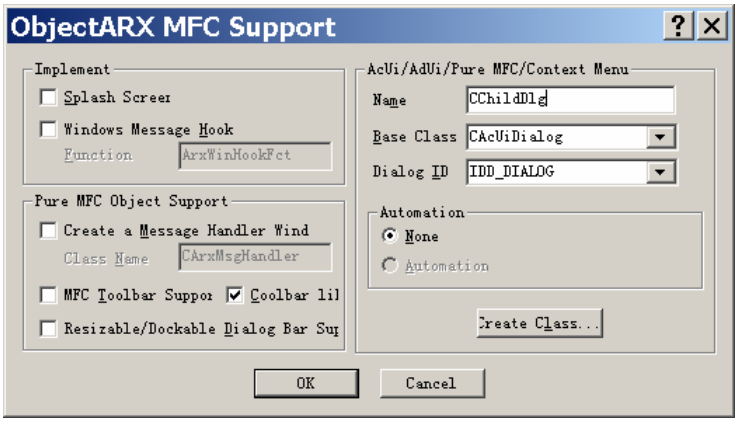

图8.25　添加对话框对应的类

（5）不要关闭这个对话框，从基类的列表中选择CAcUiDockControlBar，输入CMyDockControlBar作为派生类的名称，如图8.26所示，单击【Create Class】按钮创建可停靠窗体的类，单击【OK】按钮关闭对话框。

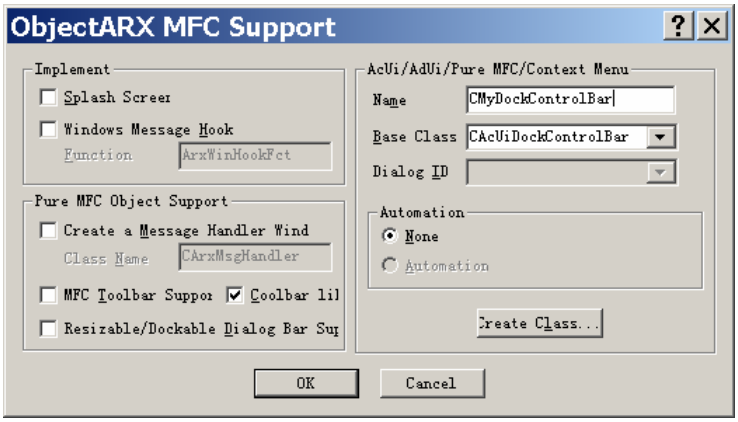

图8.26 添加可停靠窗体的类

（6）为了能够早测试（这是敏捷开发的一个非常好的思想），我们先做的是将可停靠窗体在 AutoCAD 中显示出来。简单起见，不注册新命令，直接在应用程序加载的时候显示这个窗体。

在 DockControlBar.cpp 文件中添加一个全局变量：

```
CMyDockControlBar* g_pDlgBar = NULL;
```

同时添加它需要的头文件：

```
#include "MyDockControlBar.h"
```

（7）在应用程序被加载后初始化的消息处理函数中，显示可停靠的窗体：

```
void InitApplication()
{
        // NOTE: DO NOT edit the following lines.
        //{{AFX_ARX_INIT
        //}}AFX_ARX_INIT

        // 显示可停靠的窗体
        CAcModuleResourceOverride resOverride;
        if (g_pDlgBar == NULL)
        {
                g_pDlgBar = new CMyDockControlBar();
                g_pDlgBar->Create(acedGetAcadFrame(), _T("DockBar"));
                g_pDlgBar->SetWindowText(_T("MyControlBar"));
                g_pDlgBar->EnableDocking(CBRS_ALIGN_ANY);
        }

        acedGetAcadFrame()->FloatControlBar(g_pDlgBar, CPoint(100, 100),
CBRS_ALIGN_TOP);        // 初始位置
        acedGetAcadFrame()->ShowControlBar(g_pDlgBar, TRUE, TRUE);
}
```

（8）在应用程序卸载的消息处理函数中，手工销毁这个窗体：

```
void UnloadApplication()
{
        // NOTE: DO NOT edit the following lines.
        //{{AFX_ARX_EXIT
        //}}AFX_ARX_EXIT

        // 手动销毁可停靠窗体
        if (g_pDlgBar != NULL)
        {
                g_pDlgBar->DestroyWindow();
                delete g_pDlgBar;
                g_pDlgBar = NULL;
        }
}
```

（9）编译程序，在AutoCAD加载应用程序，AutoCAD图形窗口中显示出如图8.27所示的窗体，这个窗体可以停靠在任何位置，但是没有任何的控件。

图8.27　第一步显示的可停靠窗体

尽管如此，我们走出了可喜的第一步，至少我们能够看到一个能停靠的窗体了，至于为什么没有显示任何控件？我们还没有把窗体嵌入进去呢。

（10）在 CMyDockControlBar 类中，添加一个私有的成员变量用于表示要嵌入的子窗体：

```
CChildDlg* m_childDlg;
```

同时添加它需要的头文件：

```
#include "ChildDlg.h"
```

（11）在对话框的构造函数中对这个成员变量进行初始化：

```
CMyDockControlBar::CMyDockControlBar() : CAcUiDockControlBar () {
        //{{AFX_DATA_INIT(CMyDockControlBar)
        m_childDlg = NULL;
        //}}AFX_DATA_INIT
}
```

（12）在 OnCreate 函数中添加显示子窗体的代码：

```
int CMyDockControlBar::OnCreate (LPCREATESTRUCT lpCreateStruct) {
        if ( CAcUiDockControlBar::OnCreate (lpCreateStruct) == -1 )
            return (-1) ;

        // 显示子窗体
        CAcModuleResourceOverride resourceOverride;
        m_childDlg = new CChildDlg();
        m_childDlg->Create(IDD_DIALOG, this);
        //m_childDlg->ShowWindow(SW_SHOW);
        m_childDlg->MoveWindow(0, 0, 160, 160, TRUE);

        return (0) ;
}
```

（13）在 SizeChanged 函数中处理子窗体大小与可停靠窗体的匹配：

```
void CMyDockControlBar::SizeChanged (CRect *lpRect, BOOL bFloating, int flags) {
        //
        CAcModuleResourceOverride resourceOverride;
        if (m_childDlg != NULL)
        {
            m_childDlg->SetWindowPos(this, lpRect->left, lpRect->top,
lpRect->Width(), lpRect->Height(), SWP_NOZORDER);
        }
}
```

（14）再次编译运行程序，系统弹出如图8.28所示的结果。我们创建的对话框已经添加到可停靠窗体中，成为它的一部分，不过单击按钮还没有任何效果，因为还没有对按钮的单击事件进行处理。

图8.28 嵌入子窗体之后的效果

（15）下面来处理对话框中【绘制直线】按钮的单击事件：

```
void CChildDlg::OnLine()
```

```
{
        // 类似于非模态对话框，需要锁定和解锁文档
        acDocManager->lockDocument(acDocManager->curDocument());

        // 绘制一条直线
        AcGePoint3d startPt(4.0, 2.0, 0.0);
    AcGePoint3d endPt(10.0, 7.0, 0.0);
    AcDbLine *pLine = new AcDbLine(startPt, endPt);

    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);

    AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);
    pBlockTable->close();

    AcDbObjectId lineId;
    pBlockTableRecord->appendAcDbEntity(lineId, pLine);

    pBlockTableRecord->close();
    pLine->close();

        acDocManager->unlockDocument(acDocManager->curDocument());
}
```

用操作图形数据库的代码（添加一条直线）来作为处理的事件，是为了让大家注意在可停靠窗体中操作图形数据库必须要锁定当前文档。到此为止，整个例子完成，请继续编译运行观察效果。

可能漏掉了一句话（整本书中，我可能多次会漏掉这样的话），就是需要在 CChildDlg 类的实现文件中添加头文件：

```
#include "dbents.h"
```

8.3.4 效果

编译运行程序之后，可以将停靠的窗口拖动放置在图形窗口的边缘，如图8.29所示。

图8.29 停靠窗体的效果

8.3.5 小结

学习本节内容之后，读者需要掌握下面的知识点：

- 将一个窗体作为子窗体嵌入到可停靠窗体中；
- 在可停靠窗体中访问图形数据库需要锁定文档。

8.4 使用 MFC 创建工具栏

8.4.1 说明

在 ObjectARX 应用程序中，你能够使用三种方式来创建工具栏：

- 在菜单文件中定义工具栏；
- 通过 Com 方式动态创建工具栏；
- 使用 MFC 创建工具栏。

三种方式各有优缺点，大家可以根据自己的需要选择合适的技术。

8.4.2 思路

ObjectARX 提供了良好的扩展机制，可以将 MFC 中创建的工具栏资源直接在 AutoCAD

窗口中使用，这种方式使用的技术与在 VC 创建的独立应用程序中创建工具栏几乎完全相同。

8.4.3 步骤

（1）创建支持 MFC 的 ObjectARX 工程。

（2）单击ARX向导工具栏的“ObjectARX MFC Support”按钮，系统弹出如图8.30所示的对话框，注意选择的两个复选框。

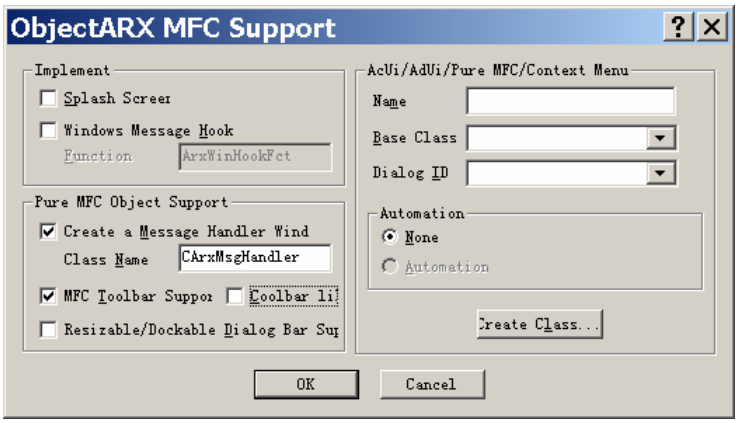

图8.30　创建 MFC 工具栏的消息映射类

（3）单击【OK】按钮，系统创建了 CAcToolBar 和 CArxMsgHandler 两个类。

（4）通过资源操作插入工具栏，如图8.31所示。

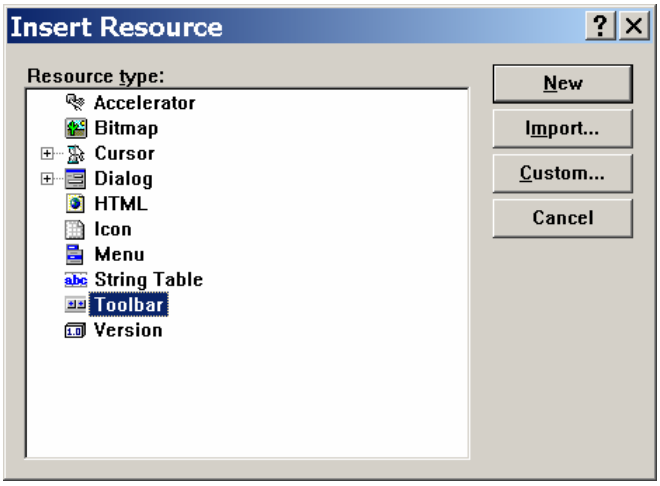

图8.31　插入工具栏资源

添加按钮，并且修改按钮的ID，如图8.32所示。

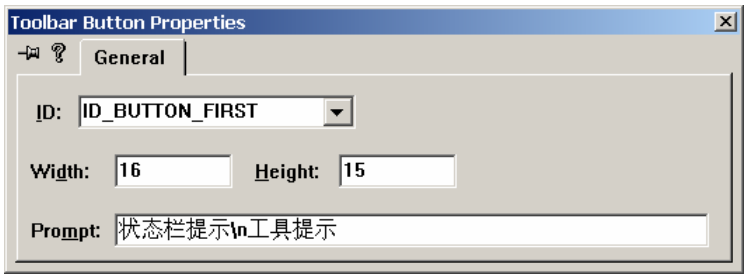

图8.32　设置工具栏上的按钮 ID

（5）选中工具栏资源的情况下按下Ctrl+W，弹出如图8.33所示的对话框。

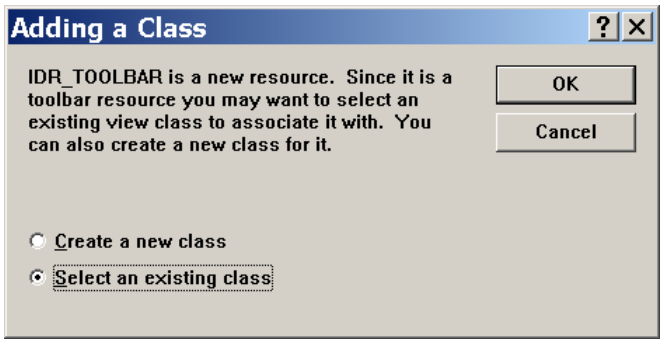

图8.33　添加资源的映射类

选择ARX向导自动生成的那个类，如图8.34所示。

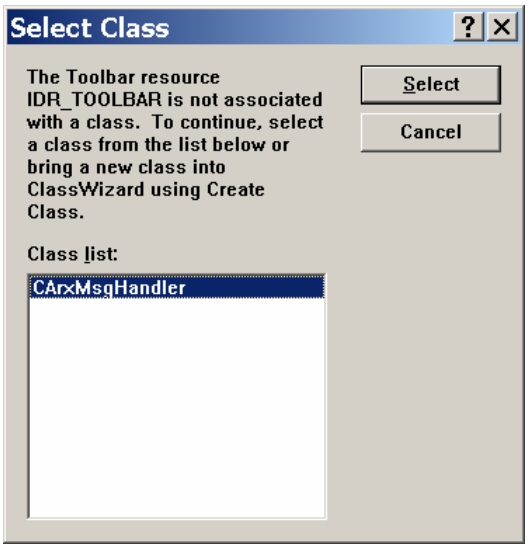

图8.34　选择工具栏资源的映射类

（6）为 CArxMsgHandler 类添加一个成员变量：

```
CAcToolBar* m_pToolbar;
```

（7）为CArxMsgHandler类映射WM_CREATE和WM_DESTROY消息，如图8.35所示。

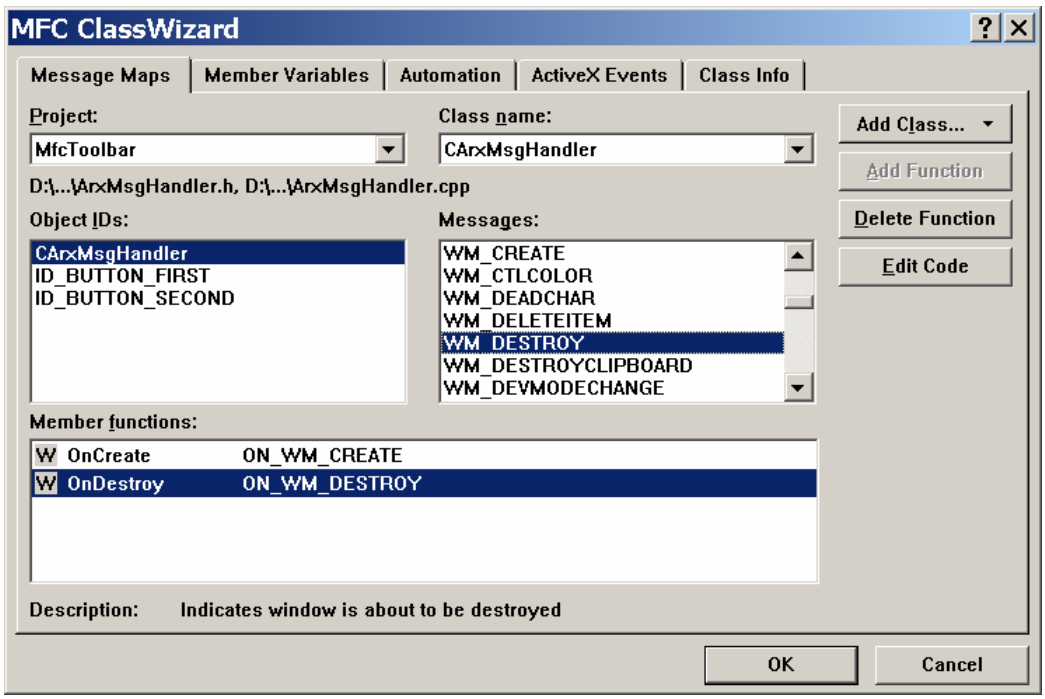

图8.35　映射消息

（8）在 WM_CREATE 事件的消息处理函数中，创建工具栏：

```
int CArxMsgHandler::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CWnd::OnCreate(lpCreateStruct) == -1)
            return -1;

        // 创建工具栏
        m_pToolbar = CreateToolBar(IDR_TOOLBAR, this, "工具栏");

        return 0;
}
```

（9）在 WM_DESTROY 事件的消息处理函数中，销毁工具栏：

```
void CArxMsgHandler::OnDestroy()
{
        CWnd::OnDestroy();

        // 销毁工具栏
        delete m_pToolbar;
        acedGetAcadFrame()->RecalcLayout();
}
```

这时候，编译后在 AutoCAD 中加载应用程序，就能够在图形窗口中显示工具栏了，但

是工具栏按钮还没有任何的功能。

（10）对于工具栏按钮，可以像MFC中普通的工具栏那样进行按钮单击事件的消息处理，如图8.36所示。

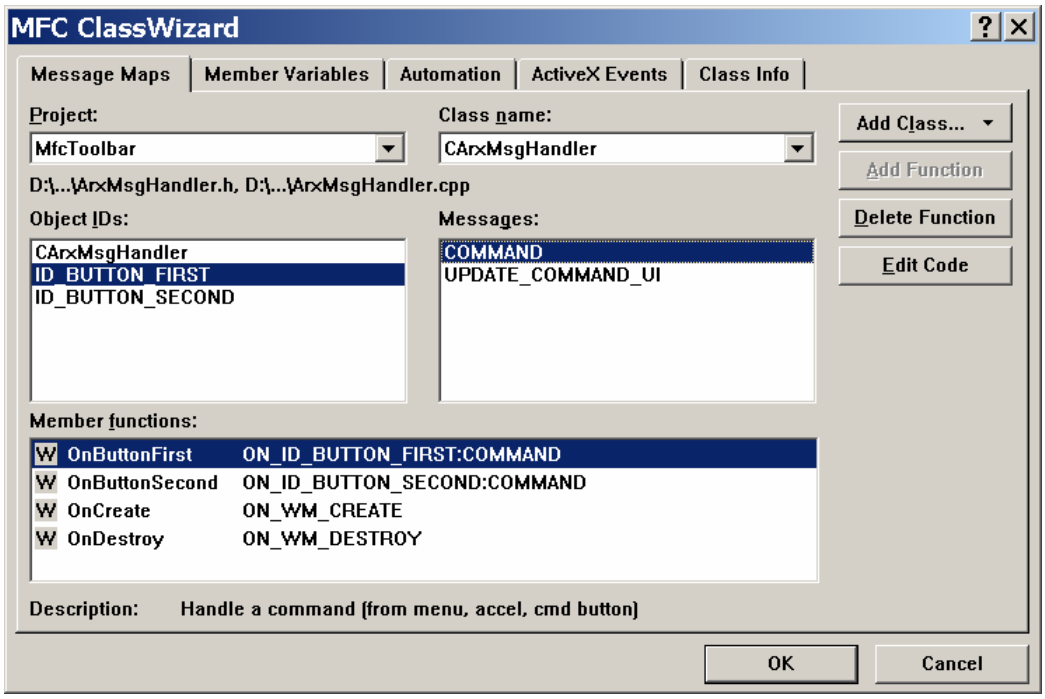

图8.36 为工具栏按钮添加消息处理函数

（11）简单起见，为这两个按钮添加下面的消息处理函数：

```
void CArxMsgHandler::OnButtonFirst()
{
        AfxMessageBox("按钮1");
}

void CArxMsgHandler::OnButtonSecond()
{
        AfxMessageBox("按钮2");
}
```

8.4.4 效果

（1）编译运行程序，启用AutoCAD 2002，加载应用程序之后，在图形窗口中可以看到动态创建的工具栏，如图8.37所示。

图8.37　在 AutoCAD 中生成的工具栏

（2）单击按钮1，系统弹出如图8.38所示的消息提示。

图8.38　按钮消息提示

8.4.5 小结

如果用户关闭了工具栏，可以通过下面的命令重新显示它。

```
void zhfZhifanRTDDrawTunnel()
{
        CAcToolBar* p = GetArxMsgHandler()->GetToolBar();
        acedGetAcadFrame ()->FloatControlBar (p, CPoint (100, 100),
CBRS_ALIGN_TOP) ;
        acedGetAcadFrame ()->ShowControlBar (p, TRUE, TRUE) ;
}
```