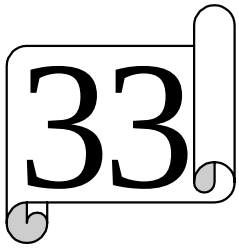

用托管 C++进行 Windows Forms 编程

在上一章中，我们介绍了用 C++托管扩展来编写托管代码的基础知识。现在我们可以用这些基础知识来做一些实际的事情了。公共语言运行时库提供了许多特性和服务，其中两个最为流行的特性是 Windows Forms 和 ASP.NET：Windows Forms 主要用于创建桌面应用程序，而 ASP.NET 则主要用于编写 Web 应用程序。我们将从 Windows Forms 开始。另一个特性是托管的数据访问机制。我们将在第 35 章介绍用 ADO.NET 进行托管的数据访问。

33.1 Windows Forms

本章前面的几部分讲述了经典的 Microsoft Windows 开发技术。我们把 MFC 看作是编写高性能 Windows 应用程序最快捷的途径。很多年以来，要想编写出最高性能的、功能丰富的桌面应用程序，最佳的途径就是使用 MFC。现在，Microsoft .NET 包含了一个名为 Windows Forms 的桌面应用程序开发框架。

尽管.NET 主要的重点在于基于 Internet 的开发，但是普通的客户应用程序总是受欢迎的。Windows 的用户界面已经有相当长一段时间了，这些基础不可能会很快消失。从根本上来说，在可预见的将来，Windows 应用程序可能仍然会保持不变。您可能仍然需要使用 WndProc 函数和 Petzold 风格的代码，或者使用 MFC 来编写 Windows 应用程序。Windows Forms 则为 Windows 开发人员提供了一个最高层次的抽象。它们使用了基于窗体的开发方式，非常类似于 Microsoft Visual Basic。然而，Windows Forms 是一套面向所有开发人员的用户界面设施，其中也包括使用托管 C++的开发人员，而对于这样的用户界面设施，Visual Basic 开发人员已经享受了很多年了。

33.1.1 Windows Forms 的本质

作为一名 MFC 程序员，您一定习惯于在 C++中只与单个类库打交道的方式了。.NET 的 Windows

Forms 库则有所不同。Windows Forms 类被内置在.NET 公共语言运行时库中。在本书前面我们已经看到了，MFC 基本上是 API 层之上的一层薄薄的编程接口。如果您浏览一下 MFC 的源代码，您将会找到 WinMain 函数以及一些消息循环——这些正是所有 Windows 程序的核心。实际上，所有 Windows 应用程序的基本工作方式都是相同的。Windows 应用程序首先注册一些窗口类，这些窗口类把一个 WndProc 与一种默认的窗口风格关联起来。然后，Windows 应用程序使用这些窗口类来创建 Windows 用户界面元素的实例。在 Windows 内部，有一些基本的窗口类已经被预先定义好了(例如 BUTTON 和 COMBOBOX 类)。

在早期的 Windows 程序设计年代，所有的应用程序都需要从头开始创建，而开发人员的大部分时间都被花在了如何保证样板代码的正确性上。一旦样板代码可以正常工作了，您就可以逐渐地增加一些事件控制函数，以便实现该应用程序的功能，做法则是在 switch 语句中加入一些 case 分支。MFC 使得开发人员不用再按照这样的方式来开发应用了，它甚至把 WinMain 和 WndProc 函数都隐藏了起来。Windows Forms 继续坚持这种做法，它进一步隐藏了一些编程细节，所以您无需再花太多的时间来编写那些乏味的代码。

33.1.2 Windows Forms 结构

Windows Forms 应用程序的结构组织非常类似于 Visual Basic 应用程序，而且 Win-dows Forms 的开发也与 Visual Basic 中标准的、基于窗体的开发方式类似。SDK 风格的应用程序则直接与 Windows API 打交道。我们在前面已经看到，MFC 只是 API 和 C++源代码之间薄薄的一层。Windows Forms 程序设计隐藏了 Windows 程序设计的许多样板细节，比 MFC 的做法有过之而无不及。Windows Forms 应用程序也具有普通 Windows 应用程序的所有一般化特性。它们也响应常见的事件，比如鼠标移动或者菜单选择。Windows Forms 也可以在客户区域内绘制图形。然而，Windows Forms 管理这些特性的语法比您用 SDK 或者 MFC 来编写程序所用到的语法要更为抽象。

Windows Forms 技术对于创建所有的标准 Windows 应用程序都非常有用，我们前面见过的标准 Windows 应用程序有：单文档界面(SDI)应用程序、多文档界面(MDI)应用程序和基于对话框的应用程序。Windows Forms 开发的大部分工作都涉及到管理一个(或多个)窗体，并从控件(组合框、标签、文本框等等)的角度来定义一个用户界面。所有这些控件在公共语言运行时库中都可以找到。Windows Forms 并不只限于基于窗体的应用程序。Windows Forms 也包含了一个画布(canvas)，您可以在画布上绘制任何东西——就像您用标准 GDI 设备环境所能够做到的那样。

Windows Forms 从多条途径简化了桌面用户界面的程序设计。例如，Windows Forms 以属性的形式定义了窗体的外观显示。为了通过编程来实现窗体在屏幕上的移动，您可以设置 Windows Forms 的 Location 属性。还记得吗？在用 MFC 来编程的时候，移动一个窗口要涉及到调用 CWnd::MoveWindow。Windows Forms 通过方法来管理它们的行为；它们也响应事件，以此来定义它们与用

户之间的交互行为。

构成 Windows Forms 应用程序的类都可以在公共语言运行时库中找到。Windows Forms 应用程序背后的基本类是 System::Windows::Forms::Form 类。所谓编写一个 Windows Forms 应用程序，实际上就是设置这个类的属性，以便让窗口看起来像您所期望的那样，并且为鼠标移动、菜单和命令增加事件控制函数。因为 Windows Forms 的窗体是一个普通的公共语言运行时库类，它完全支持继承，所以您可以按照标准的面向对象方式来建立一个基于 Windows Forms 类的层次结构。目前，公共语言运行时库只包含了用于创建应用程序的最基本的类。不过，第三方软件厂商正在快速地构造 Windows Forms 组件和控件。

33.1.3　一个 Windows Forms 向导

Microsoft Visual Studio .NET 包含了一个被称为 Managed C Windows Forms Wizard 的向导，它可以产生一个 Windows Forms 应用程序。您在 Visual Studio 的联机帮助中搜索“自定义向导示例”，就可以找到该向导。为了安装该向导，请单击 ManagedCWinFormWiz 链接，并按照指令往下走。我们将使用该向导来创建一个简单的 Windows Forms 应用程序，以便看一看 Windows Forms 是如何工作的。

Ex33a 示例程序：一个包含菜单和状态栏的基本 Windows Forms 应用程序

您可以浏览一下附带 CD 上的 Ex33a 项目，或者您也可以用 Managed C Windows Forms Wizard 来产生本示例程序。为了使用该向导，请首先确保已经安装了该向导。(当您下载该向导的时候，您可以获得关于如何安装该向导的信息。)从 File 菜单中选择 New，单击 Project，然后选择 Managed C++ Windows Forms 项目(托管的 C++ Windows 窗体项目)。在 Name 文本框中输入 **Ex33a**，再单击 OK。以下是该向导所产生的代码：

Source.cpp

```
#using <mscorlib.dll>
using namespace System;

// required dlls for WinForms
#using "System.dll"
#using "System.Windows.Forms.dll"
#using "System.Drawing.dll"

// required namespaces for WinForms
using namespace System::ComponentModel;
using namespace System::Windows::Forms;
using namespace System::Drawing;

__gc class WinForm: public Form
```

```
{
private:
    StatusBar  *statusBar;
    Button     *closeButton;
    MainMenu   *mainMenu;
    MenuItem   *fileMenu;
    Label      *todoLabel;

    String     *caption;   // Caption of the WinForm
    int        width;     // width of the WinForm
    int        height;    // height of the WinForm


public:
    WinForm()
    {
        // Set caption and size of the WinForm
        caption = "Default WinForm Example";
        width = 400;
        height = 500;

        InitForm();
    }

    void Dispose(bool disposing)
    {
        // Form is being destroyed.  Do any
        //  necessary clean-up here.
        Form::Dispose(disposing);
    }

    void InitForm()
    {
        // Setup controls here

        // Basic WinForm Settings
        Text = caption;
        Size = Drawing::Size(width, height);

        // Setup Menu
        mainMenu = new MainMenu();
        fileMenu = new MenuItem("&File");
        mainMenu->MenuItems->Add(fileMenu);
        fileMenu->MenuItems->Add(new MenuItem("E&xit",
```

```
            new EventHandler(this, &WinForm::OnFileExit)));
        Menu = mainMenu;

        // Label
        todoLabel = new Label();
        todoLabel->Text = "TODO: Place your controls here.";
        todoLabel->Size = Drawing::Size(150, 100);
        todoLabel->Location = Point (50, 50);
        Controls->Add(todoLabel);

        // Set status bar
        statusBar = new StatusBar();
        statusBar->Text = "Status Bar is Here";
        Controls->Add(statusBar);

        // Setup Close Button
        closeButton = new Button();
        closeButton->Text = "&Close";
        closeButton->Size = Drawing::Size(75, 23);
        closeButton->TabIndex = 0;
        closeButton->Location =
            Drawing::Point(width/2 - (75/2), height - 23 - 75);
        closeButton->Click +=
            (new EventHandler(this, &WinForm::OnCloseButtonClick));
        Controls->Add(closeButton);
    }


    void OnCloseButtonClick(Object *sender, EventArgs *e)
    {
        Close();
    }

    void OnFileExit(Object *sender, EventArgs *e)
    {
        Close();
    }

};

void main()
{
    // ds
    // This line creates an instance of WinForm, and
```

```
    // uses it as the Main Window of the application.
    Application::Run(new WinForm());
}
```

以上列出的代码有些微的变动。默认情况下，向导产生的“ todo: ”注释盖住了向导所产生的 Close 按钮。所以，上面的代码在显示这些注释时将它们变小了一些。稍后我们会细致地看一看 Form 类。

图 33.1 显示了正在运行中的 Ex33a Windows Forms 应用程序。

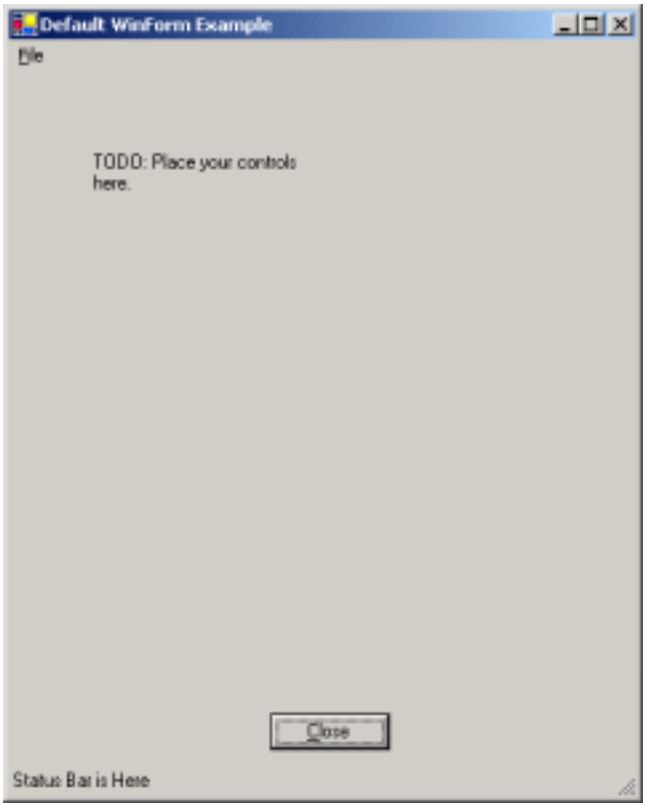

图 33.1　正在运行中的 Ex33a 示例程序

33.1.4　Form 类

Windows Forms 应用程序的基础是一个从公共语言运行时库的 Form 类派生出来的类。就如同 MFC 用一个 C++类库来隐藏了必要的管理 Windows 应用程序的细节一样，公共语言运行时库中的类也隐藏了同样的细节。这意味着您不需要定义 WndProc 函数，不需要注册窗口类和运行消息循环。

请注意 Ex33a 的 Source.cpp 代码清单最前面的#using 编译指令。这条指令引入了公共语言运行时库(公共语言运行时库位于 mscorlib.dll 中)。namespace 语句使得编写代码更加方便——它们消除了每一个变量前面的作用域修饰符。

因为这个例子是用托管的 C++编写的，所以 Form 类前面加上了__gc 声明。记住，Windows Forms 是基于公共语言运行时库的应用程序，因此要求生存在可回收的堆上。

任何标准的窗口都可以由 Form 类来表示。您只需要确保使用了正确的窗体类(例如，为了支持 MDI 应用程序，有一个 MdiClient 类)，并且设置了它的属性，使得它看起来跟您期望的外观一致。一般而言，这比使用原始的 Windows SDK 或者(甚至)MFC 要简单得多。您可以在设计时，通过 Visual Studio .NET 的 Properties 窗口来管理所有这些窗体属性，或者您也可以通过编程的方式在运行时管理这些属性。

对于使用公共语言运行时库的开发人员而言，Windows Forms 应用程序具有 Visual Basic 风格的开发简便性。最后，我们将看到应用程序之间的更多一致性，因为开发人员将使用同样的框架。这意味着，在 MFC 应用程序、Visual Basic 应用程序和 SDK 应用程序之间的差异性将不再存在。

33.1.5　处理事件

Windows 是一个事件驱动的操作系统，因此，任何一个 Windows 用户界面程序的主要目标是处理各种 Windows 事件。我们曾经编写 MFC 代码来处理各种类型的事件，包括鼠标移动、鼠标按钮被按下、键盘被按下等事件。Windows Forms 处理绝大多数事件的方式是，对于该程序将要处理的每一个事件，都插入一个事件控制函数。请注意，在前面列出的 Ex33a Source.cpp 程序代码中，它处理了由 File|Exit 菜单命令和 Close 按钮控件所产生的事件(Close 按钮控件是程序在运行过程中动态创建的)。相反，请回忆一下，MFC 应用程序是如何处理事件的呢？它们利用消息映射表，通过一套命令控制机制来处理事件。Windows Forms 对于命令和 Windows 消息使用了同样的事件机制：Form 类截获这些事件，如果存在一个控制函数是专门针对某个特定的事件的，则窗体就启动该控制函数的执行流程。

33.1.6　在窗体中绘制

任何一个 Windows 编程框架都要求您在屏幕上进行绘制操作。Form 类定义了一个名为 OnPaint 的事件，它跟踪 WM_PAINT 消息。Form 类截获了 Paint 事件，您可以加入一个控制函数以便在窗体中进行绘制操作。一般来说，在一个 Windows Form 中进行绘图比使用原始的 GDI 来进行绘制要简单多了。绘制操作被封装在一个 Graphics 对象中，该对象作为参数被传递给 OnPaint。

前面列出的 Ex33a 示例应用程序只是简单地在窗体中放入了一个标签(Label)控件和一个按钮(Button)控件。而在接下去的例子中我们将会看到，Windows Forms 的绘制模型是如何支持 Windows 开发人员所习惯使用的各种图形原语的。

Ex33b 示例程序：处理 Paint 事件

Ex33b 演示了 Paint 事件的处理方式。这个示例程序的核心代码是由前面提到的 Managed C

Windows Forms Wizard 产生的。下面是 Ex33b 的代码清单：

Source.cpp

```
#using <mscorlib.dll>
using namespace System;

// required dlls for WinForms
#using "System.dll"
#using "System.Windows.Forms.dll"
#using "System.Drawing.dll"

// required namespaces for WinForms
using namespace System::ComponentModel;
using namespace System::Windows::Forms;
using namespace System::Drawing;

__gc class Shape
{
public:
    Rectangle m_rect;
    Color m_PenColor;

    Shape()
    {
        m_rect.set_X(0);
        m_rect.set_Y(0);
        m_rect.set_Height(0);
        m_rect.set_Width(0);

        m_PenColor = Color::Black;
    }
    Shape(Rectangle r)
    {
        m_rect=r;
        m_PenColor = Color::Black;
    }
    virtual void Draw(System::Drawing::Graphics* g)
    {
    }
};

__gc class Line : public Shape
{
public:
    Line(Rectangle r) :
```

```
        Shape(r)
    {
        m_rect=r;
    }
    Line():
        Shape()
    {
    }
    void Draw(System::Drawing::Graphics* g)
    {
        g->DrawLine(new Pen(m_PenColor), m_rect.Left,
            m_rect.Top, m_rect.Right, m_rect.Bottom);
    }
};

__gc class Circle : public Shape
{
public:
    Circle(Rectangle r) :
        Shape(r)
    {
        m_rect=r;
    }
    Circle():
        Shape()
    {
    }
    void Draw(System::Drawing::Graphics* g)
    {
        g->DrawEllipse(new Pen(m_PenColor), m_rect.Left,
            m_rect.Top, m_rect.Right, m_rect.Bottom);
    }
};

__gc class Rect : public Shape
{
public:
    Rect(Rectangle r) :
        Shape(r)
    {
        m_rect=r;
    }
    Rect():
        Shape()
    {
```

```
    }
    void Draw(System::Drawing::Graphics* g)
    {
        g->DrawRectangle(new Pen(m_PenColor),
            m_rect.Left, m_rect.Top,
            m_rect.Right, m_rect.Bottom);
    }
};

__gc class WinForm: public Form
{
private:
    MainMenu  *mainMenu;
    MenuItem  *fileMenu;

    String    *caption;  // Caption of the WinForm
    int       width;     // width of the WinForm
    int       height;    // height of the WinForm

    Shape*     l; // line
    Shape*     c; // circle
    Shape*     r; // rectangle

    Shape*     l2; // line
    Shape*     c2; // circle
    Shape*     r2; // rectangle

public:
    WinForm()
    {
        // Set caption and size of the WinForm
        caption = "Default WinForm Example";
        width = 400;
        height = 500;

        InitForm();
    }

    void Dispose(bool disposing)
    {
        // Form is being destroyed.  Do any necessary clean-up here.
        Form::Dispose(disposing);
    }

    void CreateShapes()
```

```
{
    int x = 10;
    int y = 30;

    l = new Line(Rectangle(x, y, 30, 60));
    x = x + 50;

    c = new Circle(Rectangle(x, y, 30, 60));
    x = x + 170;

    r = new Rect(Rectangle(x, y, 60, 60));

    y = 160;
    x = 10;
    l2 = new Line(Rectangle(x, y, 30, 60));
    l2->m_PenColor = Color::Red;
    x = x + 50;

    c2 = new Circle(Rectangle(x, y, 30, 60));
    c2->m_PenColor = Color::Blue;
    x = x + 170;

    r2 = new Rect(Rectangle(x, y, 60, 60));
    r2->m_PenColor = Color::Green;
}

void DrawShapes(System::Drawing::Graphics* g)
{
    l->Draw(g);
    c->Draw(g);
    r->Draw(g);

    l2->Draw(g);
    c2->Draw(g);
    r2->Draw(g);
}

void InitForm()
{
    CreateShapes();

    // Setup controls here

    // Basic WinForm Settings
    Text = caption;
```

```
        Size = Drawing::Size(width, height);

        // Setup Menu
        mainMenu = new MainMenu();
        fileMenu = new MenuItem("&File");
        mainMenu->MenuItems->Add(fileMenu);
        fileMenu->MenuItems->Add(new MenuItem("E&xit",
            new EventHandler(this, &WinForm::OnFileExit)));
        Menu = mainMenu;

        //Paint Handler
        Paint += new PaintEventHandler(this, OnPaint);

    }

    void OnPaint(Object* sender, PaintEventArgs* e)
    {
        SolidBrush* b;
        b = new SolidBrush(Color::Black);

        e->Graphics->DrawString("Hello World",
            this->Font, b, System::Drawing::PointF(10, 10));
        DrawShapes(e->Graphics);
    }

    void OnFileExit(Object *sender, EventArgs *e)
    {
        Close();
    }

};

void main()
{
    // This line creates an instance of WinForm, and
    // uses it as the Main Window of the application.
    Application::Run(new WinForm());
}
```

图形输出

向导产生的示例代码并没有做很多图形处理方面的事情。Ex33b 包含了一些与图形显示相关的代码。Ex33b 中的图形显示代码用到了 GDI+，这是普通 GDI 的一个增强；我们前面在使用 MFC 时看到的是普通 GDI。注意，在 Source.cpp 刚开始的地方定义了三个形状对象，它们构成了一个类层次结构；这三个形状对象分别是直线、矩形和圆，它们都从 Shape 类派生。Shape 类有一些属性(一种颜色

和一个外接矩形)以及一个 Draw 方法。

Shape 类和它的子类都被定义为__gc 类，所以它们将生存在可回收的堆上。Draw 方法带一个类型为 System::Drawing::Graphics 的实参。该类型包装了 GDI 的设备环境句柄，并且会管理好各种调用，如 LineTo、Ellipse 和 Rectangle 等。

Form 类有一个名为 Paint 的事件，您可以给它附上一个控制函数。上面代码中的窗体在 InitForm 方法中附上了它的 Paint 事件控制函数。请注意，InitForm 创建了几个 Shape 派生类的实例。当 Windows 重画窗体的时候，Paint 控制函数将会处理所有这些 Shape 对象，并通过调用 Draw 方法来请每一个对象重画自身。

Draw 方法从实参中取到 Graphics 对象，并在该对象上调用 GDI+的函数，以便正确绘制出每一个形状。Line 对象使用 Graphics::DrawLine，矩形对象使用 Graphics::Draw-Rectangle，圆对象使用 Graphics::DrawEllipse。一般来说，使用 GDI+来绘制一个对象比使用 GDI 来绘制一个对象要简单很多。

图 33.2 显示了正在运行中的 Ex33b 程序。

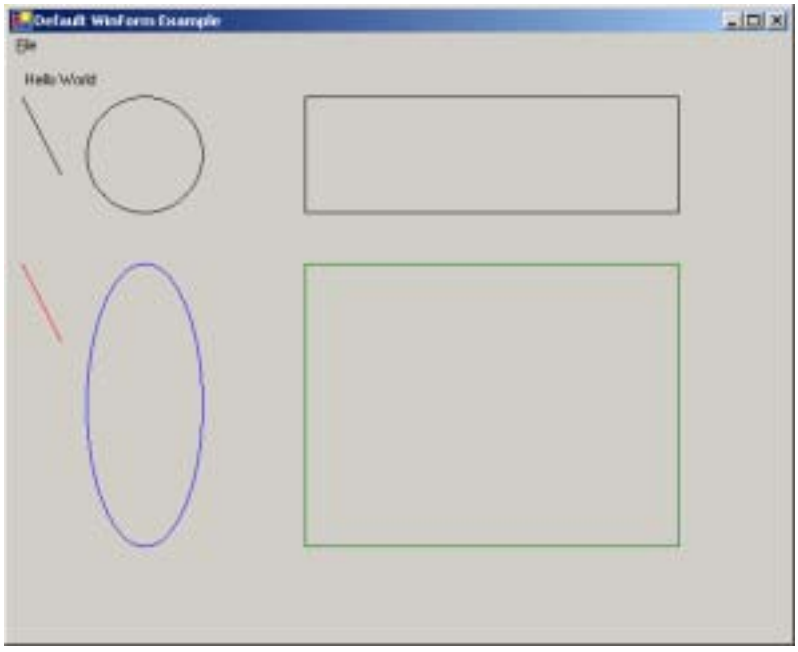

图 33.2　正在运行中的 Ex33b 示例程序

Ex33c 示例程序：一个可交互的绘图程序

为了完整地演示 Windows Forms 的工作方式，我们来看一个可以交互式地绘制形状对象的绘图程序，它所绘制的形状已经在前面列出了，其中包括直线、矩形和圆。Ex33c 只是在 Ex33b 的基础上做了轻微的变化。不过，Ex33c 处理了鼠标移动事件，并对 Graphics 对象内部的设备环境执行了一些定

制动作。

如同 Ex33a 和 Ex33b 一样，您可以用 Managed C Windows Forms Wizard 来创建 Ex33c。我删掉了“ todo: ”标签和 Close 按钮。除此之外，它是一个普通的 Windows Forms 应用程序。下面是 Ex33c 的代码清单：

Source.cpp

```
#include "stdafx.h"
#include "math.h"

#using <mscorlib.dll>
using namespace System;

// required dlls for WinForms
#using "System.dll"
#using "System.Windows.Forms.dll"
#using "System.Drawing.dll"

// required namespaces for WinForms
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Drawing;
using namespace System::Drawing::Drawing2D;

using namespace System::Diagnostics;

__value enum DrawingTypes
{
    None, Line, Circle, Rect
};
//
//
// shape hierarchy shown later…
//
//
__gc class WinForm: public Form
{
private:
    StatusBar  *statusBar;
    MainMenu  *mainMenu;
    MenuItem  *fileMenu;
    MenuItem  *drawingMenu;
    MenuItem  *circleMenu;
```

```
    MenuItem  *lineMenu;
    MenuItem  *rectMenu;
    MenuItem  *helpMenu;

    DrawingTypes drawingtype;

    ArrayList   *shapes;

    String      *caption;   // Caption of the WinForm
    int         width;     // width of the WinForm
    int         height;    // height of the WinForm

    Shape       *currentShape;

public:
    WinForm()
    {
        // Set caption and size of the WinForm
        caption = "Default WinForm Example";
        width = 600;
        height = 500;

        InitForm();
    }

    void Dispose(bool disposing)
    {
        // Form is being destroyed.  Do any
        // necessary clean-up here.
        Form::Dispose(disposing);
    }

    void InitForm()
    {
        // Setup controls here

        // Basic WinForm Settings
        this->set_BackColor(Color::White);

        Text = caption;
        Size = Drawing::Size(width, height);

        drawingtype = DrawingTypes::Line;

        // Setup Menu
```

```
        mainMenu = new MainMenu();
        fileMenu = new MenuItem("&File");
        mainMenu->MenuItems->Add(fileMenu);
        fileMenu->MenuItems->Add(
         new MenuItem("E&xit",
             new EventHandler(this, &WinForm::OnFileExit)));
        Menu = mainMenu;

        drawingMenu = new MenuItem("&Drawing");
        circleMenu =
            new MenuItem("&Circle",
                new EventHandler(this, OnDrawCircle));
        lineMenu = new MenuItem("&Line",
            new EventHandler(this, OnDrawLine));
        rectMenu =
            new MenuItem("&Rectangle",
                new EventHandler(this, OnDrawRect));
        drawingMenu->MenuItems->Add(lineMenu);
        drawingMenu->MenuItems->Add(circleMenu);
        drawingMenu->MenuItems->Add(rectMenu);
        mainMenu->MenuItems->Add(drawingMenu);

        helpMenu = new MenuItem("&Help");
        mainMenu->MenuItems->Add(helpMenu);
        helpMenu->MenuItems->Add(
            new MenuItem("&About",
                new EventHandler(this, OnHelpAbout)));

        // Set status bar
        statusBar = new StatusBar();
        statusBar->Text = "Status Bar is Here";
        Controls->Add(statusBar);

        MouseDown += new MouseEventHandler(this,
            MouseDownHandler);
        MouseMove += new MouseEventHandler(this,
            MouseMoveHandler);
        MouseUp += new MouseEventHandler(this,
            MouseUpHandler);

        Paint += new PaintEventHandler(this, OnPaint);

        shapes = new ArrayList();
        UIUpdate();
    }
```

```
void UIUpdate()
{
    // uncheck all items
    lineMenu->Checked = false;
    rectMenu->Checked = false;
    circleMenu->Checked = false;

    switch(drawingtype)
    {
    case DrawingTypes::Line:
        lineMenu->Checked = true;
        break;
    case DrawingTypes::Rect:
        rectMenu->Checked = true;
        break;
    case DrawingTypes::Circle:
        circleMenu->Checked = true;
        break;
    }
}

void OnDrawLine(Object* sender, EventArgs* e)
{
    drawingtype = DrawingTypes::Line;
    UIUpdate();
}

void OnDrawCircle(Object* sender, EventArgs* e)
{
    drawingtype = DrawingTypes::Circle;
    UIUpdate();
}

void OnDrawRect(Object* sender, EventArgs* e)
{
    drawingtype = DrawingTypes::Rect;
    UIUpdate();
}

void OnFileExit(Object *sender, EventArgs *e)
{
    Close();
}
```

```
void OnHelpAbout(Object* sender, EventArgs* e)
{
    ::MessageBox(NULL,
       "WinForms Drawing Example",
       "About WinForms Drawing Example", MB_OK);
}

void MouseDownHandler(Object* sender, MouseEventArgs* e)
{
    if(!this->Capture)
        return;

    switch(drawingtype)
    {
    case DrawingTypes::Line :
        currentShape = new Line();
        break;
    case DrawingTypes::Circle:
        currentShape = new Circle();
        break;
    case DrawingTypes::Rect:
        currentShape = new Rect();
        break;
    default:
        return;
    };

    try{
        currentShape->m_topLeft.X = e->X;
        currentShape->m_topLeft.Y = e->Y;
        currentShape->m_bottomRight.X = e->X;
        currentShape->m_bottomRight.Y = e->Y;

        this->Capture = true; // Capture the mouse
                              //  until button up
    }
    catch(Exception* ex) {
        Debug::WriteLine(ex->ToString());
    }

}


void MouseMoveHandler(Object* sender, MouseEventArgs* e)
{
```

```
    if(!this->Capture)
        return;

    try{
        Graphics* g = CreateGraphics();

        Pen *p = new Pen(this->BackColor);
        currentShape->Erase(g);

        currentShape->m_bottomRight.X = e->X;
        currentShape->m_bottomRight.Y = e->Y;

        currentShape->Draw(g);
    }
    catch (Exception* ex) {
        Debug::WriteLine(ex->ToString());
    }
}


void MouseUpHandler(Object* sender, MouseEventArgs* e)
{
    if(!currentShape)
           return;
    try{
        shapes->Add(currentShape);
        currentShape = 0;
        this->Invalidate();
        Capture = false;
        }
    catch (Exception* ex) {
        Debug::WriteLine(ex->ToString());
    }

}

void DrawShapes(System::Drawing::Graphics* g)
{
    for(int i = 0; i < shapes->Count; i++)
    {
        Shape* s = dynamic_cast<Shape*>(shapes->get_Item(i));
        s->Draw(g);
    }
}
```

```
        void OnPaint(Object* sender, PaintEventArgs* e)
        {
            Graphics* g = e->Graphics;
            DrawShapes(g);
        }


};

void main()
{
    TextWriterTraceListener * myWriter = new
        TextWriterTraceListener(System::Console::Out);
    Debug::Listeners->Add(myWriter);

    // This line creates an instance of WinForm, and
    // uses it as the main window of the application.
    Application::Run(new WinForm());
}
```

为了绘制一个形状，需要从 Drawing 菜单中选择一种形状，然后在窗体的客户区内单击并按住鼠标左键，并将鼠标拖动到一个新的位置，再释放鼠标按钮。当您拖动鼠标的时候，形状会不断地重画，或大或小。

应用程序使用了一个不同于 Ex33b 但类似的形状层次结构。Ex33c 用一个 ArrayList 来管理 Shape 对象的列表，您可以看到该 ArrayList 被声明在窗体内部。同时，您还可以看到许多 MenuItem 对象被声明并用到了。下面我们先从菜单命令的处理开始讨论。

截获命令

在 MFC 中，通过一个消息映射表，窗口的消息被映射到 C++类中的控制函数上。而 Windows Forms 的模型则使用方法代理(delegate)来暴露事件。我们将要讨论的第一种事件是命令事件(command event)，这是指来自按钮或者菜单命令的事件。

该应用程序手工建立菜单，单独加入每一个菜单命令。不幸的是，当前版本的 Visual Studio .NET 并没有包含针对 Windows Forms 和托管 C++的高层次向导工具集成，而对于 MFC 应用程序，我们已经用惯了这样的向导工具集成。每一个主菜单命令(File、Draw 和 Help)都被加入到顶层菜单结构中，然后这些主菜单中再加入相关的命令。我们只需要为每一个菜单命令提供一个字符串(将显示在菜单上)和一个指向某个方法的引用(该方法将处理菜单事件)。

该应用程序包含一个 File 菜单用于退出应用程序、一个 Drawing 菜单用于选择将要绘制的形状，以及一个 Help 菜单。Drawing 菜单设置一个内部变量，以指定当前的形状(即接下去将要被绘制的形状)。注意，Drawing 菜单命令的控制函数也在相应的菜单命令上设置了复选标记，以指明接下去要

画哪种形状(我们过去利用 MFC 的命令结构可以做到这一点)。

截获鼠标消息

除了截获命令消息，Windows Forms 应用程序往往也截获其他消息，如鼠标移动消息。Form 类暴露了一些典型的鼠标事件，如鼠标键被按下、移动鼠标、鼠标键松开。

Ex33c 处理鼠标键按下事件的做法是：捕捉(capture)鼠标，并创建当前形状类型的一个实例。一旦鼠标被应用程序捕捉住了，则所有的鼠标消息都将被送给执行了捕捉动作的窗体。Ex33c 的鼠标移动控制函数擦掉当前的形状(下面还要进一步介绍)，然后利用该控制函数的实参中的屏幕坐标，重新设置当前形状的坐标。最后，鼠标松开控制函数结束当前的形状，并将该形状对象加到内部的对象列表中。

高级图形绘制

如果您看一下 Ex33c 中有关形状层次结构的代码，您就会注意到，它与 Ex33b 中的形状层次结构有所不同。存在这种不同之处的原因在于，当您拖动鼠标时，Ex33b 的形状不会连续地重画自身。Ex33c 处理鼠标移动事件的做法是，不停地擦除当前形状，并在新的坐标上重画当前形状——对于绘图程序而言，这是一种非常合理的做法。当您松开鼠标按钮的时候，残余的线必须被清除掉。下面是来自 Ex33c 的形状层次结构，它可以完成这项清除工作：

```
__gc class Shape
{
public:
    Point m_topLeft;
    Point m_bottomRight;

    Color m_PenColor;

    Shape()
    {
        m_topLeft.X = 0;
        m_topLeft.Y = 0;
        m_bottomRight.X = 0;
        m_bottomRight.Y = 0;

        m_PenColor = Color::Black;
    }
    Shape(Point topLeft, Point bottomRight)
    {
        m_topLeft = topLeft;
        m_bottomRight = bottomRight;
        m_PenColor = Color::Black;
    }
```

```
    virtual void Draw(System::Drawing::Graphics* g)
    {
    }
    virtual void Erase(System::Drawing::Graphics* g)
    {
    }

    int SetROP(HDC hdc)
    {
        int nOldRop = ::SetROP2(hdc, R2_NOTXORPEN);
        return nOldRop;
    }

    void ResetROP(HDC hdc, int nOldRop)
    {
        ::SetROP2(hdc, nOldRop);
    }
};

__gc class Line : public Shape
{
public:
    Line(Point topLeft, Point bottomRight) :
        Shape(topLeft, bottomRight)
    {
    }
    Line():
        Shape()
    {
    }
    void Draw(System::Drawing::Graphics* g)
    {
        System:IntPtr hdc;
        hdc = g->GetHdc();

        ::MoveToEx((HDC)hdc.ToInt32(), m_topLeft.X,
            m_topLeft.Y, NULL);
        LineTo((HDC)hdc.ToInt32(), m_bottomRight.X,
            m_bottomRight.Y);
        g->ReleaseHdc(hdc);
    }

    void Erase(System::Drawing::Graphics* g)
    {
        System:IntPtr hdc;
```

```
        hdc = g->GetHdc();

        int nOldROP = SetROP((HDC)hdc.ToInt32());
        ::MoveToEx((HDC)hdc.ToInt32(), m_topLeft.X,
            m_topLeft.Y, NULL);
        LineTo((HDC)hdc.ToInt32(), m_bottomRight.X,
            m_bottomRight.Y);
        ResetROP((HDC)hdc.ToInt32(), nOldROP);
        g->ReleaseHdc(hdc);
    }
};

__gc class Circle : public Shape
{
public:
    Circle(Point topLeft, Point bottomRight) :
      Shape(topLeft, bottomRight)
    {

    }
    Circle():
        Shape()
    {
    }
    void Draw(System::Drawing::Graphics* g)
    {
        // These are absolute coordiantes, so fixup

        System:IntPtr hdc;
        hdc = g->GetHdc();

        ::Ellipse((HDC)hdc.ToInt32(),
            m_topLeft.X,
            m_topLeft.Y,
            m_bottomRight.X,
            m_bottomRight.Y);
        g->ReleaseHdc(hdc);

    }
    void Erase(System::Drawing::Graphics* g)
    {
        System:IntPtr hdc;
        hdc = g->GetHdc();

        int nOldROP = SetROP((HDC)hdc.ToInt32());
```

```
            ::Ellipse((HDC)hdc.ToInt32(),
                m_topLeft.X, m_topLeft.Y,
                m_bottomRight.X,
                m_bottomRight.Y);
            ResetROP((HDC)hdc.ToInt32(), nOldROP);
            g->ReleaseHdc(hdc);
        }
    };
    __gc class Rect : public Shape
    {
    public:
        Rect(Point topLeft, Point bottomRight) :
          Shape(topLeft, bottomRight)
        {

        }
        Rect():
            Shape()
        {
        }
        void Draw(System::Drawing::Graphics* g)
        {
            System:IntPtr hdc;
            hdc = g->GetHdc();

            ::Rectangle((HDC)hdc.ToInt32(), m_topLeft.X,
                m_topLeft.Y, m_bottomRight.X, m_bottomRight.Y);
            g->ReleaseHdc(hdc);
        }
        void Erase(System::Drawing::Graphics* g)
        {
            System:IntPtr hdc;
            hdc = g->GetHdc();

            int nOldROP = SetROP((HDC)hdc.ToInt32());
            ::Rectangle((HDC)hdc.ToInt32(), m_topLeft.X, m_topLeft.Y,
                m_bottomRight.X, m_bottomRight.Y);
            ResetROP((HDC)hdc.ToInt32(), nOldROP);
            g->ReleaseHdc(hdc);
        }
    };
```

为了使应用程序中的橡皮线行为(rubber-banding，所谓橡皮线行为是指，当您移动鼠标时拉伸形状的效果)能够正常工作，您必须执行几个标准的 GDI 调用(在 GDI+中不能使用这些调用)。特别是，您需要调用 SetROP2 来设置二进制光栅操作。当您将一个形状拖动到其他形状上的时候，默认情况下，

Windows 只是简单地用笔(pen)来绘制(覆盖了原来的颜色)。而利用光栅操作，您可以设置设备环境，使得当您绘制新的形状的时候，可以不擦除屏幕上的(用原先的笔绘制的)当前内容。

每一个 Shape 类(直线、圆和矩形)都有一个 Erase 方法和一个 Draw 方法。Erase 方法使用 System::Drawing::Graphics 对象内部的设备环境来设置光栅操作。调用 Graphics::GetHdc 所得到的设备环境与调用 Win32 API 方法 GetDC 所得到的设备环境相同。Graphics::GetHdc 的返回结果是一个托管的系统类型(Int32Ptr)。为了得到实际的设备环境，您必须调用 ToInt32 以便得到一个整数值。然后您可以把该设备环境传递给任何一个需要设备环境的函数(例如 SetROP2 方法)。

最后，如果您看一下每种形状的 Draw 方法，您将会注意到，它们均调用标准的 Win32 API 方法来绘制线条、椭圆和矩形。混合使用 GDI+和传统的 GDI 有时候会导致不可预测的副作用。对于设置光栅操作这种情形，绘制代码并不能正确地擦除老的线条。

图 33.3 显示了正在运行中的 Ex33c 程序。

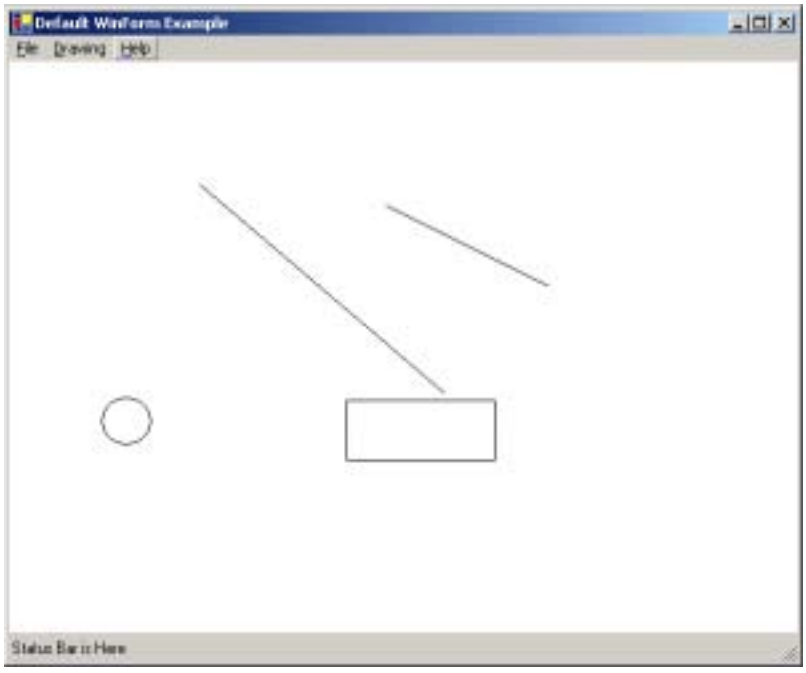

图 33.3　正在运行中的 Ex33c 示例程序

33.2　Windows Forms 的不足

Windows Forms 仍然在完善之中。虽然用于创建 Windows Forms 应用程序(它们使用公共语言运行时库)的必要工具已经有了，但是，我们作为 MFC 开发人员所习惯使用的其他一些工具却还没有。Windows Forms 提供了工具栏和状态栏支持，但是您必须手工控制它们(正如我们在本章的示例程序

中针对菜单所做的那样)。

另外缺少的一点是类似于文档/视图这样的结构。我们在 MFC 库中已经得到了这样的一种结构，但它不是公共语言运行时库的一部分。不过，编写某些文档/视图组件是非常简捷的。要想了解编写文档/视图组件的思路，您可以参考 Visual Studio .NET 的 Managed C++ Windows Forms Scribble 示例程序。

文件名：	ch33.doc
目录：	C:\WINDOWS\Desktop\desk
模板：	C:\WINDOWS\Application Data\Microsoft\Templates\技术内幕.dot
题目：	第 33 章 用托管 C++进行 Windows Forms 编程
主题：	
作者：	liyj
关键词：	
备注：	
创建日期：	2004-6-2 9:19
更改编号：	5
上次保存日期：	2004-6-2 11:38
上次保存者：	zyp
总共编辑时间：	6 分钟
上次打印时间：	2004-6-2 11:39
打印最终结果	
页数：	26
字数：	3,672 （约）
字符数：	20,934 （约）