In this chapter:

- *Overview of the BeOS Software Kits*
- *Software Kit Class Descriptions*
- *Chapter Example: Adding an Alert to MyHelloWorld*

3

BeOS API Overview

Writing a Be application generally involves starting with an existing base of code and then using several of the Be software kit classes to add new functionality to the base code. In Chapter 2, *BeIDE Projects*, you saw how to ready an existing project to serve as the base for your new project. In this chapter, you'll see how to select and use a software kit class to modify your new project.

This chapter begins with an overview of the Be software kits. Knowing the purpose of each kit will help you quickly hone in on which kits will be of the most use in your project. After finding a kit of interest, you need to locate a useful class within that kit. To do that, you'll use the Be Book—the electronic document by Be, Inc. that serves as the BeOS class reference. Once you've found a class of possible interest, you'll read through the Be Book's class description to find out all about the class: an overview of how objects are created, what they're useful for, and so forth. In this chapter, you'll see how to get the most out of the class descriptions in the Be Book.

The Be Book is essential documentation for any Be programmer—but it isn't a tutorial. In this chapter, I close by looking at how the Be Book describes one class (the `BAlert` class), and then go on to integrate an object of this class type in a simple program. The remaining chapters in this book provide example snippets and programs that "bring to life" the class descriptions found in the Be Book.

Overview of the BeOS Software Kits

Chapter 1, *BeOS Programming Overview*, provided a very brief description of each kit—only a sentence or two. Because you hadn't been exposed to any of the details of BeOS programming at that point, out of necessity those descriptions didn't give examples of kit classes and member functions. Now that you've

studied the fundamentals of Be programming and have looked at some example source code, it's time to rewrite the kit summaries, with an emphasis on the key classes and a few important member functions.

The focus of this book is on the first three kits described below: the Application Kit, the Interface Kit, and the Storage Kit. Don't feel as if you're being short-changed, though—these kits provide dozens of classes that allow you to create full-featured applications complete with windows, graphics, editable text, and all manner of controls.

While each of the software kits isn't represented by its own chapter in this book, all are at least briefly described below for the sake of completeness. A couple of the these kits can't be covered, as they aren't complete as of this writing. Be provides information on kit updates at the developer web page at *http://www.be.com/developers*, so you'll want to check that site occasionally. Other kits are complete, but their specialized functionality makes detailed descriptions out of scope for this book. Note that while some kits don't have a chapter devoted to them, some of their classes appear throughout the book. See the description of the Support Kit below for a specific example concerning the `BLocker` class.

Application Kit

The classes of the Application Kit communicate with the Application Server and directly with the kernel. Every program must create a single instance of a class derived from the Application Kit class `BApplication`—the HelloWorld program provides an example of how this is typically done. This `BApplication` object is necessary for a couple of reasons. The application object:

- Makes a connection to the Application Server. This connection is vital if the program is to display and maintain windows, which of course most Be programs do.
- Runs the program's main message loop. This loop provides a messaging system that keeps the program aware of events (such as a press of a keyboard key by the user).

An important member function of the `BApplication` class is `Run()`. The `main()` function of every Be program must create an instance of the `BApplication` class and then invoke `Run()` to start the program.

The `BApplication` class is derived from two other Application Kit classes—`BLooper` and `BHandler`. A `BLooper` object creates and then controls a message loop, a thread that exits to transfer messages to objects. A `BHandler` object is one that is capable of receiving a message from a `BLooper` object—it handles a message received from a message loop. Because a `BApplication` object is also a

`BLooper` and `BHandler` object, it acts as both a message loop and a message handler. Refer to Figure 1-2 in Chapter 1 for a look at the Application Kit class hierarchy that illustrates the relationship between the `BApplication` and `BLooper` and `BHandler` classes.

Interface Kit

With over two dozen classes, the Interface Kit is the largest of the Be software kits. It's also the one you'll make the most use of—as will this book. The chapters from Chapter 4, *Windows, Views, and Messages*, through Chapter 8, *Text*, deal almost exclusively with this kit. In Chapter 1 you saw that a window is an object derived from an Interface Kit class—the `BWindow` class. In Chapter 2 you were introduced to the concept that all drawing in a window is done via an object derived from another Interface Kit class—the `BView` class (much more on this important topic appears in Chapter 4 and Chapter 5, *Drawing*). In subsequent chapters you'll learn that controls (such as buttons and checkboxes), strings, and menus are also types of views (objects of classes that are derived from the `BView` class). Because all drawing takes place in a view, and because all of the aforementioned items are drawn, this should seem reasonable. It should also shed more light on the class hierarchy of the Interface Kit, as shown in Figure 1-4 back in Chapter 1.

Like a `BApplication` object (see the Application Kit above), a `BWindow` object is derived from both the `BLooper` and `BHandler` classes, so it is both an organizer of messages in a message loop and a handler of messages. When an event is directed at a window (such as a mouse button click while the cursor is over a window's close button), the system transfers a message to the window object's thread. Because the window is a message handler as well as a message loop, it may also be able to handle the message.

A window contains one or more views—objects of the `BView` class or one of its many derived classes. Often a window has one view that is the same size as the content area of the window (or larger than the content area of the window if it includes scrollbars). This view then serves as a holder of other views. These smaller, *nested*, views can consist of areas of the window that are to act independently of one another. Any one of these smaller views may also be used to display a single interface item, such as a button or a scrollbar. Because the contents of a view are automatically redrawn when a window is updated, it makes sense that each interface item exists in its own view. Some of the Interface Kit control classes that are derived from the `BView` class (and which you'll work with in Chapter 6, *Controls and Messages*) include `BCheckBox`, `BRadioButton`, and `BPictureButton`.

Storage Kit

All operating systems provide file system capabilities—without them, data couldn't be saved to disk. The Storage Kit defines classes that allow your program to store data to files, search through stored data, or both.

The `BNode` class is used to create an object that represents data on a disk. The `BFile` class is a subclass of `BNode`. A `BFile` object represents a file on disk. Creating a `BFile` object opens a file, while deleting the same object closes the file. A `BFile` object is the mechanism for reading and writing a file. The `BDirectory` class is another subclass of `BNode`. A `BDirectory` object represents a folder, and allows a program to walk through the folder's contents and create new files in the folder.

The concept of file attributes, associating extra information with a given file, allows for powerful file indexing and searching. The `BQuery` class is used to perform searches.

Support Kit

The Support Kit, as its name suggests, supports the other kits. This kit defines some datatypes, constants, and a few classes. While the nature of the classes of the Support Kit makes a chapter devoted to it impractical, you will nonetheless encounter a couple of this kit's classes throughout this book.

The `BArchivable` class defines a basic interface for storing an object in a message and instantiating a copy of that object from the message.

The `BLocker` class is used to limit program access to certain sections of code. Because the BeOS is multithreaded, there is the possibility that a program will attempt to access data from two different threads simultaneously. If both threads attempt to write to the same location, results will be unpredictable. To avoid this, programs use the `Lock()` and `Unlock()` member functions to protect code. Calls to these functions are necessary only under certain circumstances. Throughout this book mention of the use of `Lock()` and `Unlock()` will appear where required.

Media Kit

The Media Kit is designed to enable programs to work with audio and video data in real time—the kit classes provide a means for processing audio and video data. The Media Kit relies on nodes—specialized objects that perform media-related tasks. A node is always indirectly derived from the `BMediaNode` class, and there are several basic node types. Examples are producer and consumer nodes. A producer node sends output to media buffers, which are then received by consumer nodes.

Midi Kit

MIDI (Musical Instrument Digital Interface) is a communication standard for representing musical data that is generated by digital musical devices. MIDI was created to define a way for computer software and electronic music equipment to exchange information. The Midi Kit is a set of classes (such as `BMidiPort`) used to assemble and disassemble MIDI messages. A MIDI message describes a musical event, such as the playing of a note. To make use of the Midi Kit classes, you'll need to have prior knowledge of the MIDI software format.

Device Kit

The Device Kit classes (such as `BJoystick` and `BSerialPort`) are used for the control of input and output devices and for the development of device drivers. These classes serve as interfaces to the ports on the back of a computer running the BeOS.

Network Kit

The Network Kit consists of a number of C functions. The C functions are global (they can be used throughout your program), and exist to allow your program to communicate with other computers using either the TCP or UDP protocols. One such function is `gethostbyname()`, which is used to retrieve information about computers attached to the user's network.

OpenGL Kit

OpenGL is a cross-platform application programming interface developed to facilitate the inclusion of interactive 2D and 3D graphics in computer programs. Introduced in 1992, OpenGL has become the industry standard for high-performance graphics. The OpenGL Kit contains classes that simplify the implementation of animation and three-dimensional modeling in your programs. The OpenGL Kit is one of the newer BeOS kits, and is incomplete as of this writing. Working with the OpenGL classes requires some previous experience with OpenGL.

Game Kit

Like the OpenGL Kit, the Game Kit is incomplete. While it will eventually contain a number of classes that will aid in the development of games, at this time it includes just two classes. The `BWindowScreen` class is used by an application to gain direct access to the screen in order to speed up the display of graphics. The `BDirectWindow` class is an advanced class commonly used by game and media developers.

Kernel Kit

The primary purpose of the C functions that make up the Kernel Kit is to support the use of threads. While the BeOS automatically spawns and controls many threads (such as the one resulting from the creation of a new window), your program can manually spawn and control its own threads. This kit includes classes that support semaphores for protecting information in the BeOS multithreaded environment and shared memory areas for communicating between multiple threads and multiple applications.

Translation Kit

The Translation Kit provides services that ease the work in translating data from one format to another. For instance, this kit could be used to translate the data in an imported JPEG file into a `BBitmap` object (the `BBitmap` being a class defined in the Interface Kit) that your program could then manipulate.

Software Kit Class Descriptions

The classes (and in a few cases, the C functions and structures) that make up the BeOS software kits serve any imaginable programming need, yet they share many similarities. Becoming familiar with what makes up a software kit class definition and how Be documents such a class will help you make use of all of the software kits.

Contents of a Class

A Be software kit consists of classes. Each class can consist of member functions, data members, and overloaded operators. While a kit class will always have member functions, it isn't required to (and very often doesn't) have any data members or operators.

Data members

C++ programmers are used to creating classes that define a number of data members and a number of member functions. In the first few chapters of this book, though, you've read little about data members in Be classes. If a Be class does define data members, they are usually defined to be `private` rather than `public`. These `private` data members will be used within class member functions, but won't be used directly by your program's objects. That is, a data member generally exists for use in the implementation of the class rather than for direct use by your program—data members are thus of importance to a class, but they're almost never of importance to you.

You can see an example of the data members of a Be class by perusing the Be header files. In Chapter 1 you saw a snippet that consisted of a part of the `BWindow` class. In the following snippet I've again shown part of this class. Here, however, I've included the `private` keyword and some of the approximately three dozen data members that are a part of this class.

```
class BWindow : public BLooper {

public:
                   BWindow(BRect frame,
                           const char *title,
                           window_type type,
                           uint32 flags,
                           uint32 workspace = B_CURRENT_WORKSPACE);
virtual           ~BWindow();


...
         void  ResizeBy(float dx, float dy);
         void  ResizeTo(float width, float height);
virtual  void  Show();
virtual  void  Hide();
         bool  IsHidden() const;
...
private:
...
char      *fTitle;
uint32    server_token;
char      fInUpdate;
char      f_active;
short     fShowLevel;
uint32    fFlags;
port_id   send_port;
port_id   receive_port;
BView     *top_view;
BView     *fFocus;
...
}
```

Member functions

A class constructor and destructor are member functions, as are any class hook functions. While the constructor, destructor, and hook functions are often described and discussed separately from other member functions, all fall into the general category of member functions, as shown in Figure 3-1.

From programming in C++ on other platforms, you're familiar with constructors and destructors. But you may not know about hook functions. A *hook function* is a member function that can be called directly by a program, but can also be (and very often is) invoked automatically by the system.

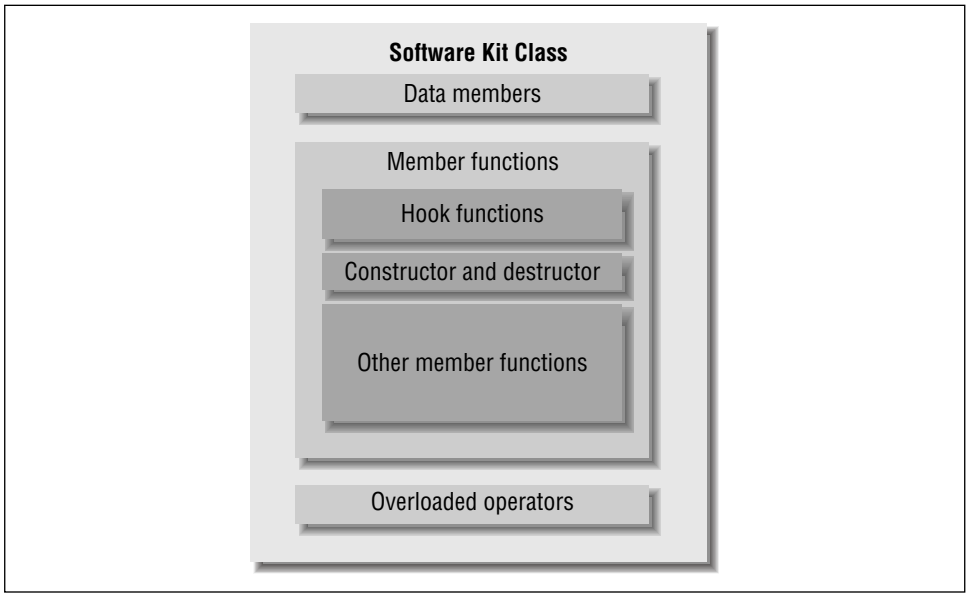

Figure 3-1. A kit class may consist of data members, member functions, and operators

Many software kit class member functions are declared using the C++ keyword `virtual`. The most common reason for declaring a member function virtual is so that a derived class can override the function. Additionally, hook functions are declared to be virtual for a second reason as well: your program may want to add functionality to that which is already provided by the hook function.

When an application-defined class defines a member function, that function is typically invoked by an object created by the application. A hook function is also a routine defined by an application-defined class, but it is one that is invoked automatically by the software kit, not by an object. In order to be called by the system, a hook function must have a specific name that the system is aware of.

You saw an example of a hook function in the SimpleApp example back in Chapter 1—the `QuitRequested()` function. When a window's close button is clicked on, the Be system automatically invokes a routine named `QuitRequested()`. If the application has defined such a function in the `BWindow`-derived class that the window object belongs to, it will be that member function that gets invoked. As a reminder, here's the `QuitRequested()` function as defined in the `SimpleWindow` class of the SimpleApp example:

```
bool SimpleWindow::QuitRequested()
{
    be_app->PostMessage(B_QUIT_REQUESTED);
    return(true);
}
```

A hook function is so named because the function serves as a place where you can "hook" your own code onto that of the Be-written software kit code. By implementing a hook function, your application in essence extends the functionality of the Be operating system. The system is responsible for calling a hook function, while your application is responsible for defining the functionality of that function.

Overloaded operators

Along with member functions and data members, you may find overloaded operators in a Be class. A few classes overload some of the C++ operators, but most classes don't overload any. You'll find that the need for a class to overload operators is usually intuitive. For instance, the `BRect` class overloads the comparison operator (`==`) so that it can be used to test for the equality of two rectangle objects. Because the comparison operator is defined in C++ such that it can be used to compare one number to another, the `BRect` class needs to rewrite its definition so that it can be used to test all four coordinates of one rectangle to the four coordinates of another rectangle.

As you just saw for the `BRect` class, if it makes sense for a class to redefine a C++ operator, it will. For most other classes, the use of operators with objects doesn't make sense, so there's no need to overload any. For instance, the `BWindow` and `BView` classes with which you're becoming familiar don't included any overloaded operators. After all, it wouldn't be easy to test if one window is "equal" to another window.

Class Descriptions and the Be Book

The definitive source of information for the many classes that make up the BeOS software kits is the Be class reference by the programmers of the BeOS. The electronic versions of this document (you'll find it in both HTML and Acrobat formats) go by the name of the Be Book, while the printed version is titled *The Be Developer's Guide* (available from O'Reilly). After programming the BeOS for awhile, you'll find the Be Book or its printed version indispensable. But now, as you take your first steps in programming the BeOS, you may find the voluminous size and the reference style of this book intimidating. While this one thousand or so page document is comprehensive and well-written, it is a class reference, not a BeOS programming tutorial. When you have a solid understanding of how classes are described in the Be Book you'll be able to use the Be Book in conjunction with this text if you wish.

The Be Book is organized into chapters. With the exception of the first chapter, which is a short introduction to the BeOS, each chapter describes the classes of one kit. Chapter 2 covers the classes of the Application Kit, Chapter 3 describes

the classes that make up the Storage Kit, and so forth. Each class description in a chapter is itself divided into up to six sections: *Overview*, *Data Members*, *Hook Functions*, *Constructor and Destructor*, *Member Functions*, and *Operators*. If any one of these six sections doesn't apply to the class being described, it is omitted from the class description. For instance, the `BWindow` class doesn't overload any operators, so its class description doesn't include an *Operators* section.

The following list provides explanations of what appears in each of the six sections that may be present in a class description in the Be Book. For each software kit class, the sections will appear in the order listed below, though some of the sections may be omitted:

Overview

A class description begins with an overview of the class. Such information as the purpose of the class, how objects of the class type are used, and related classes may be present in this section. The overview will generally be short, but for significant classes (such as `BWindow` and `BView`), it may be several pages in length.

Data Members

This section lists and describes any public and protected data members declared by the class. If a class declares only private data members (which is usually the case), this section is omitted.

Hook Functions

If any of the member functions of a class serve as hook functions, they will be listed and briefly described in this section. This section serves to summarize the purpose of the class hook functions—a more thorough description of each hook function appears in the *Member Functions* section of the class description. Many classes don't define any hook functions, so this section will be omitted from a number of class descriptions.

Constructor and Destructor

A class constructor and destructor are described in this section. A few classes don't define a destructor (objects of such class types know how to clean up and delete themselves). In such cases, this section will be named *Constructor* rather than *Constructor and Destructor*.

Member Functions

This section provides a detailed description of each of the member functions of a class, except the class constructor and destructor (which have their own section). While class hook functions have their own section, that section serves mostly as a list of hook functions—the full descriptions of such functions appear here in *Member Functions*. Every class consists of at least one member function, so this section is always present in a class description.

Operators

Here you'll find a description of any C++ operators a class overloads. Most classed don't overload any operators, so this section is frequently absent from a class description.

A BeOS Class Description: The BRect Class

Now that you've had a general look at how a class description appears in the Be Book, you'll want to see a specific example. Here I'll look at the Be Book description of the `BRect` class. Because this class doesn't have any hook functions, the *Hook Functions* section is omitted from the Be Book's class description. If you'd like to see a specific example of how a class implements hook functions, refer to the "A BeOS Class Description: The BWindow Class" section in this chapter.

As you read these pages, you may want to follow along in the electronic version of the Be Book. If you do, double-click on the Chapter 4 document and scroll to the start of the `BRect` class description.

Overview

The *Overview* section of the `BRect` class description informs you what a `BRect` object is (a rectangle) and how a `BRect` object is represented (by defining four coordinates that specify where the corners of the rectangle are located).

Next in this section is the object's general purpose (to serve as the simplest specification of a two-dimensional area) and a few specific examples of what such an object is used for (to specify the boundaries of windows, scrollbars, buttons, and so on). The `BRect` overview then provides the details of how your specification of a rectangle object's boundaries affects the rectangle's placement in a view.

As you read the overview, notice that no `BRect` data members or `BRect` member functions are mentioned by name. This is typical of a class *Overview* section; what a class object is used for is covered, but details of how to implement this usage aren't. Such details are found in the descriptions of the appropriate functions in the *Member Functions* section.

Data Members

The `BRect` class is one of the few software kit classes that declares public data members. So it is one of the few classes that includes a *Data Members* section. Here you'll find the names and datatypes of the four public data members (they're named `left`, `top`, `right`, and `bottom`, and each is of type `float`). A single-sentence description accompanies the listing of each data member. The specifics of how these data members are used by a `BRect` object appear in discussions of

the BRect constructor and member functions in the *Constructor* and *Member Functions* sections.

Hook Functions

The BRect class defines several member functions, but none of them serves as a hook function, so no *Hook Functions* section appears in the BRect class description.

Constructor and Destructor

For the BRect class, this section is named *Constructor* rather than *Constructor and Destructor*. There is no BRect destructor, which implies that a BRect object knows how to delete itself. The BRect class is somewhat atypical of kit classes in that it is more like a primitive datatype (such as an int or float) than a class. The primitive datatypes serve as the foundation of C++, and in the BeOS the BRect class serves a somewhat similar purpose; is the basic datatype that serves as the foundation of Be graphics. BRect objects are declared in the same way primitive datatype variables are declared—there's no need to use the new operator. There's also no need to use the delete operator to destroy such objects.

The *Constructor* section reveals that there are four BRect constructors. One common method of creating a rectangle is to use the constructor that accepts the four rectangle coordinates as its parameters. The remainder of the *Constructor* section of the BRect class description provides example code that demonstrates how each of the four BRect constructors can be used.

Each of the four BRect constructor functions is declared using the inline keyword. As a C++ programmer, you may have encountered inline functions. If you haven't, here's a brief summary of this type of routine. Normally, when a function is invoked, a number of non-routine instructions are executed. These instructions ensure that control is properly moved from the calling function to the called function and then back to the calling function. The execution time for these extra instructions is slight, so their inclusion is seldom a concern. If it does become an issue, though, C++ provides a mechanism for eliminating them—the inline keyword. The upside to the use of the inline keyword is that execution time decreases slightly. The downside is that this reduced time is achieved by adding more code to the executable. Declaring a function inline tells the compiler to copy the body of the function to the location at which the function is called. If the function is called several times in a program, then its code appears several times in the executable.

Member Functions

This section lists each of the `BRect` member functions in alphabetical order. Along with the name and parameter list of any given function is a detailed description of the routine's purpose and use. You've already seen one `BRect` member function in action—the `Set()` function. In the Chapter 1 example, SimpleApp, a `BRect` object is created and assigned the values that define the rectangle used for the boundaries of the SimpleApp program's one window. The *Member Functions* section's description of `Set()` states that the four parameters are used to assign values to the four `BRect` data members: `left`, `top`, `right`, and `bottom`.

Operators

The `BRect` class overrides several C++ operators in order to redefine them so that they work with operations involving rectangles. In the *Operators* section, you see each such operator listed, along with a description of how the overloaded operator is used. The operator you'll use most often is probably the assignment operator (=). By C++ definition, this operator assigns a single number to a variable. Here the `BRect` class redefines the operator such that it assigns all four coordinates of one rectangle to the four coordinates of another rectangle. The *Rect.h* header file provides the implementation of the routine that redefines the assignment operator:

```
inline BRect &BRect::operator=(const BRect& from)
{
    left = from.left;
    top = from.top;
    right = from.right;
    bottom = from.bottom;
    return *this;
}
```

A BeOS Class Description: The BWindow Class

After reading the "A BeOS Class Description: The BRect Class" section of this chapter, you should have a good feel for how classes are described in the Be Book. The `BRect` class doesn't include hook functions, though, so if you want to get a little more information on this type of member function, read this section. The `BWindow` class doesn't define any public data members or overload any operators, so you won't find talk of a *Data Members* section or an *Operators* section here. The `BRect` class, however, does define public member functions and overload operators. If you skipped the previous part, you may want to read it as well.

Following along in the Be Book isn't a requirement, but it will be helpful. If you have access to the electronic version of the Be Book, double-click on the Chapter 4 document and scroll to the start of the `BWindow` class description.

Overview

The *Overview* section of the `BWindow` class description starts by telling you the overall purpose of the `BWindow` class (to provide an application interface to windows) and the more specific purpose of this class (to enable one object to correspond to one window).

A class overview may mention other classes, kits, or servers that play a role in the workings of the described class. For instance, the `BWindow` overview mentions that the Application Server is responsible for allocating window memory and the `BView` class makes drawing in a window possible.

Before working with a class for the first time, you'll want to read the *Overview* section to get, of course, an overview of what the class is all about. But you'll also want to read the *Overview* section to pick up bits of information that may be of vital interest to your use of the class. The `BWindow` overview, for instance, notes that there is a strong relationship between the `BApplication` class and the `BWindow` class, and that a `BApplication` object must be created before the first `BWindow` object is created. The overview also mentions that a newly created `BWindow` object is hidden and must be shown using the `Show()` member function. You already knew these two facts from Chapter 1 of this book, but that chapter didn't supply you with such useful information about each of the dozens of Be software kit classes!

Data Members

Like most classes, the `BWindow` class doesn't define any public data members, so no *Data Members* section appears in the `BWindow` class description.

Hook Functions

The `BWindow` class has close to a dozen member functions that serve as hook functions—routines that are invoked by the system rather than invoked by window objects. Here you find the names of the `BWindow` hook functions, along with a single-sentence description of each. A detailed description of each function, along with parameter types, appears in the *Member Functions* section.

Many of the short descriptions of the hook functions tell why the function can be implemented. For instance, the `FrameMoved()` hook function can be implemented to take note of the fact that a window has moved. Most programs won't have to perform any special action if one of the windows is moved. If, however, your application does need to respond to a moved window, it can—thanks to the `FrameMoved()` hook function. If your application implements a version of this function, the movement of a window causes the system to automatically execute your program's version of this routine.

Constructor and Destructor

Some classes have a constructor that includes a parameter whose value comes from a Be-defined constant. In such cases the Be-defined constants will be listed and described in the *Constructor and Destructor* section. For example, the third of the `BWindow` constructor's several parameters has a value that comes from one of several Be-defined constants. This `window_type` parameter specifies the type of window to be created. Your application can use either the `B_MODAL_WINDOW`, `B_BORDERED_WINDOW`, `B_TITLED_WINDOW`, or `B_DOCUMENT_WINDOW` constant in the creation of a `BWindow` object. In this *Constructor and Destructor* section you'll find a description of each of these constants.

While a class destructor may be defined, objects of a kit class type often don't need to explicitly call the object's destructor when being destroyed. That's because the BeOS does the work. The *Constructor and Destructor* section lets you know when this is the case for a class. The `BWindow` class provides an example. As stated in the *Constructor and Destructor* section of the `BWindow` class description, a `BWindow` object is destroyed by calling the `BWindow` member function `Quit()`. This routine is responsible for using the delete operator on the window object and then invoking the `BWindow` destructor.

Member Functions

This section describes each of the more than seventy member functions of the `BWindow` class. You've worked with a couple of these routines, including the `Show()` function, in Chapter 1 and Chapter 2. If you look up `Show()` in the *Member Functions*, you'll see that `Show()` is used to display a window—as you already know. Here you'll also learn that this routine places the window in front of any other windows and makes it the active window.

Operators

The `BWindow` class doesn't overload any C++ operators, so the *Operators* section is omitted from the `BWindow` class description.

Chapter Example: Adding an Alert to MyHelloWorld

Each remaining chapter in this book will include numerous code snippets that demonstrate the several topics presented in the chapter. The chapter will then close with the source code and a walk-through of a short but comprehensive example that exhibits many or all of the chapter topics. To keep new code to a minimum and focus on only the new material presented in the chapter, each

example program will be a modification of the MyHelloWorld program introduced in Chapter 2.

This chapter provided an overview of the entire BeOS software kit layer, which makes including an example program relevant to the chapter a little difficult. Still, I'd feel uncomfortable ending a chapter without at least a short exercise in adding code to a Be application! In this section, I first rearrange some of the code in the MyHelloWorld listing to demonstrate that in Be programming—as in programming for any platform—there's more than one means to accomplish the same goal. After that, I have the new version of MyHelloWorld open both its original window and a new alert by adding just a few lines of code to the *MyHelloWorld.cpp* listing.

Revising MyHelloWorld

By this point in your studies of Be programming, you should have enough of an understanding of Be software kits, classes, and member functions that you feel comfortable making at least minimal changes to existing Be source code. While you'll often start a new programming endeavor from an existing base of code, you'll always need to adapt that code to make it fit your program's needs. You should also be gaining enough of an understanding of Be code that you feel comfortable with the idea that there is more than one way to solve any programming task. As you look at the source code of existing Be applications, be aware that different programmers will write different code to achieve the same results.

In this section I'll rearrange some of the code in the *MyHelloWindow.cpp* and *MyHelloWorld.cpp* source code files from the MyHelloWorld program introduced in Chapter 2. Building a MyHelloWorld application from the modified listings will result in a program that behaves identically to the version built in Chapter 2.

The version of MyHelloWorld presented here will be used as the basis for each of the example programs in the remainder of this book. While there's nothing tricky in the code presented here, you'll still want to take a close look at it so that you can focus on only the new code that gets added in subsequent example programs.

Two approaches to achieving the same task

While I could just go ahead and make a few alterations for the sake of change, I'll instead assume I have a valid reason for doing so! First, consider the Chapter 2 version of MyHelloWorld. The `MyHelloWindow` class has a constructor that defines an empty window. The `MyHelloView` class has a constructor that defines an empty view and a `Draw()` member function that draws the string "Hello, My

World!" in that view. Recall that there is no connection between a window object created from the `MyHelloWindow` class and a view object created from the `MyHelloView` class until after the window object is created and the view object is attached to it in the `HelloApplication` class constructor:

```
MyHelloApplication::MyHelloApplication()
    : BApplication("application/x-vnd.dps-myWorld")
{
    MyHelloWindow  *aWindow;
    MyHelloView    *aView;
    BRect           aRect;

    aRect.Set(20, 20, 200, 60);
    aWindow = new MyHelloWindow(aRect);

    aRect.OffsetTo(B_ORIGIN);
    aView = new MyHelloView(aRect, "HelloView");

    aWindow->AddChild(aView);

    aWindow->Show();
}
```

The above approach is a good one for a program that allows for the opening of windows that differ—two windows could use two different views. If MyHelloWorld defined a second view class, then a second window of the `MyHelloWindow` class type could be opened and an object of this different view could be attached to it.

Now consider a program that allows multiple windows to open, but these windows are initially to be identical. An example of such an application might be a graphics program that opens windows that each have the same tool palette along one edge. The palette could be a view object that consists of a number of icon buttons. Here it would make sense to use an approach that differs from the above. For such a program the view could be attached to the window in the window class constructor. That is, the code that is used to create a view and attach it to a window could be moved from the application constructor to the window constructor. If I were to use this approach for the MyHelloWorld program, the previously empty constructor for the `MyHelloWindow` class would now look like this:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
    MyHelloView  *aView;

    frame.OffsetTo(B_ORIGIN);
    aView = new MyHelloView(frame, "MyHelloView");

    AddChild(aView);
    Show();
}
```

For the MyHelloWorld program, the MyHelloApplication constructor would now hold less code, and would look like the version shown here:

```
MyHelloApplication::MyHelloApplication()
    : BApplication('myWD')
{
    MyHelloWindow  *aWindow;
    BRect           aRect;

    aRect.Set(20, 20, 250, 100);
    aWindow = new MyHelloWindow(aRect);
}
```

Now, when a new window is created in the application constructor, the MyHelloWindow constructor is responsible for creating a new view, adding the view to the new window, and then displaying the new window.

The new MyHelloWorld source code

Changing the MyHelloWorld program to use this new technique results in changes to two files: *MyHelloWindow.cpp* and *MyHelloWorld.cpp*. Here's how *MyHelloWindow.cpp* looks now:

```
// ------------------------------------------------------------
#ifndef _APPLICATION_H
#include <Application.h>
#endif
#ifndef MY_HELLO_VIEW_H
#include "MyHelloView.h"
#endif
#ifndef MY_HELLO_WINDOW_H
#include "MyHelloWindow.h"
#endif

MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
    MyHelloView  *aView;

    frame.OffsetTo(B_ORIGIN);
    aView = new MyHelloView(frame, "MyHelloView");

    AddChild(aView);
    Show();
}


bool MyHelloWindow::QuitRequested()
{
    be_app->PostMessage(B_QUIT_REQUESTED);
    return(true);
}

// ------------------------------------------------------------
```

The *MyHelloWorld.cpp* file doesn't get any new code—it only gets code removed. Here's the new version of *MyHelloWorld.cpp*:

```
// ------------------------------------------------------------
#ifndef MY_HELLO_WINDOW_H
#include "MyHelloWindow.h"
#endif
// removed inclusion of MyHelloView.h header file
#ifndef MY_HELLO_WORLD_H
#include "MyHelloWorld.h"
#endif

main()
{
    MyHelloApplication *myApplication;

    myApplication = new MyHelloApplication();
    myApplication->Run();

    delete(myApplication);
    return(0);
}


MyHelloApplication::MyHelloApplication()
    : BApplication('myWD')
{
    MyHelloWindow  *aWindow;
    BRect           aRect;
    // moved to MyHelloWindow constructor: MyHelloView variable declaration

    aRect.Set(20, 20, 250, 100);
    aWindow = new MyHelloWindow(aRect);
    // moved to MyHelloWindow constructor: the code to create view,
    // attach it to window, and show window
}

// ------------------------------------------------------------
```

As is the case for all of this book's examples, you'll find a folder that holds the files for this new version of MyHelloWorld on the included CD-ROM. Make sure to compile and build an application from the project file to convince yourself that this latest version of the MyHelloWorld executable is the same as the Chapter 2 version.

Adding an Alert to HelloWorld

The Interface Kit defines classes for the common interface elements. There's the `BWindow` class for a window, the `BMenuBar` class for a menubar, the `BMenu` class for a menu, the `BMenuItem` class for a menu item, and so forth. When you want to add an interface element to a program, you rely on an Interface Kit class to create

an object that represents that element. Consider a program that is to include an alert. Armed with the above knowledge, it's a pretty safe guess that the Interface Kit defines a `BAlert` class to ease the task of creating alerts.

The BAlert class description

The `BAlert` class is a simple one: it consists of a constructor and a handful of member functions. The following is the `BAlert` class definition (less its private data members, which you won't use) from the *Alert.h* header file:

```
class  BAlert : public BWindow
{
public:
                      BAlert(const char *title,
                             const char *text,
                             const char *button1,
                             const char *button2 = NULL,
                             const char *button3 = NULL,
                             button_width width = B_WIDTH_AS_USUAL,
                             alert_type type = B_INFO_ALERT);

virtual               ~BAlert();

                      BAlert(BMessage *data);
static   BArchivable  *Instantiate(BMessage *data);
virtual  status_t     Archive(BMessage *data, bool deep = true) const;

         void         SetShortcut(int32 button_index, char key);
         char         Shortcut(int32 button_index) const;

         int32        Go();
         status_t     Go(BInvoker *invoker);

virtual  void         MessageReceived(BMessage *an_event);
virtual  void         FrameResized(float new_width, float new_height);
         BButton      *ButtonAt(int32 index) const;
         BTextView    *TextView() const;

virtual  BHandler     *ResolveSpecifier(BMessage *msg,
                                        int32 index,
                                        BMessage *specifier,
                                        int32 form,
                                        const char *property);
virtual  status_t     GetSupportedSuites(BMessage *data);

static   BPoint       AlertPosition(float width, float height);
...
}
```

The *Overview* section of the `BAlert` class description in the Be Book places you on familiar ground by letting you know that an alert is nothing more than a window with some text and one or more buttons in it. In fact, the overview

states that you could forego the `BAlert` class altogether and use `BWindow` objects in their place. You *could* do that, but you won't want to. The `BAlert` class takes care of creating the alert window, establishing views within it, creating and displaying text and button objects in the alert, and displaying the alert upon creation.

The *Overview* section tells you the two steps you need to take to display an alert: construct a `BAlert` object using the new operator and the `BAlert` constructor, then invoke the `Go()` member function to display the new alert window.

The *Constructor* section of the `BAlert` class description provides the details of the variable parameter list of the `BAlert` constructor. While the constructor can have up to seven parameters, it can be invoked with fewer—only the first three parameters are required. Here's the `BAlert` constructor:

```
BAlert(const char *title,
       const char *text,
       const char *button1,
       const char *button2 = NULL,
       const char *button3 = NULL,
       button_width width = B_WIDTH_AS_USUAL,
       alert_type type = B_INFO_ALERT);
```

The required parameters specify the alert's title, its text, and a title for the alert's mandatory button. The first parameter is a string that represents the alert's title. While an alert doesn't have a tab as an ordinary window does, a title is nonetheless required. The second parameter is a string that represents the text that is to appear in the alert. The third parameter is a string (such as "OK," "Done," or "Accept") that is to be used as the title of a button that appears in the alert. This button is required so that the user has a means of dismissing the alert.

The fourth through seventh parameters have default values assigned to them so that a call to the constructor can omit any or all of them. That is, an alert can be constructed by passing just three strings to the `BAlert` constructor—illustrated in this snippet:

```
BAlert  *alert;

alert = new BAlert("Alert", "Close the My Hello window to quit", "OK");
```

The fourth and fifth parameters are optionally used for the text of up to two more buttons that will appear in the alert. The number of buttons in an alert varies depending on the number of strings passed to the `BAlert` constructor. By default, the second and third button titles are `NULL`, which tells `BAlert` to place only a single button in the alert. A Be-defined constant can be used as the sixth parameter to `BAlert()`. This constant specifies the width of the alert's buttons. By default, the width of each button will be a standard size (`B_WIDTH_AS_USUAL`) that is capable of holding most button titles. Providing a different constant here changes the width of each button. The final parameter specifies the icon to be displayed in

the upper-left of the alert. By default a lowercase "i" (for "information") is used (as denoted by the Be-defined `B_INFO_ALERT` constant).

The Alert program

In the *C03* folder on this book's CD-ROM, you'll find two folders: *MyHelloWorld* and *Alert*. The *MyHelloWorld* folder holds the new version of the MyHelloWorld project—the version just described. The *Alert* folder also holds a version of the MyHelloWorld project. The only difference between the two projects appears in the *MyHelloApplication* constructor in the *MyHelloWorld.cpp* file of the alert example. In the alert example, I've added four lines of code, shown in bold type here:

```
MyHelloApplication::MyHelloApplication()
    : BApplication('myWD')
{
    MyHelloWindow  *aWindow;
    MyHelloView    *aView;
    BRect           aRect;
    BAlert         *alert;
    long            result;

    aRect.Set(20, 20, 250, 100);
    aWindow = new MyHelloWindow(aRect);

    alert = new BAlert("", "Close the My Hello window to quit.", "OK");
    result = alert->Go();
}
```

This latest incarnation of MyHelloWorld will serve as the base from which the remainder of this book's examples are built. To make it easy to recognize the code that doesn't change from example to example (which is most of the code), I won't rename the files and classes for each example. That means you'll always find the familiar `MyHelloView`, `MyHelloWindow`, and `HelloApplication` classes. As exhibited in the above snippet, the differences between one version of MyHelloWorld and another will consist of minimal code changes or additions. My goal in this book is to present short examples that consist mostly of existing, thoroughly discussed code, with only a minimum amount of new code. The new code will be, of course, code that is pertinent to the topic at hand. To distinguish one MyHelloWorld project from the next, I'll simply rename the project folder to something descriptive of the example. For instance, the alert example discussed here appears in a folder titled *Alert*.

In the above call to `BAlert()`, the first parameter is an empty string that represents the alert's title. An alert doesn't have a tab as a window does, but a title is required. Passing an empty string suffices here. The second parameter is a string

that represents the text that is to appear in the alert. While this trivial example exists only to show how the alert code integrates into a program, I did want the alert to have at least some bearing on the program—so I've made the alert serve as a means of letting the user know how to quit the program. The final parameter is the name that appears on the alert's button. For a one-button alert, OK is typically used. Figure 3-2 shows what the alert example program looks like when running.

Figure 3-2. An alert overlapping a window