Managed C++ and .NET Development

STEPHEN R. G. FRASER

Apress™

Managed C++ and .NET Development
Copyright © 2003 by Stephen R. G. Fraser

ISBN (pbk): 1-59059-033-3
Printed and bound in the United States of America 12345678910

Technical Reviewers: Siva Challa, Douglas Milnes

Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski

Assistant Publisher: Grace Wong

Project Managers: Tracy Brown Collins, Sofia Marchant

Copy Editor: Nicole LeClerc

Production Editor: Janet Vail

Compositor: Impressions Book and Journal Services, Inc.

Indexer: Rebecca Plunkett

Cover Designer: Kurt Krames

Artist: Cara Brunk

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 9

Basic Windows Forms Applications

Console applications are fine for quick utilities and testing functionality, but Windows applications really shine when they present a graphical user interface (GUI) to the world. With the release of Visual Studio .NET 2003, Microsoft has ushered in the era of "easy-to-build" Managed C++ windows applications. It is effortless to drag and drop your complete user interface using the built-in design tool provided by Visual Studio .NET. Adding event handling to these GUI components is a breeze as well—all it requires is a double-click at design time on the component.

The available GUI options in the .NET Framework are quite staggering, and no one chapter can do them justice. As this is the case, I have broken up the topic into two parts. In this chapter I cover the more basic areas of .NET Framework Windows GUI development, better known as *Windows Forms* (or *Win Forms*). On completing this chapter, you should have a firm background on how to develop (albeit bland) Win Forms on your own. You will have to wait for the next chapter to learn more of the bells and whistles.

In this chapter you will learn how to use the design tool, but that is not the only focus of the chapter. You will also learn how to build your Win Forms without the design tool. The reason I cover both approaches is that I feel the intimate knowledge of the Win Form components that you gain by manual development will allow you to build better interfaces. Once you know both methods, you can combine the two to create the optimal interface to your Windows application.

Win Forms Are Not MFC

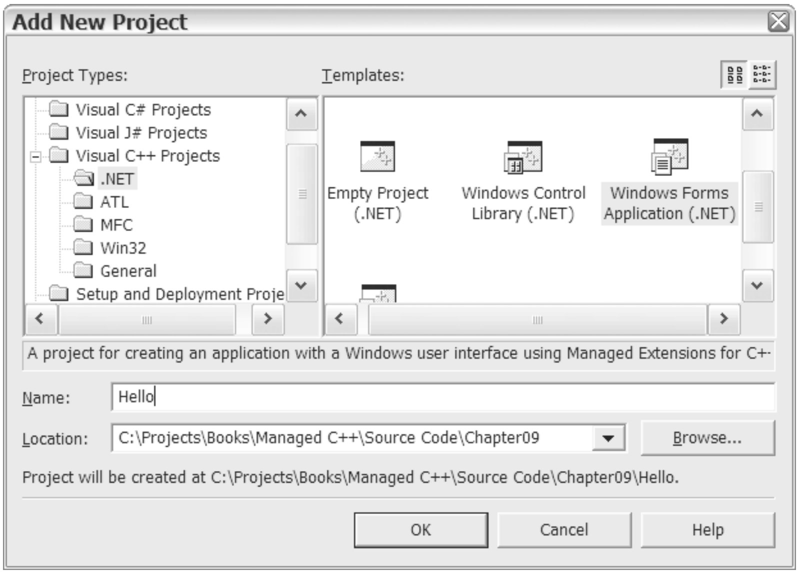
The first thing you need to know about Win Forms is that they are not an upgrade, enhancement, new version, or anything else of the Microsoft Foundation Classes (MFC). They are a brand-new, truly object-oriented Windows GUI implementation. A few classes have the same names and support the same functionalities, but that is where the similarities end.

Win Forms have a much stronger resemblance to Visual Basic's (pre-.NET) forms. In fact, Microsoft has taken the Visual Basic GUI development model of forms, controls, and properties and created a language-neutral equivalent for the .NET Framework.

When you create Windows applications with the .NET Framework, you will be working with Win Forms. It is possible to still use MFC within Visual Studio .NET, but then you are not developing .NET Framework code. However, once you have worked with Win Forms for a while, you will see that it is a much easier to code, cleaner, more object-oriented, and more complete implementation of the Windows GUI.

"Hello, World!" Win Form Style

Okay, you did the obligatory "Hello, World!" for a console application, so now you'll do it again for a Win Form application. The first thing you need to do is create a project using the Windows Forms Application (.NET) template, exactly like you did for the console application (see Figure 9-1).



A project for creating an application with a Windows user interface using Managed Extensions for C+

Name: Hello

Location: C:\Projects\Books\Managed C++\Source Code\Chapter09

Project will be created at C:\Projects\Books\Managed C++\Source Code\Chapter09\Hello.

Figure 9-1. Creating a Win Form "Hello, World!" application project

Once the project template is finished being built, you have a complete Windows application. Okay, on to the next chapter . . . Just kidding!

The process of building the "Hello, World!" application involves the following steps:

1. Expand the GUI Toolbox view.
2. Click the required GUI component in the Toolbox view.

3. Drag the component to the design form.
4. Change the component's properties in the Properties view.
5. Double-click the component to create the event handler for the component. This will bring up the IDE editor.
6. Enter the code in the IDE editor to handle the event for the component.

This is very easy and very straightforward. If this level of simplicity gives you the willies, as it did me, be comforted by the fact that you can go in and code everything by hand if you want. After a while, you will come to realize that you do not have to code much in the way of the GUI interface manually.

So what code is provided? Listing 9-1 shows Form1.cpp. It doesn't look like there's much going on, but looks can be deceiving.

Listing 9-1. The Default Form1.cpp

```
#include "stdafx.h"
#include "Form1.h"
#include <windows.h>

using namespace Hello;

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
    System::Threading::Thread::CurrentThread->ApartmentState =
       System::Threading::ApartmentState::STA;
    Application::Run(new Form1());
    return 0;
}
```

The first thing you notice is that the wizard includes the `<windows.h>` header file. The only reason I see for this is for using Windows `typedef` data types. Notice the header file is even included after the Form1.h file, which, as you'll see, contains the definition of the form. Being included after Form1.h means none of its defined types are even referenced within Form1.h.

Start up the Windows application using a standard `WinMain`. Do you notice the heavy use of Windows data types? This could also be coded as

```
int __stdcall WinMain(long hInst, long hPrevInst, long lpCmdLine, int nCmdShow)
{
}
```

Then `#include <windows.h>` could have been removed, but it doesn't hurt as it is.

The next thing the code does is initialize the current thread's apartment state. If you don't know what an *apartment state* is, don't worry about it—it's a process threading thing for COM, and this book avoids COM because it isn't managed code. In fact, the .NET Framework doesn't use apartment threads, but just to be safe, the apartment state is set to a *single-threaded apartment* (STA) in case a COM object is wrapped and used later in the application.

Finally, the program uses the `Application` class to start up `Form1`. The `Application` class is a fairly powerful class containing several static methods and properties to manage an application. The most common tasks you will use it for are starting and stopping your applications and processing Windows messages. You may also find it useful for getting information about an application via its properties.

You know what? There wasn't much there, was there? Okay, maybe all the magic is in the Form1.h file (see Listing 9-2). To access the source code of Form1.h, you need to right-click Form1.h within Solution Explorer and select the View Code menu item. You can also right-click within the form designer window.

Listing 9-2. The Default Form1.h

```
#pragma once

namespace Hello
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    public __gc class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        {
            InitializeComponent();
        }
    protected:
        void Dispose(Boolean disposing)
        {
            if (disposing && components)
            {
                components->Dispose();
            }
            __super::Dispose(disposing);
        }

    private:
        System::ComponentModel::Container * components;

        void InitializeComponent(void)
        {
            this->Size = System::Drawing::Size(300,300);
            this->Text = S"Form1";
        }
    };
}
```

Believe it or not, this is complete Win Forms code. You want to know something else? If you code this by hand, all you need is this:

```
#pragma once

namespace Hello
{
    using namespace System;
    using namespace System::Windows::Forms;

    public __gc class Form1 : public Form
    {
    public:
        Form1(void)
        {
            this->Size = Drawing::Size(300,300);
            this->Text = S"Form1";
        }
    };
}
```

All the rest of the code is for the design tool. Now this is simple! All the code does is specify the form's size and title. The rest is handled within the .NET Framework.

Okay, now for grins and giggles, change the title of the form to **Hello World**. To do this, just change the form's Text property. You can do this in a couple of ways. First, you can just type **Hello World** in the source code replacing the string Text property value **Form1**. Second, you can change the Text text box within the Properties view. Notice that if you change the property in one place, the other automatically gets updated as well.

As a thought, I guess the developers of the .NET Framework could have made things easier by calling this the Title property, but as you will soon see, the Text property is found in all Win Forms controls and is used for the default text-based property of the control.

When you finally finish staring in disbelief, go ahead and try compiling and running hello.exe. (Pressing Ctrl-F5 is the fastest way of doing this.) Rather unexciting, as you can see in Figure 9-2, but hey, what do you expect from two lines of relevant code?

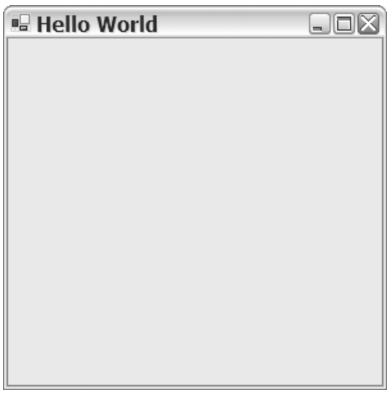


Figure 9-2. The "Hello World" form

Customizing the Form

A form by itself is not the most exciting thing, but before you move on and give it some functionality, let's look at what you'll be getting in the default form. Then let's see what else you can customize.

So what do you get for free with a form? Among many things, you get the following:

- Sized
- Minimized
- Maximized
- Moved
- Closed

It displays an icon, provides a control box, and does a lot of stuff in the background such as change the cursor when appropriate and take Windows messages and convert them into .NET events.

The `Form` is also very customizable. By manipulating a few of the `Form`'s properties you can get a completely different look from the default, along with some additional functionality that was disabled in the default form configuration. Some of the more common properties are as follows:

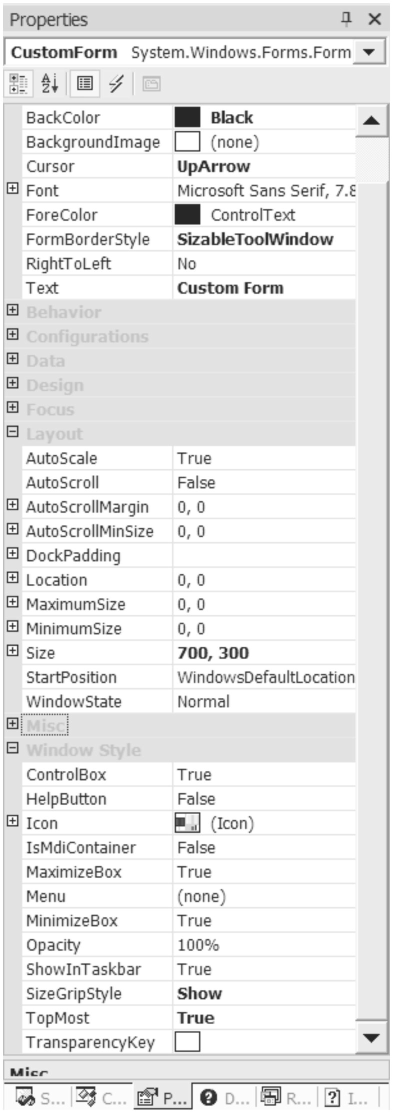
- `AutoScroll` is a `Boolean` that specifies if the form should automatically display scroll bars if sizing the window obscures a displayable area. The default value is `true`.
- `ClientSize` is a `System::Drawing::Size` that specifies the size of the client area. The *client area* is the size of the window within the border and caption bar. You use this control to adjust the size of the window to your liking or to get the dimensions of it for GDI+ drawing. You will examine GDI+ in Chapter 11.
- `Cursor` is a `Cursor` control that you use to specify the cursor to display when over the Win Form. The default is conveniently named `Cursors::Default`.
- `FormBorder` is a `FormBorderStyle` enum that specifies the style of the border. You use this control to change the look of the form. Common styles are `FixedDialog`, `FixedToolWindow`, and `SizableToolWindow`, but the style you will see most often is the default `Sizable`.
- `Icon` is a `System::Drawing::Icon` that you use to specify the icon associated with the form.

- `MaximizeBox` is a `Boolean` that specifies if the maximize button should be displayed on the caption bar. The default is `true`.
- `Menu` is a `MainMenu` control you use as the menu displayed on the form. The default is `null`, which signifies that there is no menu.
- `MinimizeBox` is a `Boolean` that specifies if the minimize button should be displayed on the caption bar. The default is `true`.
- `Size` is a `System::Drawing::Size` that specifies the size of the form. The size of the window includes the borders and caption bar. You use this control to set the size of the Win Form.
- `WindowState` is a `FormWindowState` enum that allows you to find out or specify if the Win Form is displayed as `Normal`, `Minimized`, or `Maximized`. The default window state is `FormWindowState::Normal`.

There's nothing special about working with `Form` class properties. You can either change them using the Properties view as shown in Figure 9-3 or directly in code as Listing 9-3 points out. The choice is yours. Frequently you'll start off by making general changes using the Properties window and then go into the code's `InitializeComponent()` method (which you can find in the Form1.h file for all the examples in the book) to fine-tune the changes. It doesn't really matter if you make the changes in the code or in the Properties window, as any changes you make in one will immediately be reflected in the other.

CAUTION *Be careful when you make changes within the* `InitializeComponent()` *method. The changes have to be made in exactly the same manner as the code generator or you may cause Visual Studio .NET's GUI design tool to stop functioning.*

To customize a form (or any other control, for that matter), you just assign the appropriate types and values you want to the properties and let the form handle the rest. The example in Figure 9-3 and Listing 9-3 shows a hodgepodge of different form customizations just to see what the form will look like when it's done. The biggest change happened when I changed `FormBorderStyle`.



CustomForm System.Windows.Forms.Form

BackColor	■ **Black**
BackgroundImage	□ (none)
Cursor	**UpArrow**
⊞ Font	Microsoft Sans Serif, 7.8
ForeColor	■ ControlText
FormBorderStyle	**SizableToolWindow**
RightToLeft	No
Text	**Custom Form**
⊞ Behavior	
⊞ Configurations	
⊞ Data	
⊞ Design	
⊞ Focus	
⊟ Layout	
AutoScale	True
AutoScroll	False
⊞ AutoScrollMargin	0, 0
⊞ AutoScrollMinSize	0, 0
⊞ DockPadding	
⊞ Location	0, 0
⊞ MaximumSize	0, 0
⊞ MinimumSize	0, 0
⊞ Size	**700, 300**
StartPosition	WindowsDefaultLocation
WindowState	Normal
⊞ Misc	
⊟ Window Style	
ControlBox	True
HelpButton	False
⊞ Icon	(Icon)
IsMdiContainer	False
MaximizeBox	True
Menu	(none)
MinimizeBox	True
Opacity	100%
ShowInTaskbar	True
SizeGripStyle	**Show**
TopMost	**True**
TransparencyKey	□

Misc

Figure 9-3. Customizing Form1 using the Properties view

TIP *Properties that differ from the default appear in boldface within the Properties view.*

Listing 9-3. Customizing Form1.h

```
#pragma once

namespace CustomHello
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    public __gc class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        {
            InitializeComponent();
        }
    protected:
        void Dispose(Boolean disposing)
        {
            if (disposing && components)
            {
                components->Dispose();
            }
            __super::Dispose(disposing);
        }
    private:
        System::ComponentModel::Container * components;

        void InitializeComponent(void)
        {
            this->AutoScaleBaseSize = System::Drawing::Size(6, 15);
            this->BackColor = System::Drawing::Color::Black;
            this->ClientSize = System::Drawing::Size(692, 272);
            this->Cursor = System::Windows::Forms::Cursors::UpArrow;
            this->FormBorderStyle =
                  System::Windows::Forms::FormBorderStyle::SizableToolWindow;
            this->Name = S"Form1";
            this->SizeGripStyle = System::Windows::Forms::SizeGripStyle::Show;
            this->Text = S"Custom Form";
            this->TopMost = true;

        }
    };
}
```

Running CustomHello.exe results in the display in Figure 9-4. Notice that this form is quite a bit different from the default form generated by the previous example Hello.exe. For example, this form has no control box and no minimize or maximize buttons, and in the bottom right there is a form-sizing grip and an up-arrow cursor.

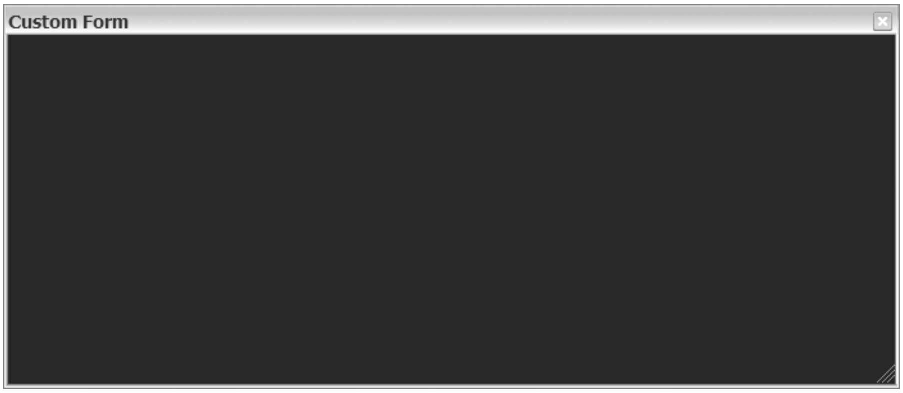


Figure 9-4. A very customized form

NOTE *For the rest of the chapter I will not list the .cpp file or repeat the constructor or dispose methods (unless something changes within them), as they are the same for every example.*

Handling Win Form Delegates and Events

Remember back in Chapter 4 when I discussed delegates and events and you thought to yourself, "That would be a great way to handle an event-driven GUI application!" You know what? You were right. This is exactly how the Win Form handles its user- and system-generated events.

Win Forms uses the .NET Framework's event model to handle all the events that take place within the form. What this requires is a delegate, an event source class, and an event receiver class. (You might want to revisit Chapter 4 if this means nothing to you.) Fortunately, all the delegates and event source classes you need to worry about are already part of the .NET Framework class library. You need to define the event receiver class.

For the following example, you'll use the `MouseDown` event that's defined in the event source class `System::Windows::Forms::Control`.

```
__event MouseEventHandler *MouseDown;
```

This event uses the `MouseEventHandler` delegate, which is defined in the `System::Windows::Forms` namespace.

```
public __gc __delegate void MouseEventHandler (
   System::Object* sender,
   System::Windows::Forms::MouseEventArgs* e
);
```

For those of you who are curious, the class `MouseEventArgs` provides five properties that you can use to figure out information about the `MouseDown` event:

- `Button`: An enum specifying which mouse button was pressed down.
- `Clicks`: The number of times the mouse was pressed and released.
- `Delta`: The number of detents the mouse wheel was rotated. A *detent* is one notch of the mouse wheel.
- `X`: The horizontal location of the mouse where it was clicked.
- `Y`: The vertical location of the mouse where it was clicked.

The first step in creating an event receiver class is to create the event handler that will handle the event generated by the event source class. So, in the case of `MouseDown`, you need to create a method with the same signature as `MouseEventHandler`. Notice also that you make the handler private. You don't want any outside method calling this event by accident, as it's only intended to be called within the event receiver class.

```
private:
    void Mouse_Clicked(System::Object * sender,
                        System::Windows::Forms::MouseEventArgs * e)
    {
    }
```

Once you have the handler, all you need to do is delegate it onto the `MouseDown` event. As you may recall from Chapter 4, Managed C++ uses multicast delegates; therefore you can chain as many handler methods as you need to complete the `MouseDown` event.

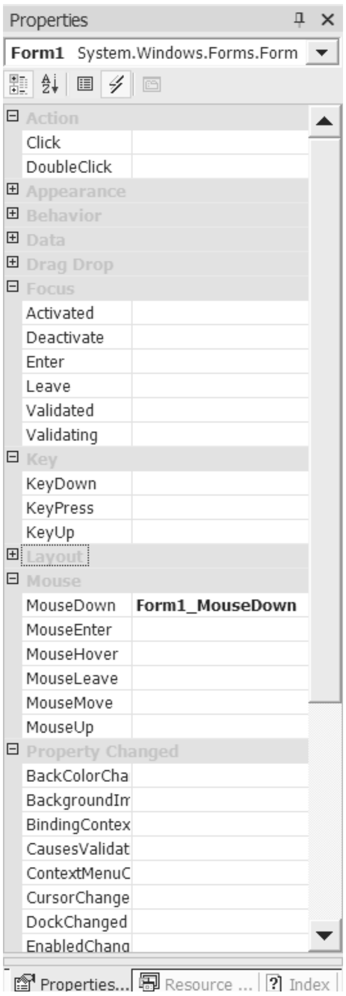
```
MouseDown += new MouseEventHandler(this, Mouse_Clicked);
```

If at a later time you no longer want this handler to handle the `MouseDown` event, all you need to do is remove the delegated method.

```
MouseDown -= new MouseEventHandler(this, Mouse_Clicked);
```

After describing all this, I'll now tell you that you can create and delegate event handlers automatically using the design tool and you don't have to worry about syntax or coding errors for the declarations. All you have to code is the functionality that handles the event. To add event handlers to a control or (in this case) a form, follow these steps:

1. In the Properties window, click the icon that looks like a lightning bolt. This will change the view from properties to events (see Figure 9-5).



MouseDown	Form1_MouseDown

Figure 9-5. Properties view of event handlers

2. Double-click the event you want to add to the control or form. This will create all the appropriate code in the form using the default name.

   or

   Enter the name of the new method in the text box next to the event handler you are creating.

   or

   If you have already written the method, select the method from the drop-down list next to the event that you want it to handle.

Listing 9-4 is a fun little program that jumps your Win Form around the screen depending on where your mouse pointer is and which mouse button you press within the client area of the form. As you can see, event handling is hardly challenging. Most of the logic of this program is just to determine where to place the form on a `MouseDown` event.

Listing 9-4. Mouse Jump: Press a Mouse Button and See the Form Jump

```
namespace MouseJump
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    public __gc class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        //...
    protected:
        void Dispose(Boolean disposing)
        //...
    private:
        System::ComponentModel::Container * components;

        void InitializeComponent(void)
        {
            this->ClientSize = System::Drawing::Size(450, 300);
            this->Name = S"Form1";
            this->Text = S"Mouse Jump";
            this->MouseDown +=
                 new System::Windows::Forms::MouseEventHandler(this,
                                                  Form1_MouseDown);
        }
    private:
        System::Void Form1_MouseDown(System::Object * sender,
                                System::Windows::Forms::MouseEventArgs * e)
        {
            // Get mouse x and y coordinates
            Int32 x = e->X;
            Int32 y = e->Y;
```

```
                // Get Forms upper left location
                Point loc = DesktopLocation;

                // Handle left button mouse click
                if (e->Button == MouseButtons::Left)
                {
                    Text = String::Format(S"Mouse Jump - Left Button at {0},{1}",
                        __box(x), __box(y));
                    DesktopLocation = Drawing::Point(loc.X + x, loc.Y +y);
                }
                // Handle right button mouse click
                else if (e->Button == MouseButtons::Right)
                {
                    Text = String::Format(S"Mouse Jump - Right Button at {0},{1}",
                        __box(x), __box(y));
                    DesktopLocation = Point((loc.X+1) - (ClientSize.Width - x),
                                            (loc.Y+1) - (ClientSize.Height - y));
                }
                // Handle middle button mouse click
                else
                {
                    Text = String::Format(S"Mouse Jump - Middle Button at {0},{1}",
                        __box(x), __box(y));
                    DesktopLocation = Point((loc.X+1) - ((ClientSize.Width/2) - x),
                                            (loc.Y+1) - ((ClientSize.Height/2) - y));
                }
            }
        };
}
```

The MouseJump.exe application shown in Figure 9-6 is hardly exciting, because you can't see the jumping of the form in a still image. You might want to notice that the coordinates at which the mouse was last clicked are displayed in the title bar.

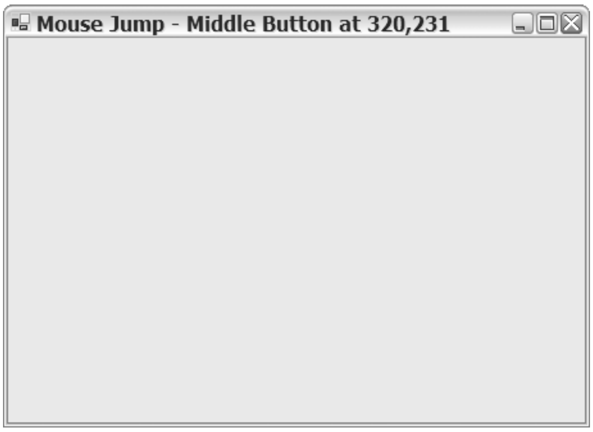


Figure 9-6. The form after a mouse jump

Adding Controls

Okay, now that you have covered the basics of a form and how to handle events from a form, you'll go ahead and make the form do something constructive. To do this, you need to add what the .NET Framework class library calls *controls*.

Controls provide you the ability to build an interface by breaking it down into smaller components. Each control provides a specific type of input and/or output functionality to your Win Form. For example, there are controls to place a label on the screen, display and input text data, select a data item from a list, and display and (if you want) update a tree of data. There is even a control to display a calendar.

All controls inherit from the `Component` and `Control` classes, with each class providing a number of standard methods and properties. Each control will have a few methods and properties of its own that make it unique. Also, all controls have events, for which you can create handlers. You can find all controls provided by the .NET Framework class library within the `System::Windows::Forms` namespace.

You can add controls to a Win Form in one of two ways, just like almost any other process when it comes to Win Forms. You can use Visual Studio .NET GUI tool to drop and drag the controls to the Win Form, or you can code the controls by hand using Visual Studio .NET's IDE editor.

Let's look at how to drag and drop controls onto a Win Form, as this is essentially what you're going to mimic when you code by hand. The steps are as follows:

1. Resize the form to the size you want by dragging the borders of the form in the design window. Make it a little bigger than you think you'll need. Don't worry—you can change the size later to enclose the controls better. I've learned from past experience that having the extra real estate makes things easier when designing.

2. Bring the cursor over the Toolbox tab (if you don't have it tacked open). This will expand the Toolbox.

3. Click, hold, and then drag the control you want from the Toolbox to the form. (If you don't have the Toolbox tacked open, you may need to drag the control to an open location on the form and release it there. This will cause the Toolbox to close so that you can click again and drag the control to the desired location on the form.)

4. Alter the properties of the controls as you wish by changing them within the Properties view. I recommend changing the `Name` property at a minimum, but there is nothing stopping you from using the default generated name for the control.

5. Add event handlers as desired. You might consider holding off on this step until you have the entire Win Form laid out.

6. Repeat steps 1 through 5 for all other required controls.

What these steps do behind the scenes is add a definition of the control to the class and then create an instance of it. Each property that is changed adds a line of code that updates one of the control's properties. Each event handler added adds a delegation statement and then creates an event handler.

As a developer, you can rely solely on the drag-and-drop functionality of Visual Studio .NET or you can do as I do and use the tool to build the basic design but then fine-tune it within the code itself. You could also be a glutton for punishment and do it all by hand. But why bother? The tool is there, so why not use it?

Okay, now that you know how to add a control to the Win Form, you'll take a look at an assortment of the more common controls provided by the .NET Framework class library, starting with one of the easiest: `Label`.

The Label Control

The name of this control is a little misleading. It gives you the impression that it is just good for displaying static text in the form. Nothing could be further from the truth. The `Label` control is also great for displaying dynamic text to the form. Heck, the `Label` control can even trigger an event when clicked.

In general, though, you'll normally use a `Label` control to statically label something else. The usual process of creating a label is simply to create the `Label` control and then set its properties so that the `Label` control looks the way you want it to. Here are some of the more common properties used by the `Label` control:

- `BackColor` is a `System::Drawing::Color` that represents the background color of the label and defaults to the `DefaultBackColor` property.
- `Font` is a `System::Drawing::Font` that represents the font used by the label and defaults to the `DefaultFont` property.
- `ForeColor` is a `System::Drawing::Color` that represents the foreground color (or the actual color of the text) of the label and defaults to the `DefaultForeColor` property.
- `Image` is a `System::Drawing::Image` that represents the image displayed within the label. The default is `null`, which signifies that no image is to be displayed.
- `ImageAlign` is a `ContentAlignment` enum that represents the alignment of the image within the label. I like to visualize the different alignments by picturing a tic-tac-toe game in my head, with each box a possible alignment. The default alignment is the center box of the tic-tac-toe game or `ContentAlignment::MiddleCenter`.
- `Text` is a `String` containing the actual text to be displayed.
- `TextAlign` is a `ContentAlignment` enum that represents the alignment of the image within the label. The default is based on the culture of the computer. Because my computer has a culture of en-us, the default alignment is the top-left corner or `ContentAlignment::TopLeft`.
- `UseMnemonic` is a `Boolean` that represents if the ampersand (&) character should be interpreted as an access-key prefix character. The default is `true`.

Now that you have seen the more common properties, for grins and giggles you'll implement a `Label` control using some of its less common properties (see Listing 9-5).

Listing 9-5. The MightyLabel, an Implementation of the Uncommon Properties

```
namespace MightyLabel
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    public __gc class Form1 : public System::Windows::Forms::Form
    {
        Boolean labelSwitch;

    public:
        Form1(void)
        {
            labelSwitch = true;
            InitializeComponent();
        }

    protected:
        void Dispose(Boolean disposing)
        //...
    private: System::Windows::Forms::Label *  MightyLabel;

    private:
        System::ComponentModel::Container * components;

        void InitializeComponent(void)
        {
            this->MightyLabel = new System::Windows::Forms::Label();
            this->SuspendLayout();
            //
            // MightyLabel
            //
            this->MightyLabel->BorderStyle =
                  System::Windows::Forms::BorderStyle::FixedSingle;
            this->MightyLabel->Cursor = System::Windows::Forms::Cursors::Hand;
            this->MightyLabel->Location = System::Drawing::Point(60, 90);
            this->MightyLabel->Name = S"MightyLabel";
            this->MightyLabel->Size = System::Drawing::Size(180, 40);
            this->MightyLabel->TabIndex = 0;
            this->MightyLabel->Text =
                 S"This is the mighty label! It will change when you click it";
            this->MightyLabel->TextAlign =
                  System::Drawing::ContentAlignment::MiddleCenter;
            this->MightyLabel->Click +=
                  new System::EventHandler(this, MightyLabel_Click);
            //
            // Form1
            //
            this->AutoScaleBaseSize = System::Drawing::Size(6, 15);
            this->ClientSize = System::Drawing::Size(300, 300);
            this->Controls->Add(this->MightyLabel);
            this->Name = S"Form1";
            this->Text = S"The Mighty Label";
            this->ResumeLayout(false);
        }
    private:
        System::Void MightyLabel_Click(System::Object *  sender,
                                       System::EventArgs *  e)
        {
            if (labelSwitch)
                MightyLabel->Text = S"Ouchie!!!  That hurt.";
            else
                MightyLabel->Text = S"Ooo!!!  That tickled.";
            labelSwitch = !labelSwitch;
        }
    };
}
```

As you can see, dragging and dropping can save you a lot of time when you're designing a form, even in such a simple case. But even this simple program shows that a programmer is still needed. A designer can drag and drop the label to where it's needed, and he or she can even change the control's properties, but a programmer is still needed to give the controls life or, in other words, to handle events.

Notice that a `Form` class is like any other Managed C++ class in that you can add your own member variables, methods, and properties. In this example, I added a `Boolean` member variable called `labelSwitch` to hold the current state of the label. I initialize it in the constructor just like I would in any other class and then use it within the `Click` event handler. Basically, as long as you don't code within the areas that the generated code says not to, you're safe to use the `Form` class as you see fit.

Figure 9-7 shows what MightyLabel.exe looks like when you execute it. Be sure to click the label a couple of times.

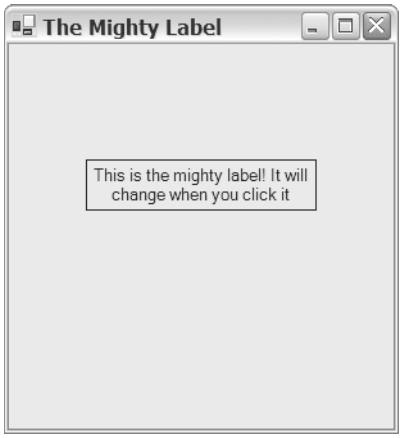


Figure 9-7. The MightyLabel example

The Button Controls

Buttons are one of the most commonly used controls for getting user input found in any Win Forms application, basically because the average user finds buttons easy to use and understand. And yet they are quite versatile for the software developer.

The .NET Framework class library provides three different types of buttons: `Button`, `CheckBox`, and `RadioButton`. All three inherit from the abstract `ButtonBase` class, which provides common functionality across all three. Here are some of the common properties provided by `ButtonBase`:

- `FlatStyle` is a `FlatStyle` enum that represents the appearance of the button. The default is `FlatStyle::Standard`, but other options are `Flat` and `Popup`.

- `Image` is a `System::Drawing::Image` that represents the image displayed on the button. The default is `null`, meaning no image is to be displayed.

- IsDefault is a protected Boolean that specifies if the button is the default for the form. In other words, it indicates if the button's Click event gets triggered when the Enter key is pressed. The default is false.

- Text is a String that represents the text that will be displayed on the button.

Remember, you also get all the properties of Control and Component. Thus, you have a plethora of properties and methods to work with.

Button

The Button control does not give much functionality beyond what is defined by abstract ButtonBase class. You might think of the Button control as the lowest level implementation of the abstract base class.

Most people think of Button as a static control that you place on the Win Form at design time. As the following example in Listing 9-6 points out (over and over again), this is not the case. Yes, you can statically place a Button control, but you can also dynamically place it on the Win Form.

Listing 9-6. The Code for "Way Too Many Buttons!"

```
namespace TooManyButtons
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    public __gc class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        //...
    protected:
        void Dispose(Boolean disposing)
        //...
    private:
        System::Windows::Forms::Button *  TooMany;
        System::ComponentModel::Container * components;
```

```
            void InitializeComponent(void)
            {
                this->TooMany = new System::Windows::Forms::Button();
                this->SuspendLayout();
                //
                // TooMany
                //
                this->TooMany->Location = System::Drawing::Point(24, 16);
                this->TooMany->Name = S"TooMany";
                this->TooMany->TabIndex = 0;
                this->TooMany->Text = S"Click Me!";
                this->TooMany->Size = System::Drawing::Size(72, 24);
                this->TooMany->Click +=
                      new System::EventHandler(this, TooMany_Click);
                //
                // Form1
                //
                this->AutoScaleBaseSize = System::Drawing::Size(6, 15);
                this->ClientSize = System::Drawing::Size(292, 270);
                this->Controls->Add(this->TooMany);
                this->Name = S"Form1";
                this->Text = S"Too Many Buttons";
                this->ResumeLayout(false);
            }
        private:
            System::Void TooMany_Click(System::Object * sender,
                                       System::EventArgs * e)
            {
                // Grab the location of the button that was clicked
                Point p = dynamic_cast<Button*>(sender)->Location;

                // Create a dynamic button
                Button *Many = new Button();
                Many->Location = Drawing::Point(p.X + 36, p.Y + 26);
                Many->Size = Drawing::Size(72, 24);
                Many->Text = S"Click Me!";
                Many->Click += new System::EventHandler(this, TooMany_Click);

                // Add dynamic button to Form
                Controls->Add(Many);
            }
        };
    }
```

There really isn't much difference between adding a `Label` control and a `Button` statically, as you can see in the `InitializeComponent()` method. The fun code in Listing 9-6 is in the `TooMany_Click()` event handler method. The first thing this method does is grab the location of the button that was clicked and place it into a `Point` struct so that you can manipulate it. You'll examine `System::Drawing::Point` in Chapter 10. You could have grabbed the whole button but you only need its location. Next, you build a button. There's nothing tricky here, except the button is declared within the event handler. Those of you from a traditional C++ background are probably jumping up and down screaming *"Memory leak!"* Sorry to disappoint you, but this is Managed C++ and the memory will be collected when it's no longer referenced, so this code is perfectly legal. And finally, the last step in placing the button dynamically on the Win Form is adding it.

Figure 9-8 shows what TooManyButtons.exe looks like when you execute it. Be sure to click a few of the newly created buttons.

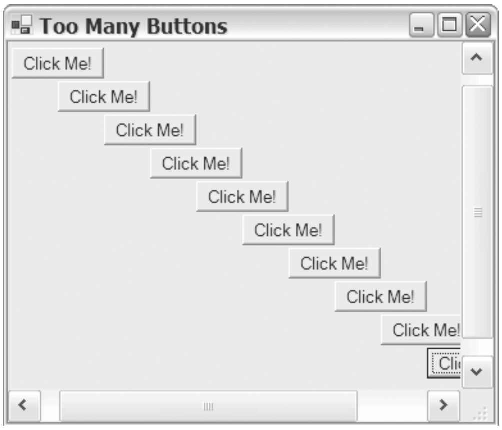


Figure 9-8. Way too many buttons

CheckBox

The `CheckBox` control is also an extension of the `ButtonBase` class. It's similar to a normal `Button` control in many ways. The two major differences are that it looks different on the Win Form and that it retains its check state when clicked. Well, the first difference isn't always true—there's a property to make a `CheckBox` look like a `Button`.

The `CheckBox` control, if configured to do so, can have three states: checked, unchecked, and indeterminate. I'm sure you understand checked and unchecked states, but what is this *indeterminate* state? Visually, in this state, the check boxes are shaded. Most likely you saw this type of check box when you installed Visual Studio .NET on your machine. Remember when you set which parts to install and some of the check marks were gray? When you selected the gray box, you found that some of the subparts were not checked. Basically, the indeterminate state of the parent was because not all child boxes were checked.

In addition to supporting the properties provided by `ButtonBase`, the `CheckBox` control also supports some properties unique to itself:

- `Appearance` is an `Appearance` enum that specifies whether the check box looks like a button or a standard check box. The default, `Appearance::Normal`, is a standard check box.

- `CheckAlign` is a `ContentAlignment` enum that represents the alignment of the check box within the `CheckBox` control. The default alignment is centered and to the left: `ContentAlignment::MiddleLeft`.

- `Checked` is a `Boolean` that represents whether the check box is checked or not. This property returns `true` if the check box is in an indeterminate state as well. The default is `false`.

- `CheckState` is a `CheckState` enum that represents the current state of the check box, either `Checked`, `Unchecked`, or `Indeterminate`. The default is `CheckState::Unchecked`.

- `ThreeState` is a `Boolean` that specifies if the check box can have an indeterminate state. The default is `false`.

In following example (see Listing 9-7) you'll have a little fun with the `CheckBox` control, in particular the `Visibility` property. Enter the following code and have some fun.

Listing 9-7. The Code for "You Can't Check Me!"

```
namespace CheckMe
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
```

```
public __gc class Form1 : public System::Windows::Forms::Form
{
public:
    Form1(void)
    //...
protected:
    void Dispose(Boolean disposing)
    //...
private: System::Windows::Forms::CheckBox *  TopCheck;
private: System::Windows::Forms::CheckBox *  checkBox1;
private: System::Windows::Forms::CheckBox *  checkBox2;
private: System::Windows::Forms::CheckBox *  BottomCheck;
private: System::ComponentModel::Container * components;

    void InitializeComponent(void)
    {
        this->TopCheck = new System::Windows::Forms::CheckBox();
        this->checkBox1 = new System::Windows::Forms::CheckBox();
        this->checkBox2 = new System::Windows::Forms::CheckBox();
        this->BottomCheck = new System::Windows::Forms::CheckBox();
        this->SuspendLayout();
        //
        // TopCheck
        //
        this->TopCheck->Location = System::Drawing::Point(50, 50);
        this->TopCheck->Name = S"TopCheck";
        this->TopCheck->Size = System::Drawing::Size(160, 25);
        this->TopCheck->TabIndex = 2;
        this->TopCheck->TabStop = false;
        this->TopCheck->Text = S"You Can\'t Check Me!";
        this->TopCheck->Enter +=
              new System::EventHandler(this, TopCheck_Entered);
        this->TopCheck->MouseEnter +=
              new System::EventHandler(this, TopCheck_Entered);
        //
        // checkBox1
        //
        this->checkBox1->Checked = true;
        this->checkBox1->CheckState =
              System::Windows::Forms::CheckState::Indeterminate;
        this->checkBox1->Location = System::Drawing::Point(50, 100);
        this->checkBox1->Name = S"checkBox1";
        this->checkBox1->Size = System::Drawing::Size(160, 25);
```

```
            this->checkBox1->TabIndex = 0;
            this->checkBox1->Text = S"Check Me! Check Me!";
            this->checkBox1->ThreeState = true;
            //
            // checkBox2
            //
            this->checkBox2->Location = System::Drawing::Point(50, 150);
            this->checkBox2->Name = S"checkBox2";
            this->checkBox2->Size = System::Drawing::Size(160, 25);
            this->checkBox2->TabIndex = 1;
            this->checkBox2->Text = S"Don\'t Forget ME!";
            //
            // BottomCheck
            //
            this->BottomCheck->Enabled = false;
            this->BottomCheck->Location = System::Drawing::Point(50, 200);
            this->BottomCheck->Name = S"BottomCheck";
            this->BottomCheck->Size = System::Drawing::Size(160, 25);
            this->BottomCheck->TabIndex = 0;
            this->BottomCheck->TabStop = false;
            this->BottomCheck->Text = S"You Can\'t Check Me!";
            this->BottomCheck->Visible = false;
            this->BottomCheck->Enter +=
                  new System::EventHandler(this, BottomCheck_Entered);
            this->BottomCheck->MouseEnter +=
                  new System::EventHandler(this, BottomCheck_Entered);
            //
            // Form1
            //
            this->AutoScaleBaseSize = System::Drawing::Size(6, 15);
            this->ClientSize = System::Drawing::Size(300, 300);
            this->Controls->Add(this->BottomCheck);
            this->Controls->Add(this->checkBox2);
            this->Controls->Add(this->checkBox1);
            this->Controls->Add(this->TopCheck);
            this->Name = S"Form1";
            this->Text = S"Can\'t Check Me";
            this->ResumeLayout(false);
        }
    private:
        System::Void TopCheck_Entered(System::Object *  sender,
                                      System::EventArgs *  e)
        {
            // Hide Top checkbox and display bottom
            TopCheck->Enabled = false;
            TopCheck->Visible = false;
            BottomCheck->Enabled = true;
            BottomCheck->Visible = true;
        }
    private:
        System::Void BottomCheck_Entered(System::Object *  sender,
                                         System::EventArgs *  e)
        {
            // Hide Bottom checkbox and display top
            BottomCheck->Enabled = false;
            BottomCheck->Visible = false;
            TopCheck->Enabled = true;
            TopCheck->Visible = true;
        }
    };
}
```

You may have noticed that I threw in the indeterminate state in the first/second/first…(whichever) check box, just so you can see what it looks like.

An important thing to take from this example is that it shows you can delegate the same event handler to more than one event. To do this in the Visual Studio .NET Properties view requires that you use the drop-down list to select the event handler that you want to redelegate.

The example also shows how to enable/disable and show/hide both in the Properties view and at runtime.

Figure 9-9 shows what CheckMe.exe looks like when you execute it. Who says programmers don't have a sense of humor!

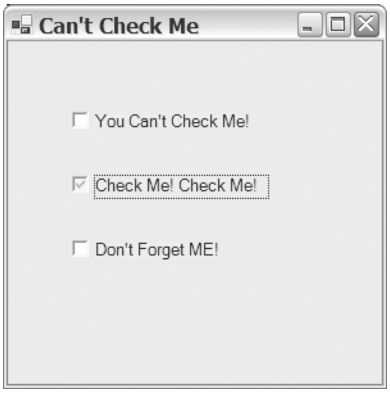


Figure 9-9. You can't check me!