12

Graphical User Interface Concepts: Part 1

Objectives

• To understand the design principles of graphical user interfaces.
• To understand, use and create event handlers.
• To understand the namespaces containing graphical user interface components and event-handling classes and interfaces.
• To create graphical user interfaces.
• To create and manipulate buttons, labels, lists, text boxes and panels.
• To be able to use mouse and keyboard events.

… the wisest prophets make sure of the event first.
Horace Walpole

…The user should feel in control of the computer; not the other way around. This is achieved in applications that embody three qualities: responsiveness, permissiveness, and consistency.

Inside Macintosh, Volume 1
Apple Computer, Inc. 1985

All the better to see you with, my dear.
The Big Bad Wolf to Little Red Riding Hood
12.1 Introduction

A graphical user interface (GUI) allows users to interact with a program visually. A GUI (pronounced “GOO-EE”) gives a program a distinctive “look” and “feel.” By providing different applications with a consistent set of intuitive user-interface components, GUIs allow users to spend less time trying to remember which keystroke sequences perform what functions and spend more time using the program in a productive manner.

**Look-and-Feel Observation 12.1**

Consistent user interfaces enable users to learn new applications faster.

As an example of a GUI, Fig. 12.1 contains an Internet Explorer window with some of its GUI components labeled. In the window, there is a menu bar containing menus, including File, Edit, View, Favorites, Tools and Help. Below the menu bar is a set of buttons; each has a defined task in Internet Explorer. Below the buttons is a text box, in which the user can type the location of a World Wide Web site to visit. To the left of the text box is a label that indicates the text box’s purpose. On the far right and bottom there are scrollbars. Scrollbars are used when there is more information in a window than can be displayed at once. By moving the scrollbars back and forth, the user can view different portions of the Web page. The menus, buttons, text boxes, labels and scrollbars are part of Internet Explorer’s GUI. They form a user-friendly interface through which the user interacts with the Internet Explorer Web browser.

GUIs are built from GUI components. A GUI component is an object with which the user interacts via the mouse or keyboard. Several common .NET GUI component classes are listed in Fig. 12.2. In the sections that follow, we discuss these GUI components in detail. In the next chapter, we discuss more advanced GUI components.
### Chapter 12  Graphical User Interface Concepts: Part 1

**Fig. 12.1** Sample Internet Explorer window with GUI components.

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>An area in which icons or uneditable text can be displayed.</td>
</tr>
<tr>
<td>TextBox</td>
<td>An area in which the user inputs data from the keyboard. The area also can display information.</td>
</tr>
<tr>
<td>Button</td>
<td>An area that triggers an event when clicked.</td>
</tr>
<tr>
<td>CheckBox</td>
<td>A GUI control that is either selected or not selected.</td>
</tr>
<tr>
<td>ComboBox</td>
<td>A drop-down list of items from which the user can make a selection, by clicking an item in the list or by typing into a box, if permitted.</td>
</tr>
<tr>
<td>ListBox</td>
<td>An area in which a list of items is displayed from which the user can make a selection by clicking once on any element. Multiple elements can be selected.</td>
</tr>
<tr>
<td>Panel</td>
<td>A container in which components can be placed.</td>
</tr>
<tr>
<td>HScrollBar</td>
<td>A horizontal scrollbar. Allows the user to access a data that cannot normally fit in its container horizontally.</td>
</tr>
<tr>
<td>VScrollBar</td>
<td>A vertical scrollbar. Allows the user to access data that cannot normally fit in its container vertically.</td>
</tr>
</tbody>
</table>

**Fig. 12.2** Some basic GUI components.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
12.2 Windows Forms

Windows Forms (also called WinForms) create GUIs for programs. A form is a graphical element that appears on the desktop. A form can be a dialog, an SDI window (single document interface window) or an MDI window (multiple document interface window, discussed in Chapter 13). A control, such as a button or label, is a component with a graphical part.

Figure 12.3 displays the Windows Forms controls and components contained in the Visual Studio .NET Toolbox—the first two screens show the controls and the last screen shows the components. When the user selects a component or control, the user then can add that component or control to the form. Note that the Pointer (the icon at the top of the list) is not a component; rather it represents the default mouse action. Highlighting it allows the programmer to use the mouse cursor instead of adding an item to the form. In this chapter and the next, we discuss many of these controls.

![Components and controls for Windows Forms.](image)

When interacting with windows, we say that the active window has the focus. The active window is the frontmost window and has a highlighted title bar. A window becomes the active window when the user clicks somewhere inside it. When a window has focus, the operating system directs user input from the keyboard and mouse to that application.
The form acts as a container for components and controls. Controls must be added to the form using code. When the user interacts with a control by using the mouse or keyboard, events (discussed in Section 12.3) are generated, and event handlers process those events. Events typically cause something to happen in response to a user interaction. For example, clicking the OK button in a MessageBox generates an event. An event handler in class MessageBox closes the MessageBox in response to this event.

Each .NET Framework class we present in this chapter is located in the System::Windows::Forms namespace. Class Form, the basic window used by Windows applications, is fully qualified as System::Windows::Forms::Form. Likewise, class Button is actually System::Windows::Forms::Button.

The general design process for creating Windows applications requires creating a Windows Form, setting its properties, adding controls, setting their properties and implementing the event handlers. Figure 12.4 lists common Form properties and events.

<table>
<thead>
<tr>
<th>Form Properties and Events</th>
<th>Description / Delegate and Event Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Common Properties</strong></td>
<td></td>
</tr>
<tr>
<td>AcceptButton</td>
<td>Specifies which button will be clicked when the user presses Enter.</td>
</tr>
<tr>
<td>AutoScroll</td>
<td>Specifies whether scrollbars appear when needed (if the data fills more than one screen).</td>
</tr>
<tr>
<td>CancelButton</td>
<td>Specifies which button is clicked when the user presses Escape.</td>
</tr>
<tr>
<td>FormBorderStyle</td>
<td>Specifies the border of the form (e.g., none, single, 3D, sizable).</td>
</tr>
<tr>
<td>Font</td>
<td>Specifies the font of text displayed on the form, as well as the default font of controls added to the form.</td>
</tr>
<tr>
<td>Text</td>
<td>Specifies the text in the form’s title bar.</td>
</tr>
<tr>
<td><strong>Common Methods</strong></td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>Closes form and releases all resources. A closed form cannot be reopened.</td>
</tr>
<tr>
<td>Hide</td>
<td>Hides form (does not release resources).</td>
</tr>
<tr>
<td>Show</td>
<td>Displays a hidden form.</td>
</tr>
<tr>
<td><strong>Common Events</strong></td>
<td>(Delegate EventHandler, event arguments EventArgs)</td>
</tr>
<tr>
<td>Load</td>
<td>Occurs before a form is shown.</td>
</tr>
</tbody>
</table>

Visual Studio .NET generates most GUI-related code when we create controls and event handlers. Programmers can use Visual Studio .NET to perform most of these tasks graphically, by dragging and dropping components onto the form and setting properties in the Properties window. In visual programming, the IDE generally maintains GUI-related code, and the programmer writes the bodies of the event handlers that specify how the application responds to user interactions.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
12.3 Event-Handling Model

GUIs are *event driven* (i.e., they generate *events* when the program’s user interacts with the GUI). Typical interactions include moving the mouse, clicking the mouse, clicking a button, typing in a text box, selecting an item from a menu and closing a window. Event handlers are methods that process events and perform tasks. For example, consider a form that changes color when a button is clicked. When clicked, the button generates an event and passes it to the event handler, and the event-handler code changes the form’s color.

Each control that can generate events has an associated delegate that defines the signature for that control’s event handlers. Recall from Chapter 10 that delegates are objects that contain pointers to methods. Event delegates are *multicast* (class `MulticastDelegate`)—they contain lists of method pointers. Each method must have the same *signature* (i.e., the same list of parameters). In the event-handling model, delegates act as intermediaries between objects that generate events and methods that handle those events (Fig. 12.5).

Software Engineering Observation 12.1

Delegates enable classes to specify methods that will not be named or implemented until the class is instantiated. This is extremely helpful in creating event handlers. For instance, the creator of the `Form` class does not need to name or define the method that will be invoked when a control is clicked. Using delegates, the class can specify when such an event handler would be called. Programmers that create their own forms then can name and define the event handlers. As long as an event handler is registered with a proper delegate, the event handler will be called at the proper time.

Once an event is raised, every method that the delegate references is called. Every method in the delegate must have the same signature, because they are all passed the same information.

12.3.1 Basic Event Handling

In most cases, we do not have to create our own events. Instead, we can just handle the events generated by .NET controls. These controls already have delegates for every event they can raise. The programmer creates the event handler and registers it with the delegate—Visual Studio .NET helps with this task. In the following example, we create a form that displays a message box when clicked. Afterwards, we will analyze the event code that Visual Studio .NET generates.

First, create a new project of type *Windows Forms Application (.NET)* and call it `SimpleEventTest`. When you create this type of application, a blank form will be created. To register and define an event handler for this form, click the `Events` icon (the yellow light-
ning bolt) and click the **Alphabetic** icon (to get listings as in Fig. 12.6) in the form’s **Properties** window (Fig. 12.6). This window allows the programmer to access, modify and create event handlers for a control. The left panel lists the events that the object can generate. The right panel lists the registered event handlers for the corresponding event; this list is initially empty. The drop-down button indicates that multiple handlers can be registered for one event. A brief description of the event appears on the bottom of the window.

In this example, the form will perform an action when clicked. Double-click the **Click** event in the **Properties** window to create an empty event handler in the program code. Visual Studio .NET inserts code of the format:

```csharp
private: System::Void FormName_Click(System::Object *sender,
                                      System::EventArgs *e)
```

This is the method that will be called when the form is clicked. As a response, we will have the form display a message box. To do this, insert the statement

```csharp
MessageBox::Show( S"Form was pressed" );
```

the body of the event handler.

We can now discuss the details of the program, which appears in Fig. 12.8. Class `Form1` (the default name chosen by Visual Studio .NET) inherits from class `Form` (which represents a form) in the .NET Framework Class Library’s `System::Windows::Form` namespace (line 26 of Fig. 12.7). A key benefit of inheriting from class `Form` is that someone else has previously defined “what it means to be a Form.” The Windows operating system expects every form (e.g., window) to have certain attributes and behaviors. How-
ever, because class `Form` already provides those capabilities, programmers do not need to “reinvent the wheel” by defining all those capabilities themselves. Class `Form` has over 400 methods! Extending class `Form` enables programmers to create forms quickly.

In this example, whenever the form is clicked, a message box appears. First, we view the event handler (lines 53–57 of Fig. 12.7) that was inserted by Visual Studio .NET. Every event handler must have the signature that the corresponding event delegate specifies. Event handlers are passed two object references. The first is a pointer to the object that raised the event (`sender`), and the second is a pointer to an `EventArgs` object (`e`). Class `EventArgs` is the base class for objects that contain event information. [Note: We split lines 53–54 over two lines for readability. However, when the event handler is generated, this code will take up only one line. The source code provided on this book’s CD does not split these generated lines of code.]

```cpp
1 // Fig. 12.7: Form1.h
2 // Creating an event handler.
3 #pragma once
4 namespace SimpleEventTest
5 {
6     using namespace System;
7     using namespace System::ComponentModel;
8     using namespace System::Collections;
9     using namespace System::Windows::Forms;
10     using namespace System::Data;
11     using namespace System::Drawing;
12     /// <summary>
13     /// Summary for Form1
14     /// WARNING: If you change the name of this class, you will need to
15     /// change the 'Resource File Name' property for the managed
16     /// resource compiler tool associated with all .resx files
17     /// this class depends on. Otherwise, the designers will not
18     /// be able to interact properly with localized resources
19     /// associated with this form.
20     /// </summary>
21     public __gc class Form1 : public System::Windows::Forms::Form
22     {
23     public:
24         Form1(void)
25         {
26             InitializeComponent();
27         }
28     protected:
29         void Dispose(Boolean disposing)
30         {
31             if (disposing && components)
32             {
33                 // Close all forms.
34                 // Dispose all types.
35                 // Release all resources.
36             }
37         }
38     }
```

Fig. 12.7 SimpleEvent demonstrates event handling. (Part 1 of 2.)

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Figure 12.8 presents function _tWinMain (lines 10–19), which is the entry point for GUI programs. The arguments on lines 10–13 can be used to specify information about the current application, such as how the form should be shown (e.g., maximized or hidden). We do not make use of these arguments in this text. Lines 15–16 (also added by the IDE) enable users of your program to interact with the Windows clipboard for cut, copy and paste operations, and allow .NET Windows applications to make use of older windows technologies such as COM (Component Object Model) and ActiveX. We do not discuss COM or ActiveX technologies in this text.

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

Figure 12.7 SimpleEvent demonstrates event handling. (Part 2 of 2.)

Figure 12.8 demonstrates event handling. (Part 1 of 2.)

```cpp
private: System::Void Form1_Click(System::Object * sender,
    System::EventArgs * e)
{
    MessageBox::Show(S"Form was pressed");
}
```
Function \_tWinMain associates a message loop with the current application, which will be handled in this application by class Form. Namespace System::Windows::Form includes class Application. Class Application contains static methods used to managed Windows applications. Method Run (line 17) displays the Windows form and starts the message loop. Windows applications communicate using messages—event descriptions. The message loop places each message on a queue and removes one at a time to forward it to the appropriate event handler. The .NET Framework hides the details of complex message processing from programmers for their convenience. Method Run also adds an event handler for the Closed event, which occurs when the user closes the window or the program calls method Close.

To create the event handler, we double-click an event name in the Properties window. This causes Visual Studio .NET to create a method with the proper signature (specified by the event's delegate). The naming convention used in the Visual Studio .NET generated code is ControlName\_EventName; in our case the event handler is Form1\_Click. \[Note: The developer can use the Properties window to determine a control's events, or can view more information about the event by looking up the event argument's class. Consult the documentation index under ControlName class (i.e., Form class) and click the events section (Fig. 12.9). This displays a list of all the events the class can generate. Click the name of an event to bring up its delegate, event argument type and a description (Fig. 12.10).\]

The format of this event-handling method is, in general,

```csharp
private: System::Void ControlName\_EventName(System::Object *sender,
                                             System::EventArgs *e)
{
    event-handling code
}
```

where the name of the event handler is by default the name of the control, followed by an underscore (_), and the name of the event. Event handlers have return type System::Void (the FCL structure equivalent to return type void). The differences between the various EventArgs classes are discussed in the following sections.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Good Programming Practice 12.1

Use the event-handler naming convention ControlName_EventName to keep methods organized. This tells a user which event a method handles, and for which control. Visual Studio .NET uses this naming convention when creating event handlers from the Properties window.

After creating the event handler, we must register it with the delegate, which contains a list of event handlers to call. Registering an event handler with a delegate involves adding
the event handler to the delegate’s invocation list. Controls have a delegate for each of their events—the delegate has the same name as the event. For example, if we are handling event EventName for object myControl, then the delegate pointer is myControl->EventName. Visual Studio .NET registers events for us with code such as the following from method InitializeComponent (which is generated automatically):

\[
\text{this->Click += new System::EventHandler( this, Form1_Click );}
\]

The first argument to the EventHandler constructor is a pointer to the object that contains the method specified as the second argument, because method Form1_Click is a method of class Form1. This parameter specifies the object that receives the method call. If Form1_Click were a static method, the first parameter to the constructor would be 0. The second argument specifies the event-handling method to call.

The lefthand side is the delegate Click. (this points to an object of class MyForm.) The delegate pointer is initially empty—we must assign to it an object pointer (the righthand side). We must create a new delegate object for each event handler. We create a new delegate object by writing

\[
\text{new System::EventHandler( this, methodName )}
\]

which returns a delegate object initialized with method methodName. The methodName is the name of the event handler, in our case it is MyForm_Click. The += operator adds an EventHandler delegate to the current delegate’s invocation list. The delegate pointer is initially empty, so registering the first event handler creates a delegate object. In general, to register an event handler, write

\[
\text{objectName->EventName += new System::EventHandler( this, MyEventHandler );}
\]

We can add more event handlers using similar statements. Event multicasting is the ability to have multiple handlers for one event. Each event handler is called when the event occurs, but the order in which the event handlers are called is indeterminate. Use the -= operator to remove an event handler from a delegate object.

**Common Programming Error 12.1**

Assuming that multiple event handlers registered for the same event are called in a particular order can lead to logic errors. If the order is important, register the first event handler and have it call the others in order, passing the sender and event arguments.

**Software Engineering Observation 12.2**

Events for prepackaged .NET components usually conform to the following naming scheme:

If the event is named EventName, its delegate is EventNameEventHandler, and the event arguments class is EventNameEventArgs. Events that use class EventArgs usually use delegate EventHandler.

To review: The information needed to register an event is the EventArgs class (a parameter for the event handler) and the EventHandler delegate (to register the event handler). Visual Studio .NET can create this code for us. For simple events and event handlers it is often easier to allow Visual Studio .NET to generate this code. For more complicated solutions, registering your own event handlers might be necessary. In the upcoming sections, we will indicate the Event-Args class and the EventHandler delegate for each event we cover. To find more information about a particular type of event, search the help documentation for ClassName class; the events will be described with the class’s other members.
12.4 Control Properties and Layout

This section overviews properties that are common to many controls. Controls derive from class *Control* (namespace `System::Windows::Forms`). Figure 12.11 contains a list of common properties and events for class *Control*. The `Text` property specifies the text that appears on a control, which may vary depending on the context. For example, the text of a Windows form appears in its title bar, and the text of a button appears on its face. The `Focus` method transfers the focus to a control. When the focus is on a control, it becomes the active control. When the `Tab` key is pressed, the `TabIndex` property determines the order in which controls are given focus. This is helpful for users with disabilities who cannot use a mouse and for the user who enters information in many different locations—the user can enter information and quickly select the next control by pressing the `Tab` key. The `Enabled` property indicates whether the control can be used. Programs can set property `Enabled` to `false` when an option is unavailable to the user. In most cases, the control’s text will appear gray (rather than black), when a control is disabled. Without having to disable a control, the control can be hidden from the user by setting the `Visible` property to `false` or by calling method `Hide`. When a control’s `Visible` property is set to `false`, the control still exists, but it is not shown on the form.

<table>
<thead>
<tr>
<th>Control Properties and Methods</th>
<th>Description</th>
</tr>
</thead>
</table>

*Common Properties*

- **BackColor**  
  Background color of the control.

- **BackgroundImage**  
  Background image of the control.

- **Enabled**  
  Specifies whether the control is enabled (i.e., if the user can interact with it). A disabled control will still be displayed, but “grayed-out”—portions of the control will become gray.

- **Focused**  
  Specifies whether the control has focus.

- **Font**  
  Font used to display control’s `Text`.

- **ForeColor**  
  Foreground color of the control. This is usually the color used to display the control’s `Text` property.

- **TabIndex**  
  Tab order of the control. When the `Tab` key is pressed, the focus is moved to controls in increasing tab order. This order can be set by the programmer if the `TabStop` property is true.

- **TabStop**  
  If `true` (the default value), user can use the `Tab` key to select the control.

- **Text**  
  Text associated with the control. The location and appearance varies with the type of control.

- **TextAlign**  
  The alignment of the text on the control. One of three horizontal positions (left, center or right) and one of three vertical positions (top, middle or bottom).

- **Visible**  
  Specifies whether the control is visible.

Fig. 12.11 *Control* class properties and methods. (Part 1 of 2.)
Visual Studio .NET allows the programmer to anchor and dock controls, which helps to specify the layout of controls inside a container (such as a form). Anchoring allows controls to stay a fixed distance from the sides of the container, even when the control is resized. Docking allows controls to extend themselves along the sides of their containers.

A user may want a control to appear in a certain position (top, bottom, left or right) in a form even when that form is resized. The user can specify this by anchoring the control to a side (top, bottom, left or right). The control then maintains a fixed distance from the side to its parent container. In most cases, the parent container is a form; however, other controls can act as a parent container.

When parent containers are resized, all controls move. Unanchored controls move relative to their original position on the form, while anchored controls move so that they will be the same distance from each side that they are anchored to. For example, in Fig. 12.12, the topmost button is anchored to the top and left sides of the parent form. When the form is resized, the anchored button moves so that it remains a constant distance from the top and left sides of the form (its parent). The unanchored button changes position as the form is resized.

Create a simple Windows Forms Application (.NET) that contains two controls (such as the two button controls in Fig. 12.12). Controls can be added to the form by double-clicking a control in the Toolbox, or selecting a control in the Toolbox and (while holding the mouse down) dragging the control onto the form. The control can be placed at a specific location on the form by dragging the control with the mouse. The control’s properties can be set using the Properties window. To display a control’s properties, select the control. If the
control’s events are currently being displayed in the **Properties** window, select the alphabetic or categorized icon.

Anchor one control to the right side by setting the **Anchor** property as shown in Fig. 12.13. Leave the other control unanchored. Now, resize the form by dragging the right side farther to the right. (You can resize many controls in this manner, or by modifying the control’s **Size** property.) Notice that both controls move. The anchored control moves so that it is always the same distance to the right wall. The unanchored control moves so that it is in the same place on the form, relative to each side. This control will continue to be somewhat closer to whatever sides it was originally close to, but will still reposition itself when the user resizes the application window.

![Click down-arrow in Anchor property to display anchoring window](image)

**Fig. 12.13** Manipulating the **Anchor** property of a control.

Sometimes a programmer wants a control to span the entire side of the form, even when the form is resized. This is useful when we want one control to remain prevalent on the form, such as the status bar that might appear at the bottom of a program. **Docking** allows a control to spread itself along an entire side (left, right, top or bottom) of its parent container. When the parent is resized, the docked control resizes as well. In Fig. 12.14, a button is docked to the top of the form. (It lays across the top portion.) When the form is resized, the button is resized as well—the button always fills the entire top portion of the form. The **Dock** dock option effectively docks the control to all sides of its parent, which causes it to fill its entire parent. Windows forms contain property **DockPadding**, which sets the distance from docked controls to the edge of the form. The default value is zero, causing the controls to attach to the edge of the form. The control layout properties are summarized in Fig. 12.15.

The docking and anchoring options refer to the parent container, which may or may not be the form. (We learn about other parent containers later this chapter.) The minimum and maximum form sizes can be set using properties **MinimumSize** and **MaximumSize**, respectively. Both properties use the **Size** structure, which has properties **Height** and **Width**, specifying the size of the form. These properties allow the programmer to design the GUI layout for a given size range. To set a form to a fixed size, set its minimum and maximum size to the same value.
Look-and-Feel Observation 12.2

Allow Windows forms to be resized—this enables users with limited screen space or multiple applications running at once to use the application more easily. Check that the GUI layout appears consistent for all permissible form sizes.

<table>
<thead>
<tr>
<th>Common Layout Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anchor</td>
<td>Side of parent container at which to anchor control—values can be combined, such as Top, Left.</td>
</tr>
<tr>
<td>Dock</td>
<td>Side of parent container to dock control—values cannot be combined.</td>
</tr>
<tr>
<td>DockPadding (for containers)</td>
<td>Sets the dock spacing for controls inside the container. Default is zero, so controls appear flush against the side of the container.</td>
</tr>
<tr>
<td>Location</td>
<td>Location of the upper left corner of the control, relative to its container.</td>
</tr>
<tr>
<td>Size</td>
<td>Size of the control. Takes a Size structure, which has properties Height and Width.</td>
</tr>
<tr>
<td>MinimumSize, MaximumSize (for Windows Forms)</td>
<td>The minimum and maximum size of the form.</td>
</tr>
</tbody>
</table>

Fig. 12.15 Control class layout properties.

12.5 Labels, TextBoxes and Buttons

Labels provide text instructions or information about the program. Labels are defined with class Label, which derives from class Control. A label displays read-only text, or text...
that the user cannot modify. Once labels are created, programs rarely change their contents. Figure 12.16 lists common Label properties.

<table>
<thead>
<tr>
<th>Label Properties</th>
<th>Description / Delegate and Event Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Properties</td>
<td></td>
</tr>
<tr>
<td>Font</td>
<td>Font used by the text on the label.</td>
</tr>
<tr>
<td>Text</td>
<td>Text to appear on the label.</td>
</tr>
<tr>
<td>TextAlign</td>
<td>Alignment of the label's text on the control. One of three horizontal positions (left, center or right) and one of three vertical positions (top, middle or bottom).</td>
</tr>
</tbody>
</table>

Fig. 12.16 Label properties.

A text box (class TextBox) is an area in which text can be either input by the user from the keyboard or displayed. The programmer can also make a text box read only. The user may read, select and copy text from a read-only text box, but the user cannot alter the text. A password text box is a text box that hides what the user entered. As the user types in characters, the password text box displays only a certain character (usually *). Altering the PasswordChar property of a text box makes it a password text box and sets the appropriate character to be displayed. Figure 12.17 lists the common properties and events of text boxes.

<table>
<thead>
<tr>
<th>TextBox Properties and Events</th>
<th>Description / Delegate and Event Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Properties</td>
<td></td>
</tr>
<tr>
<td>AcceptsReturn</td>
<td>If true, pressing Enter creates a new line in text box, if that text box spans multiple lines. If false, pressing Enter clicks the default button of the form.</td>
</tr>
<tr>
<td>Multiline</td>
<td>If true, text box can span multiple lines. Default is false.</td>
</tr>
<tr>
<td>PasswordChar</td>
<td>Single character to display instead of typed text, making the text box a password box. If no character is specified, text box displays the typed text.</td>
</tr>
<tr>
<td>ReadOnly</td>
<td>If true, text box has a gray background and its text cannot be edited. Default is false.</td>
</tr>
<tr>
<td>ScrollBars</td>
<td>For multiline text boxes, indicates which scrollbars appear (none, horizontal, vertical or both).</td>
</tr>
<tr>
<td>Text</td>
<td>Text to be displayed in the text box.</td>
</tr>
</tbody>
</table>

| Common Events                 | (Delegate EventHandler, event arguments EventArgs) |
| TextChanged                   | Raised when text changes in text box (the user added or deleted characters). |

Fig. 12.17 TextBox properties and events.
A button is a control that the user clicks to trigger a specific action. A program can use several other types of buttons, such as checkboxes and radio buttons. All the button types are derived from `ButtonBase` (namespace `System::Windows::Forms`), which defines common button features. In this section, we concentrate on class `Button`, which is often used to initiate a command. The other button types are covered in subsequent sections. The text on the face of a button is called a button label. Figure 12.18 lists the common properties and events of buttons.

<table>
<thead>
<tr>
<th>Button properties and events</th>
<th>Description / Delegate and Event Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Common Properties</strong></td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>Text displayed on the button face.</td>
</tr>
<tr>
<td><strong>Common Events</strong></td>
<td>(Delegate <code>EventHandler</code>, event arguments <code>EventArgs</code>)</td>
</tr>
<tr>
<td>Click</td>
<td>Raised when user clicks the control.</td>
</tr>
</tbody>
</table>

**Fig. 12.18** Button properties and events.

**Look-and-Feel Observation 12.3**

Although labels, text boxes and other controls can respond to mouse-button clicks, buttons naturally convey this meaning. Use buttons (e.g., **OK**), rather than other types of controls, to initiate user actions.

The program in Fig. 12.19–Fig. 12.20 uses a text box, a button and a label. The user enters text into a password box and clicks the button. The text then appears in the label. Normally, we would not display this text—the purpose of password text boxes is to hide the text being entered by the user from anyone who may be looking over a person’s shoulder.

```cpp
// Fig. 12.19: Form1.h
// Using a Textbox, Label and Button to display
// the hidden text in a password field.

#pragma once

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

    /// <summary>
    /// Summary for Form1
    /// </summary>
```

**Fig. 12.19** Using a password field. (Part 1 of 3.)
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;

        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        void InitializeComponent(void)
        {
            this->displayPasswordButton =
                new System::Windows::Forms::Button();
            this->inputPasswordTextBox =
                new System::Windows::Forms::TextBox();
            this->displayPasswordLabel =
                new System::Windows::Forms::Label();
            this->SuspendLayout();
            this->displayPasswordButton->Location =
                System::Drawing::Point(96, 96);
            this->displayPasswordButton->Text =
                "Display Password Button";
            this->displayPasswordLabel->Text =
                "Enter Password:");
        }

    /// WARNING: If you change the name of this class, you will need to
    /// change the 'Resource File Name' property for the managed
    /// resource compiler tool associated with all .resx files
    /// this class depends on. Otherwise, the designers will not
    /// be able to interact properly with localized resources
    /// associated with this form.
    /// </summary>
    /// 
    /// Required designer variable.
    /// </summary>
    System::ComponentModel::Container * components;

    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    void InitializeComponent(void)
    {
        this->displayPasswordButton =
            new System::Windows::Forms::Button();
        this->inputPasswordTextBox =
            new System::Windows::Forms::TextBox();
        this->displayPasswordLabel =
            new System::Windows::Forms::Label();
        this->SuspendLayout();
        this->displayPasswordButton->Location =
            System::Drawing::Point(96, 96);
        this->displayPasswordButton->Text =
            "Display Password Button";
        this->displayPasswordLabel->Text =
            "Enter Password:");
    }

Fig. 12.19 Using a password field. (Part 2 of 3.)
```csharp
    this->displayPasswordButton->Name = S"displayPasswordButton";
    this->displayPasswordButton->Size =
        System::Drawing::Size(96, 24);
    this->displayPasswordButton->TabIndex = 1;
    this->displayPasswordButton->Text = S"Show Me";
    this->displayPasswordButton->Click += new System::EventHandler(
        this, displayPasswordButton_Click);
    
    // inputPasswordTextBox
    //
    this->inputPasswordTextBox->Location =
        System::Drawing::Point(16, 16);
    this->inputPasswordTextBox->Name = S"inputPasswordTextBox";
    this->inputPasswordTextBox->PasswordChar = '*';
    this->inputPasswordTextBox->Size =
        System::Drawing::Size(264, 20);
    this->inputPasswordTextBox->TabIndex = 0;
    this->inputPasswordTextBox->Text = S"";
    
    //
    // displayPasswordLabel
    //
    this->displayPasswordLabel->BorderStyle =
        System::Windows::Forms::BorderStyle::Fixed3D;
    this->displayPasswordLabel->Location =
        System::Drawing::Point(16, 48);
    this->displayPasswordLabel->Name = S"displayPasswordLabel";
    this->displayPasswordLabel->Size =
        System::Drawing::Size(264, 23);
    this->displayPasswordLabel->TabIndex = 2;
    
    // Form1
    //
    this->AutoScaleBaseSize = System::Drawing::Size(S", 13);
    this->ClientSize = System::Drawing::Size(296, 133);
    this->Controls->Add(this->displayPasswordLabel);
    this->Controls->Add(this->inputPasswordTextBox);
    this->Controls->Add(this->displayPasswordButton);
    this->Name = S"Form1";
    this->Text = S"LabelTextBoxButtonTest";
    this->ResumeLayout(False);

    // display user input on label
    private: System::Void displayPasswordButton_Click(
        System::Object * sender, System::EventArgs * e)
    {
        displayPasswordLabel->Text = inputPasswordTextBox->Text;
    }
```

Fig. 12.19 Using a password field. (Part 3 of 3.)
To build this application, create a new project of type **Windows Forms Application (.NET)** and call it **LabelTextBoxButtonTest**. Create the GUI by dragging the components (a button, a label and a text box) from the **Toolbox** onto the form. Once the components are positioned, use each control’s Name property to change their names in the Properties window from the default values—**textBox1**, **label1**, **button1**—to the more descriptive **displayPasswordLabel**, **inputPasswordTextBox** and **displayPasswordButton**. Change the title of the form to display **LabelTextBoxButtonTest** by setting the form’s Text property in the Properties window. Visual Studio .NET creates the code and places it inside method **InitializeComponent**.

```
// Fig. 12.20: Form1.cpp
// Displaying the hidden text in a password field.

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

using namespace LabelTextBoxButtonTest;

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;
}
```

**Fig. 12.20** Displaying the hidden text in a password field.

To build this application, create a new project of type **Windows Forms Application (.NET)** and call it **LabelTextBoxButtonTest**. Create the GUI by dragging the components (a button, a label and a text box) from the **Toolbox** onto the form. Once the components are positioned, use each control’s Name property to change their names in the Properties window from the default values—**textBox1**, **label1**, **button1**—to the more descriptive **displayPasswordLabel**, **inputPasswordTextBox** and **displayPasswordButton**. Change the title of the form to display **LabelTextBoxButtonTest** by setting the form’s Text property in the Properties window. Visual Studio .NET creates the code and places it inside method **InitializeComponent**.

Set **displayPasswordButton**’s Text property to “**Show Me**” and clear the Text of **displayPasswordLabel** and **inputPasswordTextBox** so that they are initially blank when the program runs. Set the **BorderStyle** property of **displayPasswordLabel** to **Fixed3D**, to give the **Label** a three-dimensional appearance. Notice that text boxes have their **BorderStyle** property set to **Fixed3D** by default. Set the password character by assigning the asterisk character (\*) to the **PasswordChar** property of **inputPasswordTextBox**. This property can take only one character.

Examine the code that Visual Studio .NET generates by right-clicking the design and selecting **View Code**. This is important because not every change can be made in the Properties window.

We have learned in previous chapters that Visual Studio .NET adds comments to our code. These comments appear throughout the code, such as in lines 17–26 of Fig. 12.19. In
future examples we remove some of these generated comments to make programs more concise and readable (unless they illustrate a capability we have not yet covered). The complete code for each program is located on the CD that accompanies the book.

Visual Studio .NET inserts declarations for the controls we add to the form (lines 44–46), namely, the label, text box and button. The IDE manages these declarations for us, making it easy to add and remove controls. Line 52 declares pointer components—an array to hold the components that we add. We are not using any components in this program (only controls), so the reference is null.

The constructor for our form is created for us—it calls method InitializeComponent. Method InitializeComponent creates the components and controls in the form and sets their properties. The usual “to do” comments generated by Visual Studio .NET have been removed, because there is no more code that needs to be added to the constructor. When the code contains such comments, they appear as reminders in the Task List window of Visual Studio .NET. Method Dispose cleans up allocated resources, but is not called explicitly in our programs.

Method InitializeComponent (lines 58–113) sets the properties of the controls added to the form (the text box, label and button). The code shown here is generated as you add controls to the form and set their properties. Lines 60–66 create new objects for the controls we add (a button, a text box and a label). Lines 84–85 and 89 set the Name, PasswordChar and Text properties for inputPasswordTextBox. The TabIndex property is initially set by Visual Studio .NET, but can be changed by the developer.

The comment in lines 54–57 advises us not to modify the contents of method InitializeComponent. We have altered it slightly for formatting purposes in this book, but this is not recommended. We have done this only so that the reader is able to see the important portions of the code. Visual Studio .NET examines this method to create the design view of the code. If we change this method, Visual Studio .NET might not recognize our modifications and might show the design improperly. It is important to note that the design view is based on the code, and not vice versa.

**Error-Prevention Tip 12.1** To keep the design view accurate, do not modify the code in method InitializeComponent. Make changes in the design window or property window.

The Click event is triggered when a control is clicked. We create the handler using the procedure described in Section 12.3.1. We want to respond to the Click event of displayPasswordButton, so we first display the events for displayPasswordButton in the Properties window, then double click to the right of the Click event listing. (Alternatively, we could simply have double-clicked displayPasswordButton in design view.) This creates an empty event handler named displayPasswordButton_Click (line 116). Visual Studio .NET also registers the event handler for us (lines 77–78). It adds the event handler to the Click event, using the EventHandler delegate. We must then implement the body of the event handler. Whenever displayPasswordButton is clicked, this event handler is called and displays inputPasswordTextBox's text on displayPasswordLabel. Even though inputPasswordTextBox displays all asterisks, it still retains its input text in its Text property. To show the text, we set displayPasswordLabel's Text to inputPasswordTextBox's Text (line 119). You must program this line manually. When displayPasswordButton is clicked, the Click event is trig-
Chapter 12

Graphical User Interface Concepts: Part 1

George, and the event handler displayPasswordButton_Click runs (updating displayPasswordLabel).

Visual Studio .NET generated most of the code in this program. Visual Studio .NET simplifies tasks such as creating controls, setting their properties and registering event handlers. However, we should be aware of how these tasks are accomplished—in several programs we may set properties ourselves, using code.

12.6 Group Boxes and Panels

Group boxes (class GroupBox) and panels (class Panel) arrange components on a GUI. For example, buttons related to a particular task can be placed inside a group box or panel. All these buttons move together when the group box or panel is moved.

The main difference between the two classes is that group boxes can display a caption, and panels can have scrollbars. The scrollbars allow the user to view additional controls inside the panel by scrolling the visible area. Group boxes have thin borders by default, but panels can be set to have borders by changing their BorderStyle property.

**Look-and-Feel Observation 12.4**

Panels and group boxes can contain other panels and group boxes.

**Look-and-Feel Observation 12.5**

Organize the GUI by anchoring and docking controls (of similar function) inside a group box or panel. The group box or panel then can be anchored or docked inside a form. This divides controls into functional “groups” that can be arranged easily.

To create a group box, drag it from the toolbar and place it on a form. Create new controls and place them inside the group box, causing them to become part of this class. These controls are added to the group box’s Controls property. The group box’s Text property determines its caption. The following tables list the common properties of GroupBoxes (Fig. 12.21) and Panels (Fig. 12.22).

**GroupBox Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Common Properties</strong></td>
<td></td>
</tr>
<tr>
<td>Controls</td>
<td>Controls that the group box contains.</td>
</tr>
<tr>
<td>Text</td>
<td>Text displayed on the top portion of the group box (its caption).</td>
</tr>
</tbody>
</table>

**Panel Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Common Properties</strong></td>
<td></td>
</tr>
<tr>
<td>AutoScroll</td>
<td>Specifies whether scrollbars appear when the panel is too small to hold its controls. Default is false.</td>
</tr>
</tbody>
</table>

**Fig. 12.21** GroupBox properties.

**Fig. 12.22** Panel properties. (Part 1 of 2.)
To create a panel, drag it onto the form and add components to it. To enable the scrollbars, set the Panel’s AutoScroll property to `true` in the Properties window. If the panel is resized and cannot hold its controls, scrollbars appear (Fig. 12.23). These scrollbars then can be used to view all the components in the panel (both when running and designing the form). This allows the programmer to see the GUI exactly as it appears to the client.

**Look-and-Feel Observation 12.6**

Use panels with scrollbars to avoid cluttering a GUI and to reduce the GUI’s size.

The program in Fig. 12.24–Fig. 12.25 uses a groupbox and a panel to arrange buttons. These buttons change the text on a label.

```cpp
// Fig. 12.24: Form1.h
// Using GroupBoxes and Panels to hold buttons.

#pragma once
```

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
namespace GroupBoxPanelTest
{
  using namespace System;
  using namespace System::ComponentModel;
  using namespace System::Collections;
  using namespace System::Windows::Forms;
  using namespace System::Data;
  using namespace System::Drawing;

  /// <summary>
  /// Summary for Form1
  /// </summary>
  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::Windows::Forms::Label * messageLabel;
    private: System::Windows::Forms::Button * byeButton;
    private: System::Windows::Forms::GroupBox * mainGroupBox;
    private: System::Windows::Forms::Button * hiButton;
    private: System::Windows::Forms::Panel * mainPanel;
    private: System::Windows::Forms::Button * rightButton;
    private: System::Windows::Forms::Button * leftButton;

    private:
      /// <summary>
      /// Required designer variable.
      /// </summary>
      System::ComponentModel::Container * components;
  }
}

Fig. 12.24 GroupBox and Panel used to hold buttons. (Part 2 of 3.)
// Visual Studio .NET generated GUI code

// event handlers to change messageLabel

// event handler for hi button
private: System::Void hiButton_Click(System::Object * sender, System::EventArgs * e)
{
    messageLabel->Text = S"Hi pressed";
}

// event handler for bye button
private: System::Void byeButton_Click(System::Object * sender, System::EventArgs * e)
{
    messageLabel->Text = S"Bye pressed";
}

// event handler for far left button
private: System::Void leftButton_Click(System::Object * sender, System::EventArgs * e)
{
    messageLabel->Text = S"Far left pressed";
}

// event handler for far right button
private: System::Void rightButton_Click(System::Object * sender, System::EventArgs * e)
{
    messageLabel->Text = S"Far right pressed";
}
};

Fig. 12.24  GroupBox and Panel used to hold buttons. (Part 3 of 3.)

// Fig. 12.25: Form1.cpp
// GroupBox and Panel demonstration.

#include "stdafx.h"
#include "Form1.h"
#include <windows.h>
using namespace GroupBoxPanelTest;

int APIENTRY _tWinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR    lpCmdLine,
    int       nCmdShow)
{
    System::Threading::Thread::CurrentThread->ApartmentState =
        System::Threading::ApartmentState::STA;

Fig. 12.25  GroupBox and Panel demonstration. (Part 1 of 2.)

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
The group box (named `mainGroupBox`) has two buttons, `hiButton` (labeled Hi) and `byeButton` (labeled Bye). The panel (named `mainPanel`) has two buttons as well, `leftButton` (labeled Far Left) and `rightButton` (labeled Far Right). The `mainPanel` control also has its `AutoScroll` property set to True, allowing scrollbars to appear if needed (i.e., if the contents of the panel take up more space than the panel itself). The label (named `messageLabel`) is initially blank.

The event handlers for the four buttons are located in lines 62–87 of Fig. 12.24. To create an empty Click event handler, double click the button in design mode (instead of using the Events window). We add a line in each handler to change the text of `messageLabel` (lines 65, 72, 79 and 86). Notice that in this code example, we have replaced much of the generated code with a comment (line 57). Once again we have done this to make the code more concise and readable, but we have not modified the code provided on the book’s CD.

12.7 CheckBoxes and RadioButtons

MC++ has two types of state buttons—check boxes (class `CheckBox`) and radio buttons (class `RadioButton`)—that can be in the on/off or true/false state. Classes `CheckBox` and `RadioButton` are derived from class `ButtonBase`. A group of check boxes allow the user to select combinations of choices. A radio button is different from a check box in that there are normally several radio buttons grouped together, and only one of the radio buttons in the group can be selected (true) at any time.

A checkbox is a small white square that can be blank, can contain a checkmark or can be dimmed (i.e., the checkbox’s state is indeterminate). When a checkbox is selected, a black checkmark appears in the box. There are no restrictions on how checkboxes are used: Any number may be selected at a time. The text that appears alongside a checkbox is referred to as the checkbox label. A list of common and events of class `Checkbox` appears in Fig. 12.26.
The program in Fig. 12.27–Fig. 12.28 allows the user to select a check box to change the font style of a label. One check box applies a bold style, the other an italic style. If both checkboxes are selected, the style of the font is bold and italic. When the program initially executes, neither check box is checked.

---

Fig. 12.26  CheckBox properties and events.

The program in Fig. 12.27–Fig. 12.28 allows the user to select a check box to change the font style of a label. One check box applies a bold style, the other an italic style. If both checkboxes are selected, the style of the font is bold and italic. When the program initially executes, neither check box is checked.

---

Fig. 12.27  Using CheckBoxes to toggle italic and bold styles. (Part 1 of 2.)
public __gc class Form1 : public System::Windows::Forms::Form
{
    Form1(void)
    {
        InitializeComponent();
    }

    protected:
    void Dispose(Boolean disposing)
    {
        if (disposing && components)
        {
            components->Dispose();
        }
        __super::Dispose(disposing);
    }

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

// Visual Studio .NET generated GUI code

// make text bold if not bold, if already bold make not bold
private: System::Void boldCheckBox_CheckedChanged(System::Object * sender, System::EventArgs * e)
{
    outputLabel->Font = new Drawing::Font(
        outputLabel->Font->Name, outputLabel->Font->Size,
        static_cast< FontStyle >(outputLabel->Font->Style ^ FontStyle::Bold ));
}

// make text italic if not italic, if already italic make not italic
private: System::Void italicCheckBox_CheckedChanged(System::Object * sender, System::EventArgs * e)
{
    outputLabel->Font = new Drawing::Font(
        outputLabel->Font->Name, outputLabel->Font->Size,
        static_cast< FontStyle >(outputLabel->Font->Style ^ FontStyle::Italic ));
}
The first check box, named **boldCheckBox**, has its **Text** property set to **Bold**. The other check box is named **italicCheckBox** and has its **Text** property set to **Italic**. The label, named **outputLabel**, is labeled **Watch the font style change**.

After creating the components, we define their event handlers. To understand the code added to the event handler, we first discuss **outputLabel**’s **Font** property. To change the font, the **Font** property must be set to a **Font** object. The **Font** constructor we use takes the font name, size and style. The first two arguments make use of **outputLabel**’s **Font** object, namely, **outputLabel->Font->Name** and **outputLabel->Font->Size** (line 60 of Fig. 12.27). The third argument specifies the font style. The style is a member of the **FontStyle** enumeration, which contains the font styles **Regular**, **Bold**, **Italic**, **Strikeout** and **Underline**. (The **Strikeout** style displays text with a line through it, the **Underline** style displays text with a line below it.) A **Font** object’s **Style** property is set when the **Font** object is created—the **Style** property itself is read-only.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Styles can be combined using bitwise operators, or operators that perform manipulation on bits. All data are represented on the computer as a series of 0’s and 1’s. Each 0 or 1 is called a bit. Actions are taken and data are modified using these bit values. In this program, we need to set the font style so that the text will appear bold if it was not bold originally, and vice versa. Notice that in line 62 we use the bitwise XOR operator (^) to do this. Applying this operator to two bits does the following: If exactly one of the corresponding bits is 1, set the result to 1. By using the ^ operator as we did in line 62, we are setting the bit values for bold in the same way. The operand on the right (outputLabel->Font->Style) always has bit values set to bold. The operand on the left (outputLabel->Font->Style) must not be bold for the resulting style to be bold. (Remember for XOR, if one value is set to 1, the other must be 0, or the result will not be 1.) If outputLabel->Font->Style is bold, then the resulting style will not be bold. This operator also allows us to combine the styles. For instance, if the text were originally italicized, it would now be italicized and bold, rather than just bold.

We could have explicitly tested for the current style and changed it according to what we needed. For example, in the method boldCheckBox_CheckChanged we could have tested for the regular style, made it bold, tested for the bold style, made it regular, tested for the italic style, made it bold italic, or the italic bold style and made it italic. However, this method has a drawback—for every new style we add, we double the number of combinations. To add a checkbox for underline, we would have to test for eight possible styles. To add a checkbox for strikeout as well, we would have 16 tests in each event handler. By using the bitwise XOR operator, we save ourselves from this trouble. Each new style needs only a single statement in its event handler. In addition, styles can be removed easily, removing their handler. If we tested for every condition, we would have to remove the handler and all the unnecessary test conditions in the other handlers.

Radio buttons are similar to checkboxes, because they also have two states—selected and not selected (also called deselected). However, radio buttons normally appear as a group in which only one radio button can be selected at a time. Selecting a different radio button in the group forces all other radio buttons in the group to be deselected. Radio buttons represent a set of mutually exclusive options (i.e., a set in which multiple options cannot be selected at the same time).

**Look-and-Feel Observation 12.7**

*Use radio buttons when the user should choose only one option in a group.*

**Look-and-Feel Observation 12.8**

*Use checkboxes when the user should be able to choose many options in a group.*

All radio buttons added to a container (such as a form) become part of the same group. To create new groups, radio buttons must be added to other containers such as group boxes or panels. The common properties and events of class RadioButton are listed in Fig. 12.29.

**Software Engineering Observation 12.3**

*Forms, group boxes, and panels can act as logical groups for radio buttons. The radio buttons within each group will be mutually exclusive to each other, but not to radio buttons in different groups.*
The program in Fig. 12.30–Fig. 12.31 uses radio buttons to select the options for a `MessageBox`. Users select the attributes they want then press the display button, which causes the `MessageBox` to appear. A label in the lower-left corner shows the result of the `MessageBox` (Yes, No, Cancel, etc.). The different `MessageBox` icon and button types have been displayed in tables in Chapter 8, Object-Based Programming.

### Fig. 12.29 RadioButton properties and events.

The program in Fig. 12.30–Fig. 12.31 uses radio buttons to select the options for a `MessageBox`. Users select the attributes they want then press the display button, which causes the `MessageBox` to appear. A label in the lower-left corner shows the result of the `MessageBox` (Yes, No, Cancel, etc.). The different `MessageBox` icon and button types have been displayed in tables in Chapter 8, Object-Based Programming.

### Fig. 12.30 Using RadioButtons to set message window options. (Part 1 of 4.)

```cpp
// Fig. 12.30: Form1.h
// Using RadioButtons to set message window options.

#pragma once

namespace RadioButtonTest
{

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

    /// <summary>
    /// Summary for Form1
    /// </summary>
    /// WARNING: If you change the name of this class, you will need to
    /// change the 'Resource File Name' property for the managed
    /// resource compiler tool associated with all .resx files
    /// this class depends on. Otherwise, the designers will not
    /// be able to interact properly with localized resources
    /// associated with this form.
    /// </summary>
    public __gc class Form1 : public System::Windows::Forms::Form
    {
    
    }

```
Form1(void)
{
    InitializeComponent();
}

protected:
    void Dispose(Boolean disposing)
    {
        if (disposing && components)
        {
            components->Dispose();
        }
    _super::Dispose(disposing);
}

private: System::Windows::Forms::RadioButton * yesNoButton;
private: System::Windows::Forms::RadioButton * yesNoCancelButton;
private: System::Windows::Forms::RadioButton * abortRetryIgnoreButton;
private: System::Windows::Forms::RadioButton * okCancelButton;
private: System::Windows::Forms::RadioButton * okButton;
private: System::Windows::Forms::GroupBox * iconTypeGroupBox;
private: System::Windows::Forms::RadioButton * questionButton;
private: System::Windows::Forms::RadioButton * informationButton;
private: System::Windows::Forms::RadioButton * exclamationButton;
private: System::Windows::Forms::RadioButton * errorButton;
private: System::Windows::Forms::Label * displayLabel;
private: System::Windows::Forms::Button * displayButton;
private: System::Windows::Forms::Label * promptLabel;

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

// Visual Studio .NET generated GUI code

// change button based on option chosen by sender
private: System::Void buttonType_CheckedChanged(System::Object * sender, System::EventArgs * e)
{
    if ( sender == okButton ) // display OK button
    buttonType = MessageBoxButtons::OK;

    // display OK and Cancel buttons
    else if ( sender == okCancelButton )
    buttonType = MessageBoxButtons::OKCancel;

Fig. 12.30 Using RadioButtons to set message window options. (Part 2 of 4.)
470  // display Abort, Retry and Ignore buttons
82  else if ( sender == abortRetryIgnoreButton )
83      buttonType = MessageBoxButtons::AbortRetryIgnore;
84
85  // display Yes, No and Cancel buttons
86  else if ( sender == yesNoCancelButton )
87      buttonType = MessageBoxButtons::YesNoCancel;
88
89  // display Yes and No buttons
90  else if ( sender == yesNoButton )
91      buttonType = MessageBoxButtons::YesNo;
92
93  // only one option left--display Retry and Cancel buttons
94  else
95      buttonType = MessageBoxButtons::RetryCancel;
96 } // end method buttonType_CheckedChanged
97
98  // change icon based on option chosen by sender
99  private: System::Void iconType_CheckedChanged(
100     System::Object * sender, System::EventArgs * e)
101  {
102    if ( sender == errorButton ) // display error icon
103      iconType = MessageBoxIcon::Error;
104
105    // display exclamation point
106    else if ( sender == exclamationButton )
107      iconType = MessageBoxIcon::Exclamation;
108
109    // display information icon
110    else if ( sender == informationButton )
111      iconType = MessageBoxIcon::Information;
112
113    else // only one option left--display question mark
114      iconType = MessageBoxIcon::Question;
115 } // end method iconType_CheckedChanged
116
117  // display MessageBox and button user pressed
118  private: System::Void displayButton_Click(
119     System::Object * sender, System::EventArgs * e)
120  {
121    DialogResult = MessageBox::Show(
122       S“This is Your Custom MessageBox.,”,
123       S“Custom MessageBox”, buttonType, iconType );
124
125    // check for dialog result and display it in label
126    switch ( DialogResult ) {
127    case DialogResult::OK:
128      displayLabel->Text = S“OK was pressed.”;
129      break;
130  Fig. 12.30 Using RadioButtons to set message window options. (Part 3 of 4.)
case DialogResult::Cancel:
    displayLabel->Text = S"Cancel was pressed."
    break;

case DialogResult::Abort:
    displayLabel->Text = S"Abort was pressed."
    break;

case DialogResult::Retry:
    displayLabel->Text = S"Retry was pressed."
    break;

case DialogResult::Ignore:
    displayLabel->Text = S"Ignore was pressed."
    break;

case DialogResult::Yes:
    displayLabel->Text = S"Yes was pressed."
    break;

case DialogResult::No:
    displayLabel->Text = S"No was pressed."
    break;

} // end switch

} // end method displayButton_Click

Fig. 12.30 Using RadioButtons to set message window options. (Part 4 of 4.)

// Fig. 12.31: Form1.cpp
// RadioButton demonstration.
#include "stdafx.h"
#include "Form1.h"
#include <windows.h>
using namespace RadioButtonTest;

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;

} // end _tWinMain

Fig. 12.31 RadioButton demonstration. (Part 1 of 2.)
Fig. 12.31 RadioButton demonstration. (Part 2 of 2.)
To store the user’s choice of options, the objects `iconType` and `buttonType` are created and initialized (lines 59–60 of Fig. 12.30). Object `iconType` can be assigned an enumeration `MessageBoxIcon` value—Asterisk, Error, Exclamation, Hand, Information, Question, Stop or Warning. In this example, we use only Error, Exclamation, Information and Question.

Object `buttonType` can be assigned an enumeration `MessageBoxButton` value—AbortRetryIgnore, OK, OKCancel, RetryCancel, YesNo or YesNoCancel. The name indicates which buttons will appear in the MessageBox. In this example, we use all MessageBoxButton enumeration values.

Radio buttons (lines 44–49 and 51–54 of Fig. 12.30) are created for the enumeration options, with their labels set appropriately. The radio buttons are grouped; thus, only one option can be selected from each group box. Two group boxes are created (lines 43 and 50), one for each enumeration. Their captions are **Button Type** and **Icon**. One label is used to prompt the user (`promptLabel`), while the other is used to display which button was pressed, once the custom MessageBox has been displayed (`displayLabel`). There is also a button (`displayButton`) that displays the text **Display**.

For event handling, one event handler exists for all the radio buttons in `buttonTypeGroupBox` and another for all the radio buttons in `iconTypeGroupBox`. Each radio button generates a `CheckedChanged` event when clicked. Each radio button that lists a button-type option is associated with `buttonType_CheckedChanged` (lines 71–96), while the ones that list the icon types are associated with `iconType_CheckedChanged` (lines 99–115).

Remember, to set the event handler for an event, use the events section of the **Properties** window. Create a new `CheckedChanged` event handler for one of the radio buttons in `buttonTypeGroupBox` and rename it `buttonType_CheckedChanged`. Then set the `CheckedChanged` event handlers for all the radio buttons in `buttonTypeGroupBox` to method `buttonType_CheckedChanged`. Create a second `CheckedChanged` event handler for a radio button in `iconTypeGroupBox` and rename it `iconType_CheckedChanged`. Finally, set the `CheckedChanged` event handlers for the radio buttons in `iconTypeGroupBox` to method `iconType_CheckedChanged`.

Both handlers compare the `sender` object with every radio button to determine which button was selected. Notice that each radio button is referred to by the identifier assigned to its `Name` property when it is compared to the `sender` object (e.g., line 74). Depending on the radio button selected, either `iconType` or `buttonType` changes.

Method `displayButton_Click` (lines 118–156) creates a MessageBox (lines 121–123). Some of the MessageBox options are set by `iconType` and `buttonType`. The result of the message box is a `DialogResult` enumeration value—Abort, Cancel, Ignore, No, None, OK, Retry or Yes. The switch statement (lines 126–155) tests for the result and sets `displayLabel->Text` appropriately.

### 12.8 PictureBoxes

A picture box (class `PictureBox`) displays an image. The image, set by an object of class `Image`, can be in a bitmap (.bmp), .gif, .jpg, .jpeg, icon or metafile format (e.g., .emf or .mwf). (Images and multimedia are discussed in Chapter 16, Graphics and Multimedia.) **GIF (Graphics Interchange Format)** and **JPEG (Joint Photographic Expert Group)** files are widely used image file formats.
The `Image` property of class `PictureBox` sets the `Image` object to display, and the `SizeMode` property sets how the image is displayed (Normal, StretchImage, AutoSize or CenterImage). Figure 12.32 describes the important properties and events of class `PictureBox`.

<table>
<thead>
<tr>
<th>PictureBox properties and events</th>
<th>Description / Delegate and Event Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Common Properties</strong></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td>Image to display in the picture box.</td>
</tr>
<tr>
<td>SizeMode</td>
<td>Gets value from enumeration <code>PictureBoxSizeMode</code> that controls image sizing and positioning. Values Normal (default), StretchImage, AutoSize and CenterImage. Normal puts image in top-left corner of picture box, and CenterImage puts image in middle. (Both cut off image if too large.) StretchImage resizes image to fit in picture box. AutoSize resizes picture box to hold image.</td>
</tr>
<tr>
<td><strong>Common Events</strong></td>
<td>(Delegate <code>EventHandler</code>, event arguments <code>EventArgs</code>)</td>
</tr>
<tr>
<td>Click</td>
<td>Raised when user clicks the control.</td>
</tr>
</tbody>
</table>

Figure 12.32 PictureBox properties and events.

The program in Fig. 12.33–Fig. 12.34 uses picture box `imagePictureBox` to display one of three bitmap images—`image0`, `image1` or `image2`. They are located in the directory `images` of our project folder. Whenever the `imagePictureBox` is clicked, the image changes. The label (named `promptLabel`) on the top of the form includes the instructions `Click On Picture Box to View Images`.

```csharp
// Fig. 12.33: Form1.h
// Using a PictureBox to display images.

#pragma once

namespace PictureBoxTest
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    using namespace System::IO;
}
```

Figure 12.33 Using a PictureBox to display images. (Part 1 of 2.)
/// <summary>
/// Summary for Form1
/// WARNING: If you change the name of this class, you will need to
/// change the 'Resource File Name' property for the managed
/// resource compiler tool associated with all .resx files
/// this class depends on. Otherwise, the designers will not
/// be able to interact properly with localized resources
/// associated with this form.
/// </summary>
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::Windows::Forms::Label * promptLabel;
private: System::Windows::Forms::PictureBox * imagePictureBox;
private: static int imageNum = -1;

private:
    /// <summary>
    /// Required designer variable.
    /// </summary>
    System::ComponentModel::Container * components;

    // Visual Studio .NET generated GUI code
    // change image whenever PictureBox clicked
    private: System::Void imagePictureBox_Click(System::Object * sender, System::EventArgs * e)
    {
        imageNum = ( imageNum + 1 ) % 3; // imageNum from 0 to 2
        // create Image object from file, display on PictureBox
        imagePictureBox->Image = Image::FromFile( String::Concat( Directory::GetCurrentDirectory(), S"\images\image", imageNum.ToString(), S".bmp" ) );
    }
};

Fig. 12.33 Using a PictureBox to display images. (Part 2 of 2.)
To respond to the user’s clicks, we must handle the Click event (lines 58–67 of Fig. 12.33). Inside the event handler, we use an integer (imageNum) to store an integer representing the image we want to display. We then set the Image property of pictureBox to an Image object. Image is discussed in Chapter 16, Graphics and Multimedia, but here we overview methodFromFile, which takes a pointer to a String (the path to the image file) and creates an Image object.

To find the images, we use class Directory (namespace System::IO, specified in line 15 of Fig. 12.33) method GetCurrentDirectory (line 65 of Fig. 12.33). This returns the current directory of the executable file (usually bin\Debug) as a String pointer. To access the images subdirectory, we take the current directory and append "\images" followed by "\" and the file name. We use a double slash because an escape sequence is needed to represent a single slash in a string. We use imageNum to append the proper number, so we can load either image0, image1 or image2. Integer imageNum stays between 0 and 2, due to the modulus calculation (line 61). Finally, we append ".bmp" to the filename. Thus, if we want to load image0, the string becomes "CurrentDir\images\image0.bmp", where CurrentDir is the directory of the executable.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
12.9 Mouse Event Handling

This section explains how to handle mouse events, such as clicks, presses and moves. Mouse events are generated when the mouse interacts with a control. They can be handled for any GUI control that derives from class System::Windows::Forms::Control. Mouse event information is passed using class MouseEventArgs, and the delegate to create the mouse event handlers is MouseEventHandler. Each mouse event-handling method must take an object and a MouseEventArgs object as arguments. The Click event, which we covered earlier, uses delegate EventHandler and event arguments EventArgs.

Class MouseEventArgs contains information about the mouse event, such as the x- and y-coordinates of the mouse pointer, the mouse button pressed, the number of clicks and the number of notches through which the mouse wheel turned. Note that the x- and y-coordinates of the MouseEventArgs object are relative to the control that raised the event. Point (0,0) is at the upper-left corner of the control. The various mouse events are described in Fig. 12.35.

### Mouse Events, Delegates and Event Arguments

- **MouseEnter** (Delegate EventHandler, event arguments EventArgs)
  - Raised if the mouse cursor enters the area of the control.
- **MouseLeave** (Delegate EventHandler, event arguments EventArgs)
  - Raised if the mouse cursor leaves the area of the control.

### Mouse Events (Delegate MouseEventHandler, event arguments MouseEventArgs)

- **MouseDown**
  - Raised if the mouse button (either mouse button) is pressed while its cursor is over the area of the control.
- **MouseHover**
  - Raised if the mouse cursor hovers over the area of the control.
- **MouseMove**
  - Raised if the mouse cursor is moved while in the area of the control.
- **MouseUp**
  - Raised if the mouse button (either mouse button) is released when the cursor is over the area of the control.

### Class MouseEventArgs Properties

- **Button**
  - Mouse button that was pressed (left, right, middle or none).
- **Clicks**
  - The number of times the mouse button (either mouse button) was clicked.
- **X**
  - The x-coordinate of the event, relative to the control.
- **Y**
  - The y-coordinate of the event, relative to the control.

**Fig. 12.35** Mouse events, delegates and event arguments.

The program of Fig. 12.36–Fig. 12.37 uses mouse events to draw on the form. Whenever the user drags the mouse (i.e., moves the mouse while holding down a button), the color blue-violet is drawn on the form. Figure 12.36 (line 44) declares variable shouldPaint, which determines whether we should draw on the form. We want to draw only while the mouse button is pressed down. In the event handler for event MouseDown, shouldPaint is set to true (line 58 of...
Fig. 12.36). As soon as the mouse button is released the program should stop drawing; so the MouseUp event handler sets shouldPaint to false (line 65).

```cpp
// Fig. 12.36: Form1.h
// Using the mouse to draw on a form.
#pragma once

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

    /// <summary>
    /// Summary for Form1
    /// WARNING: If you change the name of this class, you will need to
    /// change the 'Resource File Name' property for the managed
    /// resource compiler tool associated with all .resx files
    /// this class depends on. Otherwise, the designers will not
    /// be able to interact properly with localized resources
    /// associated with this form.
    /// </summary>
    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: static bool shouldPaint = false; // whether to paint

        private:
            /// <summary>
            /// Required designer variable.
            /// </summary>
            
```
System::ComponentModel::Container * components;

// Visual Studio .NET generated GUI code

// should paint after mouse button has been pressed
private: System::Void Form1_MouseDown(System::Object * sender, System::Windows::Forms::MouseEventArgs * e)
{
    shouldPaint = true;
}

// stop painting when mouse button released
private: System::Void Form1_MouseUp(System::Object * sender, System::Windows::Forms::MouseEventArgs * e)
{
    shouldPaint = false;
}

// draw circle whenever mouse button moves (and mouse is down)
private: System::Void Form1_MouseMove(System::Object * sender, System::Windows::Forms::MouseEventArgs * e)
{
    if (shouldPaint)
    {
        Graphics *graphics = CreateGraphics();
        graphics->FillEllipse(new SolidBrush(Color::BlueViolet), e->X, e->Y, 4, 4);
    } // end if
}

Fig. 12.36 Mouse event handling. (Part 2 of 2.)
Whenever the mouse moves while the button is pressed down, the MouseMove event is generated. The event will be generated repeatedly. Inside the Form1_MouseMove event handler (lines 69–77 of Fig. 12.36), the program draws only if shouldPaint is true (indicating that the mouse button is down). Line 73 creates a Graphics object for the form, which provides methods for drawing various shapes. Method FillEllipse (lines 74–75) draws a circle at every point the mouse cursor moves over (while the mouse button is pressed). The first parameter to method FillEllipse is a SolidBrush object, which determines the color of the shape drawn. We create a new SolidBrush object by passing the constructor a Color value. Structure Color contains numerous predefined color constants—we selected Color::BlueViolet (line 75). The SolidBrush fills an elliptical region, which lies inside a bounding rectangle. The bounding rectangle is specified by the x- and y-coordinates of its upper-left corner, its height and its width. These four parameters are the final four arguments to method FillEllipse. The x- and y-coordinates are the location of the mouse event: They can be taken from the mouse event arguments (e->X and e->Y). To draw a circle, we set the height and width of the bounding rectangle equal—in this case, they are each four pixels.

12.10 Keyboard Event Handling

This section explains how to handle key events. Key events are generated when keys on the keyboard are pressed and released. These events can be handled by any control that inherits from System::Windows::Forms::Control. There are three key events. The first is event KeyPress, which fires when a key representing an ASCII character is pressed (determined by KeyEventArg's property KeyChar). ASCII is a 128-character set of alphanumeric symbols. (The full listing can be found in Appendix C, ASCII Character Set.)

Using the KeyPress event, we cannot determine whether modifier keys (such as Shift, Alt and Control) were pressed. To determine such actions, handle the two remaining key events—KeyUp or KeyDown. Class KeyEventArg contains information about special modifier keys. The key’s Key enumeration value can be returned, giving information about a wide range of non-ASCII keys. Modifier keys are often used in conjunction with the mouse to select or highlight information. The delegates for the two classes are KeyPressEventHandler (event argument class KeyEventArg) and KeyEventHandler (event argument class KeyEventArg). Figure 12.38 lists important information about key events.

The program of Fig. 12.39–Fig. 12.40 demonstrates using the key event handlers to display the key that was pressed. The program’s form contains two labels. It displays the key pressed on one label and modifier information on the other. The two labels (named charLabel and keyInfoLabel) are initially empty. The KeyDown and KeyPress events convey different information; thus, the form (KeyDemo) handles them both.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Keyboard Events, Delegates and Event Arguments

**Key Events** *(Delegate KeyEventHandler, event arguments KeyEventArgs)*

- **KeyDown**
  - Raised when key is initially pushed down.

- **KeyUp**
  - Raised when key is released.

**KeyPress** (Delegate KeyPressEventHandler, event arguments KeyPressEventArgs)

- **KeyPress**
  - Raised when key is pressed. Occurs repeatedly while key is held down, at a rate specified by the operating system.

**Class KeyPressEventArgs**

- **KeyChar**
  - Returns the ASCII character for the key pressed.

- **Handled**
  - Indicates whether the KeyPress event was handled (i.e., has an event handler associated with it).

**Class KeyEventArgs**

- **Alt**
  - Indicates whether the Alt key was pressed.

- **Control**
  - Indicates whether the Control key was pressed.

- **Shift**
  - Indicates whether the Shift key was pressed.

- **Handled**
  - Indicates whether the event was handled (i.e., has an event handler associated with it).

- **KeyCode**
  - Returns the key code for the key, as a Keys enumeration. This does not include modifier key information. Used to test for a specific key.

- **KeyData**
  - Returns the key code as a Keys enumeration, combined with modifier information. Used to determine all information about the key pressed.

- **KeyValue**
  - Returns the key code as an int, rather than as a Keys enumeration. Used to obtain a numeric representation of the key pressed.

- **Modifiers**
  - Returns a Keys enumeration for any modifier keys pressed (Alt, Control and Shift). Used to determine modifier key information only.

**Fig. 12.38** Keyboard events, delegates and event arguments.

```csharp
1 // Fig. 12.39: Form1.h
2 // Displaying information about the key the user pressed.
3 #pragma once
4
5 namespace KeyDemoTest
6 {
7     using namespace System;
8     using namespace System::ComponentModel;
9     using namespace System::Collections;
10     using namespace System::Windows::Forms;
11     using namespace System::Data;
```

**Fig. 12.39** Keyboard event handling. (Part 1 of 3.)
using namespace System::Drawing;

/// <summary>
/// Summary for Form1
/// </summary>
/// WARNING: If you change the name of this class, you will need to
/// change the ‘Resource File Name’ property for the managed
/// resource compiler tool associated with all .resx files
/// this class depends on. Otherwise, the designers will not
/// be able to interact properly with localized resources
/// associated with this form.
/// </summary>
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::Windows::Forms::Label *  charLabel;
private: System::Windows::Forms::Label *  keyInfoLabel;

private:

/// <summary>
/// Required designer variable.
/// </summary>
System::ComponentModel::Container * components;

// Visual Studio .NET generated GUI code

// display the name of the pressed key
private: System::Void Form1_KeyPress(System::Object *  sender,
System::Windows::Forms::KeyPressEventArgs *  e)
{
    charLabel->Text = String::Concat( S"Key pressed: ",
    ( e->KeyChar ).ToString() );
}

// display modifier keys, key code, key data and key value
private: System::Void Form1_KeyDown(System::Object *  sender,
System::Windows::Forms::KeyEventArgs *  e)
{

Fig. 12.39 Keyboard event handling. (Part 2 of 3.)

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Fig. 12.39 Keyboard event handling. (Part 3 of 3.)

// clear labels when key released
private: System::Void Form1_KeyUp(System::Object * sender,
                                      System::Windows::Forms::KeyEventArgs * e)
{
    keyInfoLabel->Text = S"";
    charLabel->Text = S"";
}

Fig. 12.40 Keyboard event handling demonstration.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
The KeyPress event handler (lines 55–60 of Fig. 12.39) accesses the KeyChar property of the KeyPressEventArgs object. This returns the key pressed as a __wchar_t and displays it in charLabel (lines 58–59). If the key pressed was not an ASCII character, then the KeyPress event will not fire and charLabel remains empty. ASCII is a common encoding format for letters, numbers, punctuation marks and other characters. It does not support keys such as the function keys (like F1) or the modifier keys (Alt, Control and Shift).

The KeyDown event handler (lines 63–73) displays more information, all from its KeyEventArgs object. It tests for the Alt, Shift and Control keys (lines 67–69), using the Alt, Shift and Control properties, each of which returns bool. It then displays the KeyCode, KeyData and KeyValue properties.

The KeyCode property returns a Keys enumeration, which is converted to a string. The KeyCode property returns the key that was pressed, but does not provide any information about modifier keys. Thus, both a capital and a lowercase “a” are represented as the A key.

The KeyData property returns a Keys enumeration value as well, but includes data about modifier keys. Thus, if “A” is input, the KeyData shows that the A key and the Shift key were pressed. Lastly, KeyValue returns the key code for the key that was pressed as an integer. This integer is the Windows virtual key code, which provides an integer value for a wide range of keys and for mouse buttons. The Windows virtual key code is useful when testing for non-ASCII keys (such as F12).

The KeyUp event handler clears both labels when the key is released (lines 79–80). As we can see from the output, non-ASCII keys are not displayed in the upper charLabel because the KeyPress event was not generated. The KeyDown event is still raised, and keyInfoLabel displays information about the key. The Keys enumeration can be used to test for specific keys by comparing the key pressed to a specific KeyCode. The Visual Studio.NET documentation has a complete list of the Keys enumerations.

---

**Common Programming Error 12.2**

The pointer returned by __box points to a copy of the original value. Modifying the boxed value does not alter the original unboxed object.

The KeyUp event handler clears both labels when the key is released (lines 79–80). As we can see from the output, non-ASCII keys are not displayed in the upper charLabel because the KeyPress event was not generated. The KeyDown event is still raised, and keyInfoLabel displays information about the key. The Keys enumeration can be used to test for specific keys by comparing the key pressed to a specific KeyCode. The Visual Studio.NET documentation has a complete list of the Keys enumerations.

**Software Engineering Observation 12.4**

To cause a control to react when a certain key is pressed (such as Enter), handle a key event and test for the key pressed. To cause a button to be clicked when the Enter key is pressed on a form, set the form’s AcceptButton property.

**SUMMARY**

- A graphical user interface (GUI) presents a pictorial interface to a program. A GUI (pronounced “GOO-EE”) gives a program a distinctive “look” and “feel.”

---

1. More information about value types and reference types can be found in Chapter 6.
Chapter 12  Graphical User Interface Concepts: Part 1

- By providing different applications with a consistent set of intuitive user interface components, GUIs allow the user to concentrate on using programs productively.
- GUIs are built from GUI components (sometimes called controls). A GUI control is a visual object with which the user interacts via the mouse or keyboard.
- Windows Forms create GUIs. A form is a graphical element that appears on the desktop. A form can be a dialog or a window.
- A control is a graphical component, such as a button. Components that are not visible usually are referred to simply as components.
- The active window has the focus. It is the frontmost window and has a highlighted title bar.
- A form acts as a container for components.
- When the user interacts with a control, an event is generated. This event can trigger methods that respond to the user's actions.
- All forms, components and controls are classes.
- The general design process for creating Windows applications involves creating a Windows Form, setting its properties, adding controls, setting their properties and configuring event handlers.
- GUIs are event driven. When a user interaction occurs, an event is generated. The event information then is passed to event handlers.
- Events are based on the notion of delegates. Delegates act as an intermediate step between the object creating (raising) the event and the method handling it.
- In many cases, the programmer will handle events generated by prepackaged controls. In this case, all the programmer needs to do is create and register the event handler.
- The information we need to register an event is the EventArgs class (to define the event handler) and the EventHandler delegate (to register the event handler).
- Labels (class Label) display read-only text instructions or information on a GUI.
- A text box is a single-line area in which text can be entered. A password text box displays only a certain character (such as *) when text is input.
- A button is a control that the user clicks to trigger a specific action. Buttons typically respond to the Click event.
- Group boxes and panels help arrange components on a GUI. The main difference between the classes is that group boxes can display text, and panels can have scrollbars.
- Visual C++.NET has two types of state buttons—check boxes and radio buttons—that have on/off or true/false values.
- A checkbox is a small white square that can be blank or contain a checkmark.
- Use the bitwise XOR operator (^) to combine or negate a font style.
- Radio buttons (class RadioButton) have two states—selected and not selected. Radio buttons appear as a group in which only one radio button can be selected at a time. To create new groups, radio buttons must be added to group boxes or panels. Each group box or panel is a group.
- Radio buttons and checkboxes use the CheckChanged event.
- A picture box (class PictureBox) displays an image (set by an object of class Image).
- Mouse events (clicks, presses and moves) can be handled for any GUI control that derives from System::Windows::Forms::Control. Mouse events use class MouseEventArgs (MouseEventHandler delegate) and EventArgs (EventHandler delegate).
- Class MouseEventArgs contains information about the x- and y-coordinates, the button used, the number of clicks and the number of notches through which the mouse wheel turned.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Key events are generated when keyboard’s keys are pressed and released. These events can be handled by any control that inherits from `System::Windows::Forms::Control`.

- Event `KeyPress` can return a `__wchar_t` for any ASCII character pressed. One cannot determine if special modifier keys (such as `Shift`, `Alt` and `Control`) were pressed.
- Events `KeyUp` and `KeyDown` test for special modifier keys (using `KeyEventArgs`). The delegates are `KeyPressEventHandler (KeyPressEventArgs)` and `KeyEventHandler (KeyEventArgs)`.
- Class `KeyEventArgs` has properties `KeyCode`, `KeyData` and `KeyValue`.
- Property `KeyCode` returns the key pressed, but does not give any information about modifier keys.
- The `KeyData` property includes data about modifier keys.
- The `KeyValue` property returns the key code for the key pressed as an integer.

**TERMINOLOGY**

- active window
- Alt property of class `KeyEventArgs`
- anchoring a control
- ASCII character
- autoscaling
- background color
- bitwise operator
- bitwise XOR operator (`^`)
- `__box` keyword
- boxing values
- button
- Button class
- button label
- checkbox
- CheckBox class
- checkbox label
- CheckedChanged event
- click a button
- click a mouse button
- Click event
- component
- container
- control
- Control property of class `KeyEventArgs`
- delegate
- deselected radio button
- docking a control
- drag and drop
- `Enabled` property of class `Control`
- Enter mouse event
- event
- event argument
- event delegate
- event driven
- event handler
- event multicasting

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
SELF-REVIEW EXERCISES

12.1 State whether each of the following is true or false. If false, explain why.

a) A GUI is a pictorial interface to a program.
   - True

b) Windows Forms commonly are used to create GUIs.
   - True

c) A control is a nonvisible component.
   - False. A control is a nonvisual user interface element.

d) All forms, components and controls are classes.
   - True

e) In the event-handling model, properties act as mediators between objects that generate events and methods that handle those events.
   - True

f) Class Label is used to provide read-only text instructions or information.
   - True

g) Button presses raise events.
   - True

h) Checkboxes in the same group are mutually exclusive.
   - True

i) Scrollbars allow the user to maximize or minimize a set of data.
   - True

j) All mouse events use the same event arguments class.
   - False. Different mouse events use different event arguments classes.

k) Key events are generated when keys on the keyboard are pressed and released.
   - True

12.2 Fill in the blanks in each of the following statements:

a) The active window is said to have the __________.
   - Title

b) The form acts as a(n) __________ for the components that are added.
   - Container

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
c) GUIs are driven.
d) Every method that handles the same event must have the same signature.
e) The information required when registering an event handler is the class and the...
f) A(n) text box displays only a single character (such as an asterisk) as the user types.
g) Class and class help arrange components on a GUI and provide logical group for radio buttons.
h) Typical mouse events include, and .
i) events are generated when a key on the keyboard is pressed or released.
j) The modifier keys are , , and .
k) A(n) event or delegate can call multiple methods.

ANSWERS TO SELF-REVIEW EXERCISES
12.1 a) True. b) True. c) False. A control is a visible component. d) True. e) False. In the event-handling model, delegates act as intermediaries between objects that generate events and methods that handle those events. f) True. g) True. h) False. Radio buttons in the same group are mutually exclusive. i) False. Scrollbars allow the user to view data that normally cannot fit in its container. j) False. Some mouse events use EventArgs, while others use MouseEventArgs. k) True.

12.2 a) focus. b) container. c) event. d) signature. e) event arguments, delegate. f) password. g) GroupBox, Panel. h) mouse clicks, mouse presses, mouse moves. i) Key. j) Shift, Control, Alt. k) multicast.

EXERCISES
12.3 Extend the program in Fig. 12.27–Fig. 12.28 to include a check box for every font style option. [Hint: Use XOR rather than testing for every bit explicitly.]

12.4 Create the GUI in Fig. 12.41. You do not have to provide any functionality.

Fig. 12.41 GUI for Exercise 12.4.

12.5 Create the GUI in Fig. 12.42. You do not have to provide any functionality.
12.6 Extend the program of Fig. 12.36–Fig. 12.37 to include options for changing the size and color of the lines drawn. Create a GUI similar to the one in Fig. 12.43. [Hint: Have variables to keep track of the currently selected size (int) and color (Color object). Set them using the event handlers for the radio buttons. For the color, use the various Color constants (such as Color::Blue). When responding to the mouse moves, simply use the size and color variables to determine the proper size and color.]

12.7 Write a program that plays “guess the number” as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1–1000. The program then displays the following text in a label:

I have a number between 1 and 1000–can you guess my number?
Please enter your first guess.

A text box should be used to input the guess. As each guess is input, the background color should change to either red or blue. Red indicates that the user is getting “warmer,” and blue indicates that the user is getting “colder.” A label should display either "Too High" or "Too Low" to help the user choose a number closer toward the correct answer. When the user obtains the correct answer, "Correct!" should be displayed. The background should become green and the text box used for input should become uneditable. Provide a button that allows the user to play the game again. When the button is clicked, generate a new random number, change the background to the default color and reset the input text box to editable.