Chapter 3

Working with GUI APIs

This chapter introduces us to the principles of working with GUI APIs. This chapter will:

- identify GUI element as the fundamental entity in GUI API programming;
- describe the process of building GUI application: the front-end layout design and the back-end programming support;
- differentiate front-end GUI elements from the back-end control variables;
- demonstrate the above principles based on GUI API; and
- demonstrate how to extend, organize, and customize GUI API for the needs of our application.

After this chapter we should:

- understand the approach to learning a modern GUI API; and
- be able to design/discuss implementation of GUI applications independent from API technologies;

In addition, with respect to hands-on and programming, we should:

- understand source code of simple GUI applications independent from the implementation API technologies; and
- be able to implement simple GUI applications based on MFC;

In this chapter, we want to understand the *principles* of working with modern GUI APIs. We emphasize *principles* because we are going to rely heavily on the MFC API to illustrate the ideas presented. However, it is important to remember that our focus is not learning the skills of using any particular API. Instead, we are interested in understanding the basic capabilities of modern GUI APIs such that we can examine and implement the requirements of event-driven interactive programs. For example, we should be able to apply the lessons learned from this chapter to other GUI APIs (e.g. the *Java Swing Library*, or the *Microsoft Forms Library*). After this chapter (and with some practice), we want to be able to pick up the reference manual of any modern GUI API and commence developing a simple interactive application.

The ultimate goal of this chapter is the understanding of GUI API's support for user interactivity and not to implement the Ball Shooting Program. A proper implementation of the Ball Shooting Program can only commence after we learn the software architecture for organizing our solution structure, which will be covered in Chapter 6.

3.1 Our Application and Existing Libraries

Graphical UI. Notice that the "*graphical*" in *GUI* refers to a user interface that is "*graphically*" oriented. For example a user interface with visually pleasing "*graphical*" buttons, or slider bars. This should not be confused with the "*computer graphics*" we are learning in this book.

When developing interactive computer graphics programs we work with existing tools, or software libraries, to develop our applications. In this book we are learning how to develop *user interactive graphics applications*, and thus we work with *graphical user interface* (*GUI*) libraries and *graphics* libraries. These libraries provide well defined *application programming interfaces* (*API*) with well documented functions and utility classes. The applications we develop use/subclass from the utility classes and call appropriate functions through the APIs. In this way, our application interacts with the users through a GUI API, and draws graphics through a Graphics API.

Examples of popular GUI API include: Graphics Utility Toolkit (GLUT), Fast and Light Toolkit (FLTK), Microsoft Foundation Classes (MFC). Examples of popular Graphics APIs include: OpenGL API, Microsoft Direct-X Direct3D (D3D), and Java 3D. In this book, we will work with FLTK and MFC to interact with the user; OpenGL and D3D to draw graphics. We show examples in more than one APIs to demonstrate that:

- GUI API: although the utility classes and function names may be significantly different, the *principles* of working with GUI APIs are very similar.
- Graphics API: although configuration procedures are different, and the functions have very different names, these APIs are designed based on exactly the same fundamental graphics *concepts*.

When reading this book, it is important to remember that the programming and APIs are there to help us learn the *concepts* and *knowledge*. In general, the skills in working with an API should be readily transferable new APIs.

In the rest of this chapter we will analyze how our computer graphics programs draw squares (or circles). It is important to keep in mind that at this point we are *not* learning the APIs, we are interested in understanding the process of drawing squares.

3.2 GUI Elements

As we have seen in Chapter 2, after the initialization, event-driven programs are simply a collection of routines that are driven by asynchronous external events. For this reason, facilitating the generation of appropriate events is key to implementing event-driven programs. In our case, since our programs are built to interact with users, our users are the main source of asynchronous events. The modern GUI APIs are designed to facilitate users in triggering appropriate events for our applications.

Modern GUI APIs define an elaborated set of *GUI elements* and associate extensive event structures with these GUI elements to support interactivity with the users and programmability of event service routines. For example, a GUI API would define a *button* GUI element and associate events like, *mouse over* (no click), *clicked*, *double clicked*, etc. with the button GUI element. The GUI API would then allow application programmers to register and service these events.

Recall that GUI elements are *virtual* input/output devices, typically represented as graphical icons. Other examples of common GUI elements include, slider bars, checkboxes, radio buttons, combo boxes, text boxes etc. A *window*, or an area with fancy boarders, is also an example of GUI element. A window GUI element is special because it serves as a container for other GUI elements. In general, a GUI-based application has at least one GUI element: the main application window. From the user's perspective, GUI elements should be:

- *visually pleasing*: for example, representing a button with a three dimensional looking icon; and
- *semantically meaningful*: for example the button should be properly labeled, and users understand one would move the mouse pointer over the icon and click the left mouse button to *activate* (depress) the button.

On the opposite end, from the programmer's perspective, a GUI element should have:

- *an unique identifier*: the application must be able to uniquely identify each GUI element to differentiate the actual triggering source of individual events;
- *default behaviors*: GUI elements should define default behaviors for mundane situations. For example, a depressed GUI button should *look* different from a *un-activated* button. As application programmer, we expect GUI elements behave appropriately to these types of *typical* situations.

- *customizable behaviors*: as application developers, we want to have the option of customizing the behaviors of GUI elements. For example, our application may demand a depressed button to have a special color.
- *state information*: certain types of interactions with the users require the corresponding GUI element types to retain *state information*. For example, a slider bar should record the knob position, a check-box should record if it is currently checked (true) or un-checked (false). When servicing events generated by these types of GUI elements, our application must *poll* the corresponding state information.

Control Variables. Variables in our program code that are associated with and represent GUI elements.

- *abstract representation*: customizable behaviors and polling of state information imply that our program must have *variables* referencing the corresponding GUI elements. These types of variables are referred to as *control variables*, where through a control variable, our program code can *control* a GUI element. To properly reflect the different functionality, control variables for different GUI element types should be of different data types. For example, there should be a $CButton$ class for buttons and $CSliderBar$ class for representing slider bar GUI elements.
- *event service registration mechanisms*: as we have seen many times, this is probably the single most important functionality we expect from GUI elements.

It is interesting to note that sometimes it is desirable to have GUI elements controlled by the application. For example, in the ball shooting program, as the hero ball free falls under gravitational force, the slider bar GUI elements are controlled by the application to correctly reflect the hero ball velocity.

3.3 Building A GUI Application

We can perceive a GUI application as being a collection of GUI elements, where through these GUI elements a user can trigger events to cause changes to an application state in accomplishing desired tasks. Event-driven programming model taught us the *back-end* of a GUI application: how to react to events and cause changes to an application state. GUI API provides the mechanism for building the *front-end* of a GUI application: how to put together the collection of GUI elements to support the generation of appropriate events.

3.3.1 Front-End: Layout of GUI Elements

The first step in building a GUI application is to *design the layout* of the user interface system. This is referred to as the *front-end* because the results are the *front* of our application where the user can see and have to interact with. In this step, the application developer must determine the locations and appearances of

every GUI elements. Modern GUI APIs typically support this process with a *GUI builder* program. A GUI builder is an interactive graphical editor that allows its user interactively place and manipulate the appearances of all GUI elements. This layout process typically involves a developer placing icons representing GUI elements into a rectangular area representing the application window. The developer would then adjust each GUI elements appearances (e.g., color, size, etc.). The results of GUI element layout is usually stored in some data files. The developer would include these data files with the rest of the development source code. When compiled and linked appropriately, the resulting program would display the designed GUI layout.

The goal of this first step is to arrange and manipulate the GUI elements to present an aesthetic pleasing, logically meaningful, and intuitively easy to use user interface. These are the topics of *User Interface Design*, an entire field in Computer Science discipline. In this book we will only describe the basic process involved in building a GUI. Our GUI front-ends are *sufficient* but they may *not* be the best, or even good.

3.3.2 Back-End: Establish Semantic Correspondence

As we have already seen, the semantic meanings of GUI elements are defined by the corresponding event service routines. For example, a mouse click over a button has no real meaning. It is the event service routine which quits the application that defines the semantic meaning of the button. We refer to this as the *back-end* because the results of this step are programming codes that operate *behind* the visible user interface. Typically, GUI builder programs provide mechanisms for supporting the registration of event service routines. For example, a GUI builder would display a list of defined events for a particular GUI element. The developer would have the option of *registering* for an event by entering a service function name.

Similar types of support are also available for defining control variables. The developer would indicate to the GUI builder (e.g., by clicking on an appropriate property sheet) that a control variable should be defined for a particular GUI element. Based on the GUI element type (e.g., a button), the GUI builder would typically pre-determine the data type (e.g., CButton) for the control variable and prompt the developer for the variable name.

Notice that in both cases, the developer would entered program code fragments (names of service functions and control variables) into the GUI builder program. As mentioned, at the conclusion of GUI builder program, information are saved into some data files. It is important for the GUI builder to integrate these code fragments with the rest of the event-driven program source codes. There are two different mechanisms for supporting this integration:

1. External Linkage. Some GUI builders require the developer to enter the entire event service routine program code directly into the GUI builder pro-

gram. The GUI builders would then generate extra program modules in the form of source code files that contains the event service routines. The developer would include these source code files as part of the development project.

2. Internal Direct Code Modification. Some GUI builders modify and insert function prototypes and/or control variable declaration/initialization directly into the source files in the development project. The developer would then edit the same source file to enter event service routine program code.

The advantage of the external linkage mechanism is that the GUI builder has minimal knowledge of the application source code. This provides a simple and flexible development environment where the developer is free to organize the source code structure, variable names, etc., in any appropriate way. However, the externally generated programming module implies a loosely integrated environment. For example, to modify the behavior of a GUI element, the application developer must invoke the GUI builder, modify code fragments, and re-generate the external program module.

The internal direct code modification mechanism in contrast, provides a better integrated environment where the GUI builder modifies the application program source code directly. However, to support proper "direct code modification," the GUI builder must have intimate knowledge of, and often places severe constraints on, the application source code system (e.g., source code organization, file names, variable names, etc.).

Control Variables

There are two situations where the back-end event driven program must define a control variable to represent a front-end GUI element.

- Polling/Setting of GUI element state information. A control variable should be defined for GUI elements with state information that must be polled/set during run time. For example, a checkbox's state information is altered every time a user clicks it. When servicing a checkbox's click event, the service routine must poll the GUI element (through the control variable) for its state. Notice that in this case, the data type of the checkbox control variable should be of $bool$. To support polling and setting operations, the data type of the corresponding control variable should reflect the state information and *not* the GUI element type. For example, slider bar's state information is the position of the knob, or a $float$. Thus, control variables for polling and setting a slider bar GUI element should be a variable with $float$ data type and *not CSliderBar*.

- Customization of GUI element behavior. A control should be defined for GUI elements when an application demands customized behaviors. For

example, an application may demand state of a checkbox be associated with the color of the checkbox: green for checked and red for un-checked. In this case, during event servicing, we must first poll the GUI element state and then modify the color attributes of the checkbox. This means, the application needs to have access to the $CCheckBox$ data type.

3.4 Examples: FLTK and MFC

3.4.1 FLTK - Fluid and External Service Linkage

Figure 3.1 shows a screen shot of working with *Fluid*, FLTK's GUI builder. In the lower-right corner of Figure 3.1, we see that (A) Fluid allows an application developer to interactively place graphical representations of GUI elements (3D-looking icons); (B) is an area representing the application window. In addition (C), the application developer can interactively select each GUI element to define its physical appearances (color, shape, size, etc.). In the lower-left corner of Figure 3.1, we see that (D) the application developer has the option to type in program fragments to define control variable for and to service events generated by the corresponding GUI element. In this case, we can see that the developer must type in the program fragment for handling the X velocity slider bar events. This program fragment will be separated from the rest of the program source code system and will be associated with Fluid (the GUI builder). At the conclusion of the GUI layout design, the user can instruct Fluid to generate source code files to be included with the rest of the application development environment. In this way, some source code files are controlled and generated by the GUI builder and the application developer must invoke the GUI builder in order to update/maintain the control variables and the event service routines. FLTK implements the external service linkage.

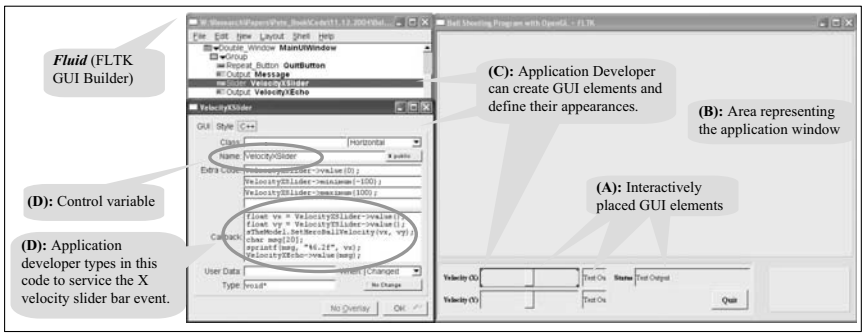

Figure 3.1: Fluid: FLTK's GUI Builder.

3.4.2 MFC - Resource Editor and Direct Code Modification

Figure 3.2 shows a screen shot of the MFC resource editor, MFC's GUI builder. Similar to Fluid (Figure 3.1), in the middle of Figure 3.2, (A) we see that the resource editor also supports interactive designing of the GUI element layout in (B), an area representing the application window. Although the GUI builder interfaces operate differently, we observe that in (C), the MFC resource editor also supports the definition/modification of the physical appearance of GUI elements. However, unlike Fluid, the MFC resource editor is tightly integrated with the rest of the development environment. In this case, a developer can register event services by inheriting or overriding appropriate service routines. The MFC resource editor automatically inserts code fragments into the application source code system. To support this functionality, the application source code organization is governed/shared with the GUI builder; the application developer is not entirely free to rename files/classes and/or to re-organize implementation source code file system structure. MFC implements internal direct code modification for event service linkage.

3.5 Implementation Notes

Before we begin examining examples of implementations in detailed, it is important that we take note of a few important characteristics/pitfalls of programming with GUI APIs.

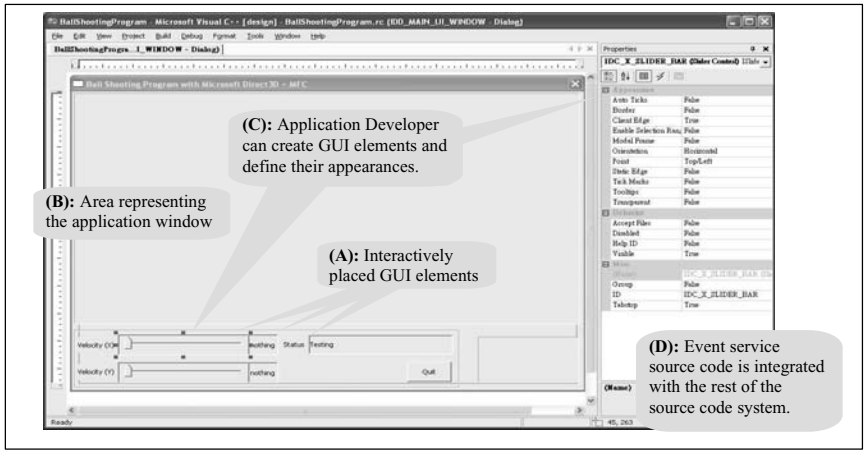

Figure 3.2: The MFC resource editor.

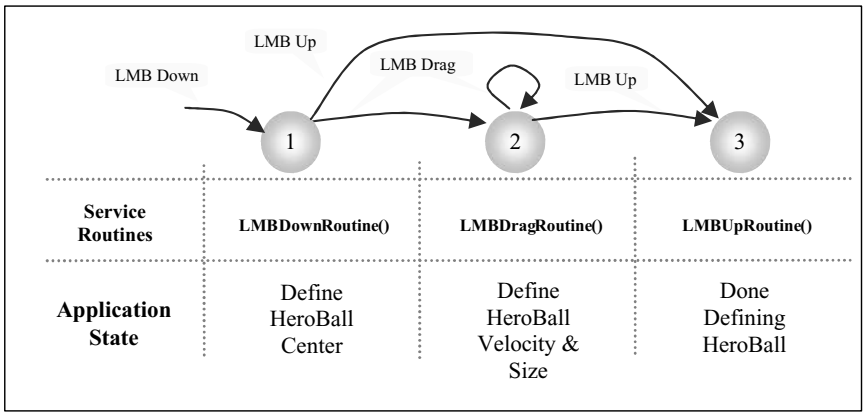

Figure 3.3: State diagram for defining the HeroBall.

Application State: the application state of an event-driven program must persist over the entire life time of the program. In terms of implementation, this means that the application state should be defined based on variables that are dynamically allocated during run time and that reside on the heap memory. These are in contrast to local variables that reside on the stack memory and which do not persist over different function invocations.

Implicit Events: the mapping of user actions to events in the GUI system often results in *implicit* and/or undefined events. In our ball shooting solution, the actions to define a HeroBall involve left mouse button down and drag. When mapping these actions to events in our implementation (in Listing 2.2 and Listing 2.6), we realize that we should also pay attention to the implicit mouse button up event. Another example is the HeroBall selection action: right mouse button down. In this case, right mouse button drag and up events are not serviced by our application, and thus, they are undefined (to our application).

Consecutive User Actions: when one user action (e.g., "*drag out the HeroBall*") is mapped to a group of consecutive events (e.g., mouse button down, then drag, then up) a finite state diagram can usually be derived to help design the solution. Figure 3.3 depicts the finite state diagram for defining the HeroBall. The left mouse button down event puts the program into State 1 where, in our solution from Listing 2.6, *LMBDownRoutine*() implements this state and defines the center of the HeroBall, etc. In this case the transition between states is triggered by the mouse events, and we see that it is physically impossible to move from State 2 back to State 1. However, we do need to handle the case where the user action causes a transition from State 1 to State 3 directly (mouse button down

and release without any dragging actions). This state diagram helps us analyze possible combinations of state transitions and perform appropriate initializations.

Input/Output Functionality of GUI Elements: an input GUI element (e.g., the quit button) is an artifact (e.g., an icon) for the users to generate events to cause changes the application state, while an output GUI element (e.g., the status bar) is an avenue for the application to present application state information to the user as feedback. For both types of elements, information only flows in one direction—either from the user to the application (input) or from the application to the user (output). When working with GUI elements that serve both input and output purposes, special care is required. For example, after the user selects or defines a HeroBall, the slider bars reflects the velocity of the free falling HeroBall (output), while at any time, the user can manipulate the slider bar to alter the HeroBall velocity (input). In this case, the GUI element's displayed state and the application's internal state are connected. The application must ensure that these two states are consistent. Notice that in the solution shown in Listing 2.2, this state consistency is not maintained. When a user clicks the RMB (B2 in Listing 2.2) to select a HeroBall, the slider bar values are updated properly; however, as the HeroBall free falls under gravity, the slider bar values are not updated. The solution presented in Listing 2.6 fixes this problem by using the *ServiceTimer*() function.

Redraw/Paint Events: in Section 2.4.3, it is stated that

> "... Redraw/Paint is the single most important event that an application must service ... "

Recall that the Redraw/Paint event is a reminder from the window manager that the content of our application window may be out-of-date. It is in our application's interests to update the window content as soon as possible, preferably before the user notices any inconsistency. As we have seen in the Ball Shooting Program solution presented in Section 2.6, to maintain smooth animation of simulated results, applications with real-time simulation must redraw the application state at a rate of more than 20 times in a second. With such high redraw rate, the application's window content can never be out-of-date for more than a few milliseconds. For this reason, applications with default high-redraw rates do not need to service the Redraw/Paint event. Since most of the applications presented in this book involve real-time simulations with more than 20 redraws in a second, Redraw/Paint events are typically *not* serviced.

OnIdle Event: many GUI API defines an *OnIdle* event to be triggered when *nothing* is happening, or when the application is *idling*. Some application with real-time simulation chooses to service this event to compute simulation updates.

In this way, the application can taking advantage of idled CPU cycles for simulation computations. The main disadvantage of this approach is that as developer, we cannot predict the events:

1. if the application is completely idle, this event will be triggered continuously, as a result the simulation process will be updated more frequently than necessary;

2. if the application gets busy, (e.g., bursts of user activities, etc.) and are short on idle times, there are chances that the simulation may lag behind real-time resulting in inconsistent display.

In practice, modern machines are fast enough that idle cycles are *almost* always available. Many commercial applications are developed with updates during the *OnIdle* events. For dependable and predictable update rates, we have chosen to compute simulations during the application programmed *OnTimer* events.

3.6 Tutorials and Code Base

Modern GUI APIs are highly sophisticated often with steep initial learning curves. When learning to work with GUI API we should constantly remind ourselves that ultimately, we are interested in building interactive graphics programs. Our goal is not to become an expert in GUI programming. Rather, we are interested in understanding the principles of working with GUI API such that we can commence on building graphics applications. In this section, we use MFC to experience the different aspects of event programming studied. It is important to understand the underlying concepts behind the APIs, and then be *comfortable* at using one of the APIs. The important lesson we should learn from the short history of our discipline is that APIs change and evolve rapidly; we must always be ready to to learn new ones.

The tutorials from this section serves two purposes, demonstrate: how to work with GUI API; *and* how to work with MFC. In the beginning of each tutorial, explicit **goal** and **approach** statements identify the key ideas that will be demonstrated in the tutorial. These are "API independent" statements, where to work with any GUI API, one must understand how to accomplish these tasks. Typically the procedures involved in programming with different GUI API may be different, but the end results would contains similar elements: e.g. control variables with appropriate data types, event service routines via object oriented override and/or via callback function registration, etc.

In the following tutorials we describe results of working with MFC and relate these results to our learning from earlier sections. For a detailed *step-by-step* how-to's for the MFC Resource editor, please refer to ***THE CD* - William Frankhouser's guide**. To demonstrate the concepts presented are indeed language and GUI API independent, the following tutorials are also implemented in

C# and WinForm API. The corresponding detailed *step-by-step* guide can also be found on the ***THE CD* - Ethan Verrall's guide**. On-line resources are excellent avenues for learning basic skills in working with these software systems. Jason Pursell has developed an excellent *step-by-step* guide on working with the MFC editor that are tailored for the MFC-based tutorials in this book[1].

Here is a brief summary of the tutorials in this section:

Tutorial	Demonstrates	Reusable Codebase
3.1	Demystify source code files	-
3.2	GUI elements and control variables	-
3.3	More with control variables	-
3.4	Application defined events	-
3.5	Input/Output GUI elements	-
3.6	Extending GUI API with classes	$SliderCtrlWithEcho$
3.7	Custom GUI library	$MFC\_Library$
3.8	Grouping of GUI elements	$ReplaceDialogControl()$

Tutorial 3.1.
Project Name
MFC_SimpleDialog

Tutorial 3.1: Demystifying the files.

- **Goal:** to demonstrate that while on first look, programming with GUI API can appear to be overwhelming, with systemic approach of analyzing the purpose of the files in the environment, we can gain significant understanding of the system.

- **Approach:** familiarize and demystify the typically intimidating GUI development source code structure; create the simplest possible application with a GUI API and analysis every file; identify the purpose of files and draw attention to how these files are related/un-related to GUI development.

Figure 3.4: Tutorial 3.1.

Figure 3.4 is a screen shot of running Tutorial 3.1. This entire tutorial is automatically generated by the MFC wizard of the Microsoft Visual-C++ (VC++) Integrated Development Environment (IDE). This program does not do anything at all. When user clicks on the "OK" or the "Cancel" button, the application simply quits. This application does not do anything else.

Here is the source code structure:

[1]This guide can be found off Jason's home page at: http://home.myuw.net/jpursell/ or http://faculty.washington.edu/ksung/biga/chapter_tutorials/VC_Guide/CreatingDialogAppWithMFC7.pdf.

Purpose	**Files**	**Folder**
Source code	TutorialApp(.cpp,.h) TutorialDlg(.cpp,.h) stdafx(.cpp,.h), Resource.h	MFC_SimpleDialog (main project folder)
Documentation	Readme.txt	
GUI builder	*.rc	
IDE project	*.vcproj, *.sln	
IDE scratch	*.aps, *.ncb	
Icon	*.ico, *.rc2	\res
Debugging	*.exe, *.obj (*compiled results)*	\Debug
Release	*.exe, *.obj (*compiled results)*	\Release

We will examine the files according to the above ***purpose*** of all these files.

- **Source code:** these are the files that contain the source code to the application. Notice that there are seven (7) source files for this simple project! We will examine the details of these source files after examining the rest of the supporting files in this development project.

- **Documentation:** The $ReadMe.txt$ file is a *text* file meant for programmer to fill-in comments.

- **GUI builder:** The $*.rc$ file is the data file of the MFC GUI builder (i.e. the resource editor):

Source File. *MFC_SimpleDialog.rc* file in the *Resource Files* folder of the *MFC_SimpleDialog* project.

Source Listing. For the convenience of description, in general the presented source code has been significantly edited from the original source file. Readers should understand the book, and expect to see the code fragments in different orders when examining the source code files.

```
// Microsoft Visual C++ generated resource script.
...
#include "Resource.h"
...
   IDD_MFC_SIMPLEDIALOG_DIALOG is the identifier for the main applicaiton window.
   The following describe the dimension and appearance of this GUI element.
IDD_MFC_SIMPLEDIALOG_DIALOG DIALOGEX 0, 0, 320, 200
STYLE DS_SHELLFONT | WS_POPUP | WS_VISIBLE | WS_CAPTION | DS_MODALFRAME
EXSTYLE WS_EX_APPWINDOW
CAPTION "MFC_SimpleDialog"
...
   The following lines descrbe the locations and dimenstions of OK and Cancel buttons.
BEGIN
    DEFPUSHBUTTON    "OK",IDOK,263,7,50,16
    PUSHBUTTON       "Cancel",IDCANCEL,263,25,50,16
...
```

Listing 3.1: Snippet from $MFC\_SimpleDialog.rc$ (Tutorial 3.1).

Listing 3.1 shows that this is a simple *text* file, where it is possible to open/edit this file with any simple text editor (e.g. notepad). We see that among other data, this file includes information that describes the location and appearance of all the GUI elements on the application window.

- **IDE project:** the $.vcproj/.sln$ files are used by the VC++ IDE for defining the source code project (e.g., what are the source file, how to compile the project, etc.). These are also text files, interested readers can open/edit these files with any text editors to examine the details.

- **IDE scratch:** the $.aps/.ncb$ are temporary files kept/generated by the IDE while supporting our development process. These files can be deleted in between editing/development sessions.
- **Debug:** The IDE uses this folder to record all the compile results, including all the $.obj$ files and the eventual $.exe$ file (executable program). The content of this folder are meant to support interactive debugging in the IDE. All the $.obj$ and the executable files contain intermediate symbols for interactive debugging. As a result, files from this folder are much larger than they need be.
- **Release:** The IDE uses this folder to record all the compiled results. In contrast to the $Debug$ folder, the files in this folder are for supporting final *release* of the program (after development is done). The executable files from this folder is optimized for size and speed.

Here are the details of the 7 source files:

GUI ID. The *GUI ID* is an unique identifier associated with each GUI element. GUI ID is used mainly by the MFC resource editor.
Source File. *Resource.h* file in the *Header Files* folder of the *MFC_SimpleDialog* project.

- $Resource.h$: this file defines the symbolic names (or $GUIID$) of GUI elements used in the resource editor:

```
// Microsoft Visual C++ generated include file.
// Used by MFC_SimpleDialog.RC
//
...
#define IDD_MFC_SIMPLEDIALOG_DIALOG        102
...
```

Listing 3.2: Snippet from $Resource.h$ file (Tutorial 3.1).

For example, as shown in Listing 3.2, $IDD\_MFC\_SIMPLEDIALOG\_DIALOG$ is the symbolic name of the GUI element that represents the main application window. In this case, we see that this symbol is being defined to be a unique integer. Notice that in Listing 3.1, the $MFC\_SimpleDialog.rc$ file includes the $Resource.h$ file to define the $IDD\_MFC\_SIMPLEDIALOG\_DIALOG$ symbol.

Working with pre-compile header. After changing and/or creating a header file in a project, it is a good idea first to re-compile $stdafx.cpp$ to ensure up-to-date pre-compile header information.

- $StdAfx.h/.cpp$: These two files contain *pre-compile header* information. They help speed up the compilation process. These two files are important and exists in all of our projects. Fortunately the handling of *pre-compile header* is fairly transparent and we will not change these files.

Source File.
TutorialApp.h/cpp file in the *Source* and *Header* Files folders of the *MFC_SimpleDialog* project.

- $TutorialApp.h/.cpp$: These two files define and implement the $CTutorialApp$ class:

```
class CTutorialApp : public CWinApp {
  ...
  virtual BOOL InitInstance();
};
...
```

theApp: *this is the abstraction of our application.*

```
    CTutorialApp theApp; // The one and only CTutorialApp object
...
BOOL CTutorialApp::InitInstance() {
```

```
    ... // Code for intialization
    // Create and show our dialog
    CTutorialDlg dlg;    // This is the our main application window
    m_pMainWnd = &dlg;
    ...
    // display the main application window, when the application window
    // quits, the application should terminate.
    ...
}
```

Listing 3.3: The $CTutorialApp$ class (Tutorial 3.1).

Listing 3.3 shows that $CTutorialApp$ subclasses from the API system $CWinApp$. This class represents our application. The global variable, $theApp$, represents the instance of our running program. Notice that our main application window is a $CTutorialDlg$ object and is instantiated and invoked in the $InitInstance()$ function. When our application window exits, the control will return to $InitInstance()$ and the application terminates.

- $TutorialDlg.h/.cpp$: These two files define and implement the $CTutorialDlg$ class. As shown in Listing 3.4, $CTutorialDlg$ subclasses from the MFC $CDialog$ class. Recall that our main application window is a GUI element, and that we can define control variables for GUI elements. Now, refer to Listing 3.3, the dlg object instantiated in in the $InitInstance()$ function is the control variable for our main application window, and the data type for our main application window is $CTutorialDlg$. Listing 3.3 shows dlg is referenced by $m\_pMainWnd$, an instance variable of $theApp$.

Source File.
TutorialDlg.h/cpp file in the *Source* and *Header* Files folders of the *MFC_SimpleDialog* project.

```
class CTutorialDlg : public CDialog {
    ...
    CTutorialDlg(CWnd* pParent = NULL);    // standard constructor
    virtual BOOL OnInitDialog();           // The SystemInitialization() function
    afx_msg void OnPaint();                // To Support Redraw/Paint event
    DECLARE_MESSAGE_MAP()                  // Identify the events to override from super class
    virtual void DoDataExchange(CDataExchange* pDX);  // GUI element input/output support
    ...
};
```

***TutorialDlg.cpp**: implementation file*

```
CTutorialDlg::CTutorialDlg( ... ){...}  // Constructor
```

A:
```
BOOL CTutorialDlg::OnInitDialog() { ... }
    ...
```

B:
```
BEGIN_MESSAGE_MAP(CTutorialDlg, CDialog)
    ON_WM_PAINT() ...
```

C:
```
void CTutorialDlg::OnPaint() { ... }
    ...
```

D:
```
void CTutorialDlg::DoDataExchange(CDataExchange* pDX) { ... }
    ...
```

Listing 3.4: The $CTutorialDlg$ class (Tutorial 3.1).

When we examine the structure of the $TutorialDlg.cpp$ file (refer to Listing 3.4), we observe:

- *Object Oriented Mechanism:* The $CTutorialDlg$ subclasses from the MFC $CDialog$ class, this means, our main application window will inherit ***vast*** predefined functionality from the $CDialog$ class (e.g., mouse events, timer events, etc). Notice the virtual functions that are override (e.g. $OnInitDialog()$), these are examples of modifying predefined GUI element behaviors.

- *SystemInitialization* (**A**): The $OnInitDialog()$ is our opportunity to perform initialization for the application window, much like the functionality of the $SystemInitialization()$ function discussed in Section 2.2. Since we are overriding a *virtual* function, it is important to follow the protocol. In this case, we must the super-class method $CDialog :: OnInitDialog()$, and we must a return $TRUE$ upon successful operations.

- *Redraw/Paint Event* (**B** and **C**): The $BEGIN\_MESSAGE\_MAP$ macro (**A**) helps register events for services. Here we see the registration of ON_WM_PAINT, the Redraw/Paint event. In this case, $CTutorialDlg$ must override the $OnPaint()$ function to service the event. Since the application window is empty, this function does not do anything.

- *GUI Element and Control Variable* data exchange (**D**): As we will see in later tutorials, this function helps maintain the consistency of state information between the front-end GUI elements and their corresponding control variables in the back-end.

Source file naming convention: Notice that of the above file names, $Resource.h$, $StdAfx.h, .cpp$ are names governed by MFC. We will name *all* of the tutorials in this book $TutorialApp.h/.cpp$ for the application and $TutorialDlg.h/.cpp$ for the main application window. For the rest of the tutorials in this chapter, we will only need to work with the $CTutorialDlg$ class (our main application window).

Tutorial 3.2.
Project Name
MFC_EchoButtonEvent

Figure 3.5: Tutorial 3.2.

Tutorial 3.2: GUI elements and control variables

- **Goal:** practice and experience event-driven programming design and implementation; **Approach:** design a solution to a very simple problem, examine how the solution is implemented.

- **Goal:** experience and understand front-end GUI elements vs. back-end control variables; **Approach:** work with output-only GUI element, where at run time, our application must change the state of the GUI element.

Figure 3.5 is a screen shot of running of Tutorial 3.2. This is a slightly more interesting application where we count the number of times the "*Click to Add*" button has been clicked. If we have to design a solution for this tutorial, our experience from Section 2.2 and Listing 2.4 would lead us to an application state with a simple integer counter, and a button event service routine the increments the counter:

System Initialization:

```
// Define Application State: count the number of button clicks
    int m_OkCount = 0;  // initialize to 0
// Register Event Service Routines
    Register for:  ButtonClick Event
```

Events Services:

```
ButtonClick      // service routine for button click
    m_OKCount++;            // if button is clicked, increment the count
```

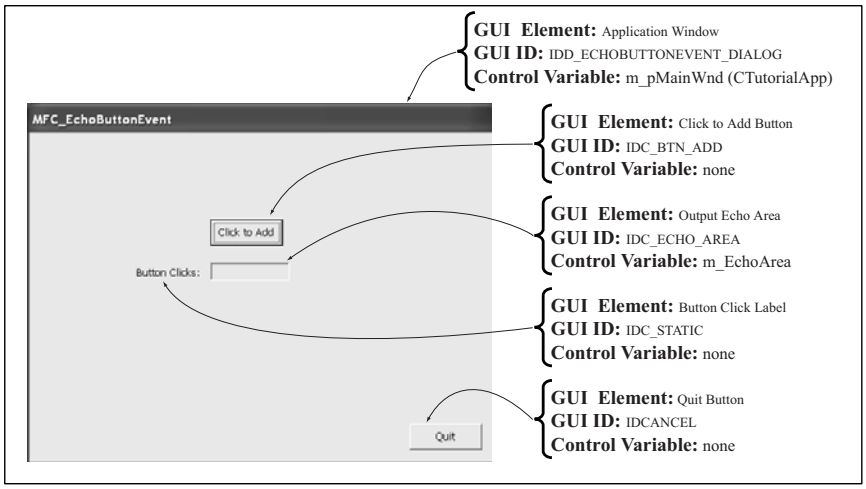

Figure 3.6: GUI Elements of Tutorial 3.2.

```
EchoToScreen(m_OKCount) // echo this new count to the application window
```

Listing 3.5: Abstract Event-Driven Programming Solution (Tutorial 3.2).

To implement the solution of Listing 3.5, we must first design the layout of the GUI application and then register the event services with the appropriate GUI element.

Front-end GUI Layout Design

Figure 3.6 shows that in our Tutorial 3.2 implementation, the front-end user interface has five GUI elements:

- The "*application window*": We assign $IDD\_ECHOBUTTONEVENT\_DIALOG$ to be the GUI ID for the application window GUI element. As in Tutorial 3.1, the $CTutorialDlg$ class represents this GUI element, and that $theApp.m\_pMainWnd$ is the control variable that references to our main application window.

- The "*Click to Add*" button: We assigned ID_BTN_ADD to be the GUI ID of this button. This is an *input-only* GUI element because our application never changes its state. Our application is only interested in receiving *click* events generated by this button. Since our application does not have any need to refer to this element, we do not need to define a control variable for this element.

theApp. Recall from Tutorial 3.1, that *theApp* is an instance of *CTutorialApp*, it represents our application. In *TutorialApp.cpp*, we set the *m_pMainWnd* instance variable to reference to the control variable representing our application main window.

GUI ID. GUI IDs are defined in the *Resource.h* file.

- The "*Output Echo Area*": We assigned IDC_ECHO_AREA to be the GUI ID this element. This is the echo area where the application state information (number of clicks) will be displayed. At run time, our application needs to change the content of this GUI element. In our implementation, we define $m\_EchoText$ to be the control variable of this GUI element.
- The "*Button Clicks*" label: This is a *static* GUI element because it provides neither input nor outpu functionality. It is a simple label.
- The "*Quit*" button: This is the same button from Tutorial 3.1. Our application has no reference to this GUI element. This GUI element supports the default MFC service, where it reacts to a click event by quitting the application.

Back-end Implementation

Source File. *TutorialDlg.h/cpp* file in the *Source* and *Header* Files folders of the *MFC_EchoButtonEvent* project.

```
class CTutorialDlg : public CDialog {
    ... // removed content similar to that from Listing 3.4 (Tutorial 3.1).
    A1:  int      m_OkCount;                  // count of "Click to Add" button is clicked
    B1:  afx_msg void OnBnClickedBtnAdd(); // "Click to Add" service routine
    C1:  CString m_EchoText;                 // For controling the output GUI element
};
```

```
... // removed content similar to that from Listing 3.4 (Tutorial 3.1).
BOOL CTutorialDlg::OnInitDialog() {
    A2:  m_OkClick = 0;    // initialize application state.
    ...
BEGIN_MESSAGE_MAP(CTutorialDlg, CDialog)
    B2:  ON_BN_CLICKED(ID_BTN_ADD, OnBnClickedBtnAdd)
    ...
void CTutorialDlg::DoDataExchange(CDataExchange* pDX) {
    C2:  DDX_Text(pDX, IDC_ECHO_AREA, m_EchoText);
    ...
B3: This is the "Click to Add" button serviceroutine that we have registered with the MFC.
void CTutorialDlg::OnBnClickedBtnAdd() {
    A3:  m_OkCount++;                          // update application state
    C3:  m_EchoText.Format("%d", m_OkCount); // convert to text for output
... C4:  UpdateData(FALSE);                    // flush count to window
```

Listing 3.6: $CTutorialDlg$ class (Tutorial 3.2).

The development environment structure for Tutorial 3.2 is identical to that of Tutorial 3.1 with the same folder structure and same seven source code files. In addition, following the naming convention of the source code files, we notice $TutorialApp.h/.cpp$ defining the application, and $TutorialDlg.h/.cpp$ defining the main application window. When we compare the $TutorialDlg.h/.cpp$ with those from Tutorial 3.1, we notice slight differences. These differences are the implementation of the functionality in Tutorial 3.2. Listing 3.6 highlights these differences. From Listing 3.6 we notice:

- Application State (**A**): as defined by the solution from Listing 3.5, at label **A1** the application state is defined as an integer. At label **A2**, the application

OnInitDialog(). Recall that the *OnInitDialog()* should implement the *SystemInitialization()* functionality.

state is initialized in the $OnInitDialog()$ function. The application state is updated during the button event service routine (**A3**).

- Support for the button GUI element (**B**): the even service routine is declared at label **B1** in the $TutorialDlg.h$ file. At label **B2**, we register the $ON\_BN\_CLICKED$ (on button click) event and associate the GUI element ID_BTN_ADD with the $OnBnClickedBtnAdd()$ routine. This process registers the routine as a *callback* function for the click event on the button. The event service routine is implemented at label **B3**, where we update the application state and display the updated information to the output echo area.

ID_BTN_ADD. Refer to Figure 3.6 this is the GUI ID for the "*Click to Add*" button.

- Output GUI element (**C**): at label **C1** we define the variable $m\_EchoText$, and at label **C2**, this variable is *bound* to the GUI element IDC_ECHO_AREA. Notice that before the binding statement at label **C2**, $m\_EchoText$ is simply just another instance variable. However after the statement at **C2**, $m\_EchoText$ becomes the control variable of the IDC_ECHO_AREA. After the application state is updated in the button event service routine, at label **C3**, the $m\_EchoText$ is set with the updated application state. At this point, the value of the $m\_EchoText$ control variable is *different* from the state information presented in the IDC_ECHO_AREA. The statement at **C4** flushes the content of $m\_EchoText$ to the GUI element to ensure consistency.

IDC_ECHO_AREA. Refer to Figure 3.6 this is the GUI ID for the "*output echo*" area.

Tutorial 3.3: More with control variables

Tutorial 3.3.
Project Name
MFC_SliderControls

- **Goal:** understand that control variables can be complex data types; and demonstrate working with control variables of input GUI elements.
- **Approach:** work with slider bars, a fairly complex GUI element.

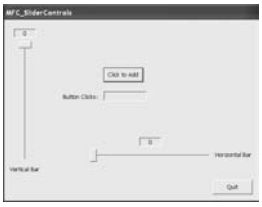

Figure 3.7: Tutorial 3.3.

Figure 3.7 is a screen shot of running Tutorial 3.3. This application is basically Tutorial 3.2 with two extra slider bar sets. From the GUI layout, we see that each slider bar set actually consists of three GUI elements: the label, the slider bar, and the echo area:

GUI Element	GUI ID	Control Variable
Slider Bars	$IDC\_V\_SLIDER\_BAR$	$m\_VSliderBar$
	$IDC\_H\_SLIDER\_BAR$	$m\_HSliderBar$
Slider Bar Labels	*Vertical Bar*	–
	Horizontal Bar	–
Slider Bar Echo Areas	$IDC\_V\_SLIDER\_ECHO$	$m\_VSliderEcho$
	$IDC\_H\_SLIDER\_ECHO$	$m\_HSliderEcho$

We have worked with *label* and *output* echo area in Tutorial 3.2. In this tutorial, we will concentrate on working with the slider bar, and servicing of slider bar

events. When we examining the development environment, we see the familiar source code structure. Once again, we will concentrate on $TutorialDlg.h/.cpp$. Listing 3.7 highlights the new programming code fragments that are associated with the slider bars:

Source File. *TutorialDlg.h/cpp* file in the *Source* and $Header$ Files folders of the *MFC_SliderControls* project.

```
class CTutorialDlg : public CDialog {
    ... // removed content similar to that from Listing 3.6 (Tutorial 3.2).
    A1: CString m_HSliderEcho, m_VSliderEcho;
    B1: CSliderCtrl m_VSliderBar, m_HSliderBar;
    C1: afx_msg void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);
        afx_msg void OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);
};

... // removed content similar to that from Listing 3.6 (Tutorial 3.2).
void CTutorialDlg::DoDataExchange(CDataExchange* pDX) {
    A2: DDX_Text(pDX, IDC_H_SLIDER_ECHO, m_HSliderEcho);
        DDX_Text(pDX, IDC_V_SLIDER_ECHO, m_VSliderEcho);
    B2: DDX_Control(pDX, IDC_V_SLIDER_BAR, m_VSliderBar);
        DDX_Control(pDX, IDC_H_SLIDER_BAR, m_HSliderBar);
...
BEGIN_MESSAGE_MAP(CTutorialDlg, CDialog)
    C2: ON_WM_HSCROLL()
        ON_WM_VSCROLL()
...
BOOL CTutorialDlg::OnInitDialog() {
    A3: m_VSliderEcho.Format("%d", 0); m_HSliderEcho.Format("%d", 0);
    B3: m_VSliderBar.SetRange(0, 100, TRUE); m_VSliderBar.SetPos(0);
        ...
    B4: UpdateData(false);
...
C3: This is the Horizontal scroll bar serviceroutine we override from MFC.
void CTutorialDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar) {
  // check to make sure we know which slider bar is generating the events
  if (pScrollBar == (CScrollBar *) &m_HSliderBar) {
    C4: int value = m_HSliderBar.GetPos();
    C5: m_HSliderEcho.Format("%d", value);
    C6: UpdateData(false);
...
C7: This is the Vertical scroll bar serviceroutine we override from MFC.
void CTutorialDlg::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar) {
  // content complements that of OnHScroll
```

Listing 3.7: $CTutorialDlg$ class (Tutorial 3.3).

- Slider bar echo control variables (**A**): we see declaration of the variables at label **A1**. Note that $m\_HSliderEcho$ is simply another variable until the DDX_Text macro call at label **A2**. After this macro call, $m\_HSliderEcho$ becomes the control variable for $IDC\_H\_SLIDER\_ECHO$ GUI element. These control variables are initialized at label **A3**.

- Slider bar control variables (**B**): we see the declaration of the variables at label **B1**. These will be the control variables for the slider bars. Notice the $CSliderCtrl$ data type. MFC pre-defines data types to support every types of GUI elements (e.g. $CButton$ for button GUI elements, $CComboBox$ for combo-box GUI elements, etc.). At label **B2**, these variables are bound to the corresponding slider bar GUI elements. At label **B3**, during the initialization $OnInitDialog()$ function call, we initialize the

max, min, and initial position of the slider bar knobs. After the initialization function calls, the values of the control variable (e.g. $m\_VSliderBar$) becomes out-of-sync with the state of the corresponding GUI element (e.g. $IDC\_V\_SLIDER\_BAR$). The $false$ parameter to the $UpdateData()$ function call at label **B4**, flushes the control variables' value onto their corresponding GUI elements.

- Slider bar service routines (**C**): at label **C1** we declare $OnScroll()$ for servicing all *horizontal* and *vertical* scroll bars. Notice that before the service registration at label **C2**, these two are just simple functions. At label **C2**, we register for ON_WM_SCROLL (on window scroll) event. Notice that in case *no* callback functions are passed in. The horizontal and vertical scroll event services are pre-defined by the MFC $CDialog$, and in this case, we override the $OnScroll()$ functions to customize the services. The definition of event service routines can be found at labels **C3** and **C7**. Notice that if our application has more than one horizontal scroll bars, *all* horizontal scroll events will be serviced by the the same $OnHScroll()$ function. This means, as illustrated by the if statement in **C3**, when servicing horizontal scroll events we must identify which of the scroll bars triggered the event. At label **C4** we call the $GetPos()$ function on the control variable to obtain the up-to-date knob value, this new value is updated in the echo control variable at (**C5** and flushed to the corresponding GUI element at label **C6**.

H and V. To avoid repeating every sentence with an "*H*" for horizontal and an "*V*" for vertical, we *omit* the "*H/V*" characters and use one identifier to refer to both. For example, *OnScroll()* is referring to *On**H**Scroll()* and *On**V**Scroll*.

Tutorial 3.4: application defined events

- **Goal:** experience with events triggered by the application. **Approach:** Work with GUI timer event.
- **Goal:** experience working with servicing events from the mouse. **Approach:** Service all events from the mouse and echo all relevant information to the application window.

Tutorial 3.4.
Project Name
MFC_MouseAndTimer

Source File.
TutorialDlg.h/cpp file in the *Source* and $Header$ Files folders of the *MFC_MouseAndTimer* project.

```
class CTutorialDlg : public CDialog {
    ... // removed content similar to that from Listing 3.7 (Tutorial 3.3).
    A1: int m_Seconds;
    B1: CString m_MouseEcho, m_TimerEcho;
    C1: afx_msg void OnTimer(UINT nIDEvent);
    D1: afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
        afx_msg void OnMouseMove(UINT nFlags, CPoint point);
        afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
};
```

```
... // removed content similar to that from Listing 3.7 (Tutorial 3.3).
BOOL CTutorialDlg::OnInitDialog() {
    A2: m_Seconds = 0;
    C2: SetTimer(0, 1000, NULL);
...
void CTutorialDlg::DoDataExchange(CDataExchange* pDX) {
    B2: DDX_Text(pDX, IDC_MOUSEECHO, m_MouseEcho);
        DDX_Text(pDX, IDC_TIMERECHO, m_TimerEcho);
```

```
...
BEGIN_MESSAGE_MAP(CTutorialDlg, CDialog)
    C3:  ON_WM_TIMER()
    D2:  ON_WM_LBUTTONDOWN()
         ON_WM_MOUSEMOVE()
         ON_WM_RBUTTONDOWN()
...
C4: This is the timer service routine.
void CTutorialDlg::OnTimer(UINT nIDEvent) {
    m_Seconds++;      // update time passed and echo to user
    m_TimerEcho.Format("%d: Seconds have passed", m_Seconds);
    UpdateData(false);
...
D3: These are the left/right mouse button down service roune
void CTutorialDlg::OnLButtonDown(UINT nFlags, CPoint point)
void CTutorialDlg::OnRButtonDown(UINT nFlags, CPoint point)
    if(nFlags & MK_CONTROL)    // check if the control/alt/shift key is pressed
    ...
    m_MouseEcho.Format("%sLeft mouse down at %d,%d", prefix, point.x, point.y);
    UpdateData(false);
    ...
D4: This is the mouse move service routine
void CTutorialDlg::OnMouseMove(UINT nFlags, CPoint point)
```

Listing 3.8: $CTutorialDlg$ class (Tutorial 3.4).

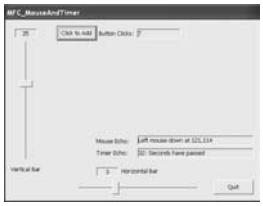

Figure 3.8: Tutorial 3.4.

Figure 3.8 is a screen shot of running Tutorial 3.4. The source code of this application is based on that from Tutorial 3.3. When compared to the main application window from Tutorial 3.3, we can see two extra sets of outputs:

- *Mouse Echo:* when the mouse pointer is inside the application window, this echo prints out the position of the mouse and the status (e.g., clicked) of the mouse buttons.
- *Timer Echo:* we will enabled the timer alarm to trigger an event for our application every second. This echo will print out the number of seconds that has passed since we start running this application.

As in all previous tutorials, all changes in programming code are localized to the $TutorialDlg.h/.cpp$ files. Listing 3.8 highlights the changes from the previous tutorial:

- Application state (**A**): since the applicaiton counts the number of elapsed seconds, we must defined (**A1**), and initialize (**A2**) a counter that we can count in *seconds*. This counter will be updated in timer service routine in **C4**.
- Mouse and timer echo (**B**): these are simple output echo set ups. As we have seen in previous tutorials, we must declare the variables (**B1**), associate the variables with the GUI IDs (**B2**). The content of these echo regions are updated during corresponding service routines, i.e., mouse and timer.
- Application timer events (**C**): label **C1** shows the declaration of the $OnTimer()$ service function. During the initialization in $OnInitDialog()$, at label **C2**, the alarm is set to go off every 1000 milliseconds (or 1 second). At label

C3, we call the ON_WM_TIMER (on window timer) macro to registration for the timer event. We do not see any callback function during the registration. Once again, the $OnTimer()$ function is defined by the MFC $CDialog$ class and we will override it to customize to our application. The timer service routine is defined at label **C4**. This function is invoked once every second. We service the timer event by incrementing the $m\_Seconds$ and echoing the new value to the defined echo area.

- Mouse events (**D**): just like timer in this tutorial, slider bars from previous tutorial, the $CDialog$ class has default support for mouse events. We know we must override the functions (**D1**); register for the events (**D2**), and implement the functions (**D3**, **D4**). Notice the mouse positions are passed in the $CPoint$ structure. If you run the tutorial, and move the mouse around in the application window, notice that the mouse positions are defined relative to the *top-left* corner (origin is located at the top-left corner).

Hardware Coordinate. The coordinate system where the *top-left* is the origin with y-axis incrementing downwards and x-axis increase rightwards.

Tutorial 3.5: Input/Output GUI Elements.

- **Goal:** experience working with GUI elements that serves both input (*from user to application*) and output (*from application to user*) functions for the application.
- **Approach:** continue with the previous tutorial, where this tutorial will allow both the application and the user to control the slider bars.

Tutorial 3.5.
Project Name
MFC_UpdateGUI

Figure 3.9: Tutorial 3.5.

Source File.
TutorialDlg.h/cpp file in the *Source* and $Header$ Files folders of the *MFC_UpdateGUI* project.

```
class CTutorialDlg : public CDialog {
    ... // removed content similar to that from Listing 3.8 (Tutorial 3.4).
    A1: BOOL m_TimerCtrlSliders;
        afx_msg void OnBnClickedTimerControlSliders();
};

... // removed content similar to that from Listing 3.8 (Tutorial 3.4).
void CTutorialDlg::DoDataExchange(CDataExchange* pDX) {
    A2: DDX_Check(pDX, IDC_TIMER_CONTROL_SLIDERS, m_TimerCtrlSliders);
...
BEGIN_MESSAGE_MAP(CTutorialDlg, CDialog)
    A3: ON_BN_CLICKED(IDC_TIMER_CONTROL_SLIDERS, OnBnClickedTimerControlSliders)
...
BOOL CTutorialDlg::OnInitDialog() {
    A4: m_TimerCtrlSliders = TRUE;
        UpdateData(false);
...
Timer service rouine: update slider bar and check-box when appropriate.
void CTutorialDlg::OnTimer(UINT nIDEvent) {
    B1: If the check-box is checked:
    if (m_TimerCtrlSliders)
        B2: int hvalue = m_HSliderBar.GetPos();
            if (hvalue > 0)
              m_HSliderBar.SetPos(hvalue-1);
              m_HSliderEcho.Format("$%$d", hvalue-1);
        // Do the same for the vertical slider bar ...
        int vvalue = m_VSliderBar.GetPos();
        ...
        B3: if ( (hvalue==0) && (vvalue==0) )
              m_TimerCtrlSliders = false;
              UpdateData(FALSE);
...
```

```
Check-box service rouine: copy state from GUI element to control variable
void CTutorialDlg::OnBnClickedTimerControlSliders() {
}   A5:   UpdateData(true);
```

Listing 3.9: $CTutorialDlg$ class (Tutorial 3.5).

Figure 3.9 is a screen shot of running Tutorial 3.5. Once again, this tutorial is very similar to that of Tutorial 3.4. From the front-end user interface, the only difference is the *Timer Control* (TC) check-box located in the center of the application window. In this case, if the TC is *checked*, the application will decrement both of the slider bar's values, one unit per second. When both slider bars has zero values, the application will *uncheck* TC. At any point, the user can check/uncheck the TC and change the slider bar values by adjusting the knobs on the slider bars. In this way, both of the slider bars, and the TC can be controlled by both the application and the user.

Listing 3.9 highlights the changes in the $TutorialDlg.h/.cpp$ files (from that of Tutorial 3.4).

- The Timer Control (TC) check-box (**A**): the variable and the event service function are declared at label **A1**; the variable $m\_TimerCtrlSliders$ becomes the control variable of $IDC\_TIMER\_CONTROL\_SLIDERS$ (the check box GUI element) at label **A2**; the check box event is registered at label **A3**; the TC value is initialized to $TRUE$ at label **A4**; and the service function is defined at label **A5**. At label **A3**, we see another example of event registration with callback function.

  It is interesting that the check box service routine at **A5** only has a single statement: $UpdateData(TRUE)$. When user *click* on the TC check box, the front end GUI will automatically flip the state of the check box GUI element. However, this information is not reflected in the control variable $m_TimerCtrlSldiers$. $UpdateData()$ with the $TRUE$ parameter sets the control variable according to the state of the GUI element.

UpdateData(FALSE). Recall from previous tutorials that *UpdateData()* with a *FALSE* parameter flushes control variable values to the GUI element.

- The application controls GUI in the timer service routine (**B**): on the per-second timer event, the timer service routine first check to ensure the sliders under the application control with the if statement at label **B1**. If the condition is favorable, at label **B2**, the slider bar positions are polled and decremented accordingly. When both of the slider bars have zero values, at label **B3**, the control variable for the TC check box is updated and flushed to the front-end GUI element (with $UpdateData(FALSE)$).

As we continue to program with GUI API, we would begin to encounter repeated patterns of working with multiple GUI element types. For example, we will find ourselves constantly working with slider bars that require the numeric echoing functionality. The next few tutorials demonstrates how we can customize and/or organize our interface with the GUI API to better support programming in a moderately complex development environment.

Tutorial 3.6: The SliderCtrlWithEcho class

Tutorial 3.6.
Project Name
MFC_SliderCtrlWithEcho

- **Goal:** demonstrate that when appropriate we should apply knowledge from previous programming classes and define/customize new GUI element classes. In most cases, this will ease the programming effort, increase readability/-maintainability of our system.
- **Approach:** customize GUI behavior by creating new GUI element types. We will customize slider bar functionality to define a new slider bar type that supports numeric echo area.

Figure 3.10: Tutorial 3.6.

Figure 3.10 is a screen shot or running Tutorial 3.6. Notice that the application behaves identical to that of Tutorial 3.5. In this case the only difference between these two applications is the support for the two slider bars. When we compare the source code of the two tutorials we see one extra class in Tutorial reftut:gut7: the $SliderCtrlWithEcho.cpp/.h$ files. This new class extends the MFC $CSliderCtrl$ class in two ways:

1. Slider bar range: $CSliderCtrl$ only supports integer values. $CSliderCtrlWithEcho$ presents a floating point range with 10^5 unique positions.
2. Numeric echo area: it is convenient, and often important, for the user to know the exact numeric value of the bar knob position. $SliderCtrlWithEcho$ supports the echoing of the bar knob numeric values.

Source File.
SliderCtrlWithEcho.h file in the $Controls$ folder of the *MFC_SliderCtrlWithEcho* project.

Source code folders. For readability, folders and sub-folders (when appropriate) are created in the *Visual Studio Solution Explorer* to organize source files according to their corresponding functionality.

```
class CSliderCtrlWithEcho : public CSliderCtrl {
    A: public interface functions
        void Initialize(float min, float max, float init);  // initialization
        bool SetSliderValue(float userValue);               // set the slider bar for output
        float GetSliderValue();                             // get user input value from the slider bar
    B: Internal representation and impelementation
        int UserToMFCPos(float userValue);          // translation from user (float) to MFC values (integer)
        float MFCToUserPos(int mfcValue);           // translation from MFC (integer) to user (float) values
        void UpdateSliderEcho();                    // update current slider value to numeric echo area
        CStatic m_MessageWnd;                       // control varaible for the echo area
    C: Override MFC event service routines
        afx_msg void HScroll(UINT nSBCode, UINT nPos);      // horizonatl scroll service routine
        afx_msg void VScroll(UINT nSBCode, UINT nPos);      // vertical scroll service routine
};
```

Listing 3.10: The $CSliderCtrlWithEcho$ class (Tutorial 3.6).

Listing 3.10 shows the definition of the $CSliderCtrlWithEcho$ class. We can see a simple public interface where the user can initialize, set, and get slider bar values. We also see familiar declarations of service routines and control variables. $Initialize()$ is the only MFC specific function where we must create and insert an echo GUI element for the numeric display. The rest of the class are fairly straightforward. Please do explore the implementations.

Tutorial 3.7: The MFC Library

Tutorial 3.7.
Project Name:
MFC_UseLibrary1
Library Support:
MFC_Library1

- **Goal:** demonstrate the advantage of collecting functional specific files into a separate software library.

- **Approach:** gather all GUI API specific functions and create a customize GUI library for our application.

The source code of Tutorial 3.7 is identical to that of Tutorial 3.6. The only difference here is in the *organization* of the source codes. When we examine the source code for Tutorial 3.7, we see that we have collected all MFC specific utilities and created the $MFC\_Library1$ software library. As the number of files grows in our development environment, creation of library to group functional specific files will become very important in maintaining a manageable source code structure. In all of our implementation, we would call functions from $MFC\_Library1$ whenever possible. In this way, we have *customized* the GUI API, where instead of calling the underlying MFC functions, we call our library functions where the support is customized to specifically support our requirements.
In general, as the *developer* of a software library we must provide:

- *Manual* and *Sample Code*: this is to support ease of use by developers using our library. The *manual* should describe the functionality and document all classes/functions in the the library. The sample code should illustrate examples of *how to use* the provided facilities in the library. In our case, the tutorials in this textbook serves as the sample code, while the explanation accompanied each tutorial serves as the manual for our libraries.

- *Header files*: this is to support *compilation* of the programming code that are based on our libraries. For example, if a programmer wants to declare a $CSliderBarWithEcho$ object, she must include the $SliderBarWithEcho.h$ file; otherwise, at compile time, the compiler will not understand what is a $CSliderBarWithEcho$. In all of our libraries, we dedicate a *library header* file that includes *all* of the classes/functions defined in the library. For example, for UWB_MFC_Lib1 the file: $uwbgl\_MFC\_Lib1.h$ includes all the definition for all the classes and functions defined in this library. A developer only needs to include this file to take advantage of facilities provided by this library.

  The advantage of *one* dedicated library header file is in its simplicity: developers only needs to know about this single file. The main disadvantage comes in the form of compilation time; including *all* definitions in a library means the compile must process much more information. We chose this approach mainly for the simplicity. In our all of our tutorial projects, the library header files are included in the $StdAfx.h$ file. Since all source files must include this pre-compile header file, all source files have access to all functionality provided by our libraries.

- *The library* this is to support *linking* of the programming code that are based on our library. In modern development environments *software libraries* typically comes in the form of a *file*. This file contains all the machine code

Library File. Library files typically ends with *.lib* or *.dll* extensions in the Microsoft Windows environment. Other examples of library extensions include, *.a*, and *.dso* in the Unix environment.

for all the functions/classes defined in the library. At link time, a developer's compiled code will be *linked* with the contents of this library file. For example, if a programmer has properly included the $SliderBarWithEcho.h$ file and declared an object for this class. At link time, the linker will locate and extract the machine code that implements the $SliderBarWithEcho$ functionality.

In modern development environment there are two types of software libraries: *statically linked* and *dynamically linked*. Statically linked libraries are processed at *link time*, where the machine code is included in the executable program. Dynamically linked libraries supports the loading of the library at *run time*. We have chosen to work with *statically linked* libraries for it simplicity.

The result of compiling our library project is a *.lib* file. For example, for the $MFC\_Library1$ project, the result of compilation is the $MFC\_Library1.lib$ library file. A developer that uses our $MFC\_Library1$ library must include this library file in the final linking of her program.

Compile results. Microsoft Visual Studio IDE store the compile results from C++ projects into the $Debug$ or the $Release$ folder.

The library we have created, $MFC\_Library1$, contains the $SliderCtrlWithEcho$ class and a utility function ($ReplaceDialogControl()$). Let's examine this function in more details.

Tutorial 3.8: Grouping of GUI Element

- **Goal:** demonstrate that sometimes it is advantages to work with a container GUI element and the corresponding programming code to organize user interface.
- **Approach:** define a "*container*" object to contain related GUI elements, define a corresponding data type and *control variable* to manage the new "*container*" object.

Tutorial 3.8.
Project Name:
MFC_GroupControls
Library Support:
MFC_Library1

Figure 3.11: Tutorial 3.8.

Figure 3.11 is a screen-shot of running Tutorial 3.8. The check box and the slider bar at the lower-left corner is meant for controlling the radius of a circle. Since we have not learn how to draw a circle, this application does not do anything. In this case, we are interested in the organization of the circle radius control GUI elements and the corresponding programming code.

Front-end GUI layout design

So far in all of the tutorials, the GUI elements are defined to be contained inside the default application window. Figure 3.12 shows the structure of the GUI elements in Tutorial 3.8. Notice that the GUI elements for controlling the circle radius are defined inside a separate container window (GUI ID: $IDD\_CONTROLS\_CHILDDLG$). As illustrated in Figure 3.12, a place holder (GUI ID: $IDC\_PLACEHOLDER$) is defined on the main application window for *placing* the *container window*.

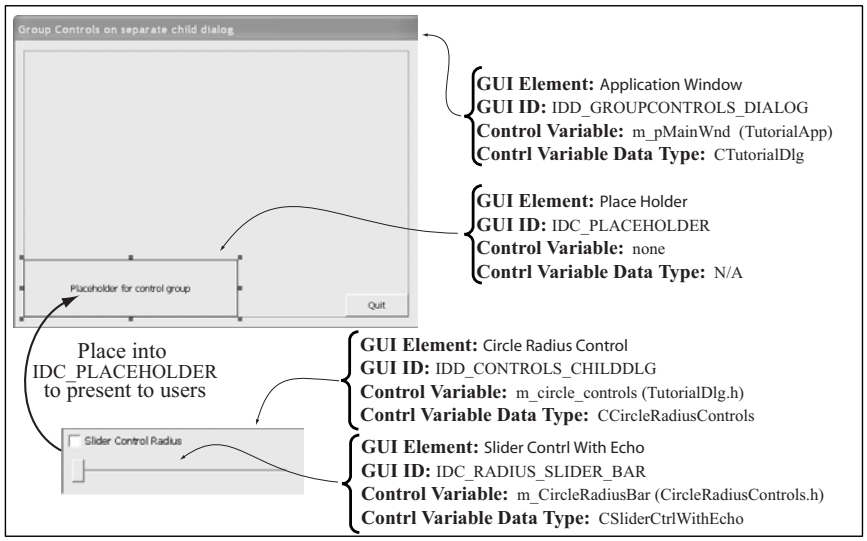

Figure 3.12: GUI Elements in the Tutorial 3.8.

Back-end Implementation

On the back-end, we must define a new data type to support the new *container window*. Notice that our main application window is also an example of *container window*, where the main application window is a GUI element and it is defined to contain other GUI elements. We have been defining the $CTutorialDlg$ class as the data type to support our main application window. When we examine the implementations (e.g., Listing 3.9 of Tutorial 3.5), we observe that $CTutorialDlg$ is a subclass of the MFC $CDialog$ class to take advantage of the vast pre-defined behaviors. Based on this experience, we can define a new data type for the new container window:

Source File. *CircleRadiusControls.h/cpp* file in the $Controls$ folder of the *MFC_GroupControls* project.

```
class CCircleRadiusControls : public CDialog {
    A1: virtual BOOL OnInitDialog();
    B1: CSliderCtrlWithEcho m_CircleRadiusBar;        // control variable for the slider bar
        BOOL m_bSliderControl;                        // control variable for the check box
    C1: afx_msg void OnBnClickedControlRadiusCheck(); // service routine for the check box
...
};
```

```
... // for the ease of reading, come code are removed (e.g. constructor, etc.)
void CCircleRadiusControls::DoDataExchange(CDataExchange* pDX) {
    B2: DDX_Control(pDX, IDC_RADIUS_SLIDER_BAR, m_CircleRadiusBar);
        DDX_Check(pDX, IDC_CONTROL_RADIUS_CHECK, m_bSliderControl);
...
BEGIN_MESSAGE_MAP(CCircleRadiusControls, CDialog)
    C2: ON_BN_CLICKED(IDC_CONTROL_RADIUS_CHECK, OnBnClickedControlRadiusCheck)
...
A2: BOOL CCircleRadiusControls::OnInitDialog() {
        B3: m_CircleRadiusBar.Initialize(0.0f, 100.0f, 10.0f);
...
```

```
C3: void CCircleRadiusControls::OnBnClickedControlRadiusCheck()
...
```

Listing 3.11: The $CCircleRadiusControls$ class (Tutorial 3.8).

Listing 3.11 shows that similar to the $CTutorialDlg$ class, the new $CCircleRadiusControls$ class is also a subclass of the MFC $CDialog$ class. This means all of our previous experiences can be applied. We observe:

- Window initialization (**A**): all *container* objects must initialize their contents. As we have learned previously, the $OnInitDialog()$ function will be invoked during the initialization of the window. In Listing 3.11, we observe the declaration (**A1**) and implementation (**A2**) of this function.

- Control variables (**B**): we need to have references to the slider bar and the check box during run time. As we have seen many times, the variables are defined at label **B1**; bound to the GUI elements at label **B2**; and initialized at label $B3$. In this tutorial we are using the slider bar data type defined in Tutorial 3.6 (**B1**), notice that even without any services, the numeric echo area reflects the knob positions correctly.

- Event services (**C**): the service of the check box event is established in familiar procedure: declaration of service routine at label **C1**; registration of event at label **C2**; and implementation at label **C3**.

With the $CCircleRadiusControls$ data type definition, we can now define control variables for the *container window* in our main window:

Source File.
TutorialDlg.h file in the $HeaderFiles$ folder of the *MFC_GroupControls* project.

```
class CTutorialDlg : public CDialog {
  ...
  CCircleRadiusControls m_circle_controls;
  ...
};
```

Listing 3.12: $CTutorialDlg$ class (Tutorial 3.8).

We have seen the definition of the two data types for the two container windows: $CCircleRadiusControls$ and $CTutorialDlg$. In addition, Listing 3.12 shows us that $CTutorialDlg$ has the $m\_circle\_controls$ control variable referencing a $CCircleRadiusControls$ window. However, we have also seen that during the *front-end* GUI layout design, we did *not* place any *CCircleRadiusControls* GUI element into the main application window area. This means, at this point, after our main application window started, the *back-end* implementation will have a control variable to a $CCircleRadiusControls$ window, however, on the *front-end* GUI there will be *no* GUI element showing the window.

At run-time, we must *place* the $m\_circle\_controls$ at the region defined by the placeholder GUI element (refer to Figure 3.12, GUI ID: $IDC\_PLACEHOLDER$). The $ReplaceDialogControl()$ function defined in our $MFC\_Library1$ is designed to accomplish this task:

Source File.
UtilityFunctions.cpp file in the $SourceFiles$ folder of the *MFC_Library1* project.

```
bool ReplaceDialogControl(CDialog& dlg, UINT placeholder_id, CDialog& new_control_group, UINT control_group_id)
  // dlg is the main application window
  // placeholder_id is ID of the place holder
  // new_control_group is control variable of the new container GUI element
  // control_group_id is ID of the new container GUI element
Places the control_group_id GUI element in the area defined by placeholder_id.
```

Listing 3.13: The $ReplaceDialogControl()$ function (Tutorial 3.8).

At run time, when $CTutorialDlg$ initializes itself in the $OnInitDialog()$ function:

Source File.
TutorialDlg.cpp file in the $SourceFiles$ folder of the *MFC_GroupControls* project.

```
BOOL CTutorialDlg::OnInitDialog() {
  ...
  ReplaceDialogControl(*this, IDC_PLACEHOLDER, m_circle_controls, IDD_CONTROLS_CHILDDLG);
      // this — is the main application window
      // IDC_PLACEHOLDER — defines the region for the circle control
      // m_circle_controls — control variable for the circle control
      // IDD_CONTROLS_CHILDDLG — GUI ID of the circle radius control window
  ...
```

Listing 3.14: $CTutorialDlg :: OnInitDialog()$ (Tutorial 3.8).

In this way, the circle control window is *replaced* into the region that was occupied by the placeholder GUI element. As we can see, defining separate GUI container window involves significant effort, however, some advantages include:

1. *Semantic mapping*: by grouping functional related GUI elements into the a container window and defining a new class representing the container window, we have created a new *user interaction object* that supports higher-level of abstraction and interaction. For example, in this tutorial we have created an *object* that is suitable for adjusting the radius of a circle. From this point on, we can work with the circle radius control object and not be concerned with slider bars and check boxes.

2. *Code organization*: instead of having a laundry list of every GUI element defined in the main application window, the main application window now contains a list of high-level interaction objects. This directly helps the organization of our source code system.

3. *Reuse*: we can instantiate multiple copies of the newly defined interaction object. For example, if I have an application with 2 circles and would like to have 2 separate radius controls for each of the circle. In this case, we can instantiate two $CircleRadiusControls$ objects to accomplish the task. In addition, it is straightforward for to resue the $CircleRadiusControls$ in another application.