In this chapter:
- *Features of the BeOS*
- *Structure of the BeOS*
- *Software Kits and Their Classes*
- *BeOS Programming Fundamentals*
- *BeOS Programming Environment*

1

BeOS Programming Overview

A few years back, the Macintosh operating system was considered innovative and fun. Now many view it as dated and badly in need of a rewrite rather than a simple upgrade. Windows 95 is the most popular operating system in the world—but this operating system is in many ways a copy of the Mac OS, less the Mac's character. Many programmers and computer enthusiasts enjoy the command-line interface power of Unix—but Unix isn't nearly intuitive enough for the average end user. What users really want is an operating system that has an easy-to-use graphical user interface, takes advantage of the power of today's fast microprocessor chips, and is unencumbered with the burdens of backward compatibility. Enter Be, Inc., and the BeOS—the Be operating system.

In this introductory chapter, you'll learn about the features of the BeOS from a programmer's perspective. In particular, you'll read about the terminology relating to the Be operating system. You'll also get an overview of the layout of the application programming interface, or API, that you'll be using to aid you in piecing together your programs. After the overview, you'll look at some of the fundamentals of writing applications for the BeOS. No attempt will be made to supply you with a full understanding of the concepts, techniques, and tricks of programming for this operating system—you've got the whole rest of the book for that! Instead, in this chapter I'll just give you a feel for what it's like to write a program for the BeOS. Finally, this chapter concludes with a first look at Metrowerks CodeWarrior—the integrated development environment you'll be using to develop your own applications that run on the BeOS.

Features of the BeOS

With any new technology comes a plethora of buzzwords. This marketing hype is especially true in the computer industry—innovative software and hardware seem

to appear almost daily, and each company needs some way to ensure that the public views their product as the best. Unsurprisingly, the BeOS is also accompanied by a number of buzzwords—multithreaded, multiprocessor support, and preemptive multitasking being a few. What may be surprising is that this nomenclature, when applied to BeOS, isn't just hype—these phrases really do define this exciting operating system!

Multithreaded

A *thread* is a path of execution—a part of a program that acts independently from other parts of the program, yet is still capable of sharing data with the rest of program. An OS that is *multithreaded* allows a single program to be divided into several threads, with each thread carrying out its own task. The processor devotes a small amount of time first to one thread and then to another, repeating this cycle for as long as it takes to carry out whatever task each thread is to perform. This parallel processing allows the end user to carry out one action while another is taking place. Multithreading doesn't come without a price—though fortunately in the BeOS this price is a rather small one. A program that creates multiple threads needs to be able to protect its data against simultaneous access from different threads. The technique of locking information when it is being accessed is one that is relatively easy to implement in BeOS programs.

The BeOS is a multithreaded operating system—and a very efficient one. While programmers can explicitly create threads, much of the work of handling threads is taken care of behind the scenes by the operating system itself. For instance, when a window is created in a program, the BeOS creates and maintains a separate thread for that one window.

Multiprocessor Support

An operating system that uses multithreading, designed so that threads can be sent to different processors, is said to use *symmetric multiprocessing*, or SMP. On an SMP system, unrelated threads can be sent to different processors. For instance, a program could send a thread that is to carry out a complex calculation to one processor and, at the same time, send a thread that is to be used to transfer a file over a network to a second processor. Contrasting with symmetric multiprocessing (SMP) is *asymmetric multiprocessing*, or AMP. A system that uses AMP sends a thread to one processor (deemed the master processor) which in turn parcels out subtasks to the other processor or processors (called the slave processor or processors).

The BeOS can run on single-processor systems (such as single-processor Power Macintosh computers), but it is designed to take full advantage of machines that

have more than one processor—it uses symmetric multiprocessing. When a Be program runs on a multiprocessor machine, the program can send threads to each processor for true parallel processing. Best of all, the programmer doesn't need to be concerned about how to evenly divide the work load. The Be operating system is responsible for distributing tasks among whatever number of processors are on the host machine—whether that be one, two, four, or more CPUs.

The capability to run different threads on different processors, coupled with the system's ability to assign threads to processors based on the current load on each processor, makes for a system with very high performance.

Preemptive Multitasking

An operating system that utilizes *multitasking* is one that allows more than one program to run simultaneously. If that operating system has *cooperative multitasking*, it's up to each running program to yield control of system resources to allow the other running applications to perform their chores. In other words, programs must cooperate. In a cooperative multitasking environment, programs can be written such that they don't cooperate graciously—or even such that they don't cooperate at all. A better method of implementing multitasking is for an operating system to employ preemptive multitasking. In a *preemptive multitasking* environment the operating system can, and does, preempt currently running applications. With preemptive multitasking, the burden of passing control from one program to another falls on the operating system rather than on running applications. The advantage is that no one program can grab and retain control of system resources.

If you haven't already guessed, the BeOS has preemptive multitasking. The BeOS microkernel (a low-level task manager discussed later in this chapter) is responsible for scheduling tasks according to priority levels. All tasks are allowed use of a processor for only a very short time—three-thousandths of a second. If a program doesn't completely execute a task in one such *time-slice*, it will pick up where it left off the next time it regains use of a processor.

Protected Memory

When a program launches, the operating system reserves an area of RAM and loads a copy of that program's code into this memory. This area of memory is then devoted to this application—and to this application only. While a program running under any operating system doesn't intentionally write to memory locations reserved for use by other applications, it can inadvertently happen (typically when the offending program encounters a bug in its code). When a program writes outside of its own address space, it may result in incorrect results or an aborted program. Worse still, it could result in the entire system crashing.

An operating system with *protected memory* gives each running program its own memory space that can't be accessed by other programs. The advantage to memory protection should be obvious: while a bug in a program may crash that program, the entire system won't freeze and a reboot won't be necessary. The BeOS has protected memory. Should a program attempt to access memory outside its own well-defined area, the BeOS will terminate the rogue program while leaving any other running applications unaffected. To the delight of users, their machines running BeOS rarely crash.

Virtual Memory

To accommodate the simultaneous running of several applications, some operating systems use a memory scheme called virtual memory. Virtual memory is memory other than RAM that is devoted to holding application code and data. Typically, a system reserves hard drive space and uses that area as virtual memory. As a program executes, the processor shuffles application code and data between RAM and virtual memory. In effect, the storage space on the storage device is used as an extension of RAM.

The BeOS uses virtual memory to provide each executing application with the required memory. For any running application, the system first uses RAM to handle the program's needs. If there is a shortage of available physical memory, the system then resorts to hard drive space as needed.

Less Hindered by Backward Compatibility

When a company such as Apple or Microsoft sets about to upgrade its operating system, it must take into account the millions of users that have a large investment in software designed to run on the existing version of its operating system. So no matter how radical the changes and improvements are to a new version of an operating system, the new OS typically accommodates these users by supplying backward compatibility.

Backward compatibility—the ability to run older applications as well as programs written specifically for the new version of the OS—helps keep the installed base of users happy. But backward compatibility has a downside: it keeps an upgrade to an operating system from reaching its true potential. In order to keep programs that were written for old technologies running, the new OS cannot include some new technologies that would "break" these existing applications. As a new operating system, the BeOS had no old applications to consider. It was designed to take full advantage of today's fast hardware and to incorporate all the available modern programming techniques. As subsequent releases of the BeOS are made available, backward compatibility does become an issue. But it will be quite a while

before original applications need major overhauling (as is the case for, say, a Macintosh application written for an early version of the Mac OS).

Structure of the BeOS

Be applications run on hardware driven by either Intel or PowerPC microprocessors (check the BeOS Support Guides page at *http://www.be.com/support/guides/* for links to lists of exactly which Intel and PowerPC machines are currently supported). Between the hardware and applications lies the BeOS software. As shown in Figure 1-1, the operating system software consists of three layers: a microkernel layer that communicates with the computer's hardware, a server layer consisting of a number of servers that each handle the low-level work of common tasks (such as printing), and a software kit layer that holds several software kits—shared libraries (known as dynamically linked libraries, or DLLs, to some programmers) that act as a programmer's interface to the servers and microkernel.

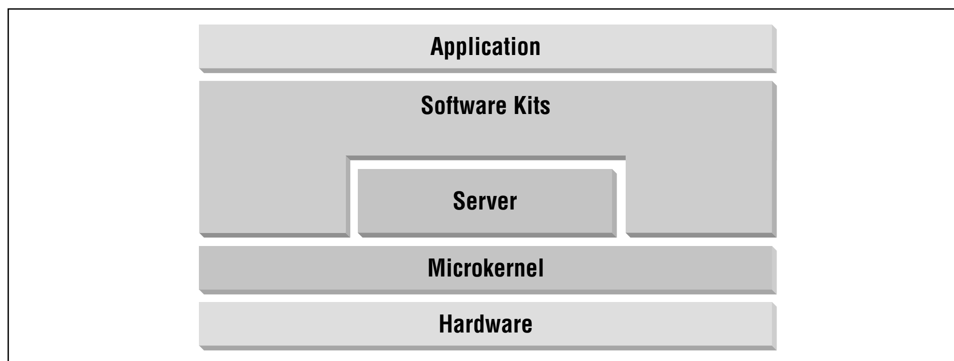

Figure 1-1. The layers of the BeOS reside between applications and hardware

Microkernel

The bottom layer consists of the microkernel. The microkernel works directly with the hardware of the host machine, as well as with device drivers. The code that makes up the microkernel handles low-level tasks critical to the control of the computer. For instance, the microkernel manages access to memory. The kernel also provides the building blocks that other programs use: thread scheduling, the file system tools, and memory-locking primitives.

Servers

Above the microkernel lies the server layer. This layer is composed of a number of servers—processes that run in the background and carry out tasks for applications that are currently executing. For example, the purpose of the Input Server is

to handle access to all the various keyboards, mice, joysticks, and other input devices that may be connected to a machine running the BeOS. Another server is the Application Server, a very important server that handles the display of graphics and application communication. As a programmer you won't work directly with servers; instead, you'll rely on software kits to access the power of the server software.

Kits

Above the server layer is the software kit layer. A kit consists of a number of object-oriented classes that a programmer makes use of when writing a BeOS program. Collectively the classes in the software kits comprise the BeOS API. You know that the abbreviation *API* stands for *application programming interface*. But what does the application interface to? Other software. For Be applications, the kits are the interface to the various servers. For instance, the Application Kit holds several classes used by programmers in your position who are trying to create tools for users. The programmer writes code that invokes methods that are a part of the classes of the Application Kit, and the Application Kit then communicates with the Application Server to perform the specified task. A couple of the other servers you'll encounter in your Be programming endeavors are the Print Server and the Media Server.

Some kits don't rely on servers to carry out microkernel-related operations—the chores they take care of may be simple and straightforward enough that they don't need their own server software. Instead, these kits directly invoke microkernel code. As you can see in Figure 1-1, an application relies directly on the software kits and indirectly on the servers and microkernel.

As you become more proficient at BeOS programming, you'll also become more intimate with the classes that comprise the various software kits. Now that you know this, you'll realize that it is no accident that the majority of this book is devoted to understanding the purpose of, and working with, the various BeOS kits.

This book is tutorial in nature. Its purpose is to get you acquainted with the process of developing applications that run on the BeOS and to provide an overview of the BeOS API. Its purpose *isn't* to document the dozens of classes and hundreds of member functions that make up the BeOS API. After—or while—reading this book, you may want such a reference. If you do, consider the books *Be Developer's Guide* and *Be Advanced Topics*, also by O'Reilly & Associates.

Software Kits and Their Classes

The application programming interface of the BeOS is object-oriented—the code that makes up the software kits is written in C++. If you have experience programming in C++ on any platform, you're already at the midpoint in your journey to becoming adept at BeOS programming. Now you just need to become proficient in the layout and use of the classes that make up the software kits.

Software Kit Overview

The BeOS consists of about a dozen software kits—the number is growing as the BeOS is enhanced. Don't panic, though—you won't be responsible for knowing about all the classes in all of the kits. Very simple applications require only the classes from a very few of the kits. For instance, an application that simply displays a window that holds text uses the Application Kit and the Interface Kit. A more complex application requires more classes from more kits. Presentation software that stores sound and video data in files, for example, might require the use of classes from the Storage Kit, the Media Kit, and the Network Kit—as well as classes from the two previously mentioned kits. While it's unlikely that you'll ever write a program that uses all of the BeOS kits, it's a good idea to at least have an idea of the purpose of each.

The kits of the BeOS are subject to change. As the BeOS matures, new functionality will be added. This functionality will be supported by new classes in existing kits and, perhaps, entirely new software kits.

Application Kit

The Application Kit is a small but vitally important kit. Because every application is based on a class derived from the `BApplication` class that is defined in this kit, every application uses the Application Kit.

The Application Kit defines a messaging system (described later in this chapter) that makes applications aware of events (such as a click of a mouse button by the user). This kit also give applications the power to communicate with one another.

Interface Kit

The Interface Kit is by far the largest of the software kits. The classes of this kit exist to supply applications with a graphical user interface that fully supports user interaction. The definition of windows and the elements that are contained in windows (such as scrollbars, buttons, lists, and text) are handled

by classes in this kit. Any program that opens at least one window uses the Interface Kit.

Storage Kit
The Storage Kit holds the classes that store and update data on disks. Programs that work with files will work with the Storage Kit.

Support Kit
As its name suggests, the contents of the Support Kit support the other kits. Here you'll find the definitions of datatypes, constants, and a few classes. Because the Support Kit defines many of the basic elements of the BeOS (such as the Boolean constants `true` and `false`), all applications use this kit.

Media Kit
The Media Kit is responsible for the handling of real-time data. In particular, this kit defines classes that are used to process audio and video data.

Midi Kit
The Midi Kit is used for applications that process MIDI (Musical Instrument Digital Interface) data.

Kernel Kit
The Kernel Kit is used by applications that require low-level access to the BeOS microkernel. This kit defines classes that allow programmers to explicitly create and maintain threads.

Device Kit
The Device Kit provides interfaces to hardware connectors (such as the serial port), and is necessary only for programmers who are developing drivers.

Network Kit
The Network Kit exists to provide TCP/IP services to applications.

OpenGL Kit
The OpenGL Kit provides classes that allow programmers to add 3D capabilities to their programs. The classes aid in the creation and manipulation of three-dimensional objects.

Translation Kit
The Translation Kit is useful when a program needs to convert data from one media format to another. For instance, a program that can import an image of one format (such as a JPEG image) but needs to convert that image to another format might make use of this kit.

Mail Kit
The Mail Kit assists in adding Internet email services (such as sending messages using Simple Mail Transfer Protocol (SMTP) to an application).

Game Kit

The Game Kit—which is under development as of this writing—consists of two major classes that support game developers.

BeOS Naming Conventions

Some of the many classes that make up the BeOS are discussed a little later. As they're introduced, you'll notice that each starts with an uppercase letter "B," as in `BMessage`, `BApplication`, and `BControl`. This is no accident, of course—the software of the kits follows a naming convention.

The BeOS software kits consist of classes (which contain member functions and data members), constants, and global variables. The BeOS imposes a naming convention on each of these types of elements so that anyone reading your code can readily distinguish between code that is defined by the BeOS and code that is defined by your own program. Table 1-1 lists these conventions.

Table 1-1. BeOS Naming Conventions

Category	Prefix	Spelling	Example
Class name	B	Begin words with uppercase letter	`BRect`
Member function	none	Begin words with uppercase letter	`OffsetBy()`
Data member	none	Begin words (excepting the first) with uppercase letter	`bottom`
Constant	B_	All uppercase	`B_LONG_TYPE`
Global variable	be_	All lowercase	`be_clipboard`

Classes of the BeOS always begin with an uppercase "B" (short for "BeOS", of course). Following the "B" prefix, the first letter of each word in the class name appears in uppercase, while the remainder of the class name appears in lowercase. Examples of class names are `BButton`, `BTextView`, `BList`, and `BScrollBar`.

Member functions that are defined by BeOS classes have the first letter of each word in uppercase and the remainder of the function name in lowercase. Examples of BeOS class member function names are `GetFontInfo()`, `KeyDown()`, `Frame()`, and `Highlight()`.

Data members that are defined by BeOS classes have the first letter of each word in uppercase and the remainder of the data member name in lowercase, with the exception of the first word—it always begins in lowercase. Examples of BeOS class data member names are `rotation` and `what`.

I've included only a couple of examples of data member names because I had a hard time finding any! Be engineers went to great lengths to hide data members. If you peruse the Be header files you'll find a number of data members—but most are declared private and are used by the classes themselves rather than by you, the programmer. You'll typically make things happen in your code by invoking member functions (which themselves may access or alter private data members) rather than by working directly with any data members.

Constants defined by BeOS always begin with an uppercase "B" followed by an underscore. The remainder of the constant's name is in uppercase, with an underscore between words. Examples include: `B_WIDTH_FROM_LABEL`, `B_WARNING_ALERT`, `B_CONTROL_ON`, and `B_BORDER_FRAME`.

The BeOS software includes some global variables. Such a variable begins with the prefix "be_" and is followed by a lowercase name, as in: `be_app`, `be_roster`, and `be_clipboard`.

Software Kit Inheritance Hierarchies

The relationships between classes of a software kit can be shown in the *inheritance hierarchy* for that kit. Figure 1-2 shows such an inheritance hierarchy for the largest kit, the Interface Kit.

The kits that make up the BeOS don't exist in isolation from one another. A class from one kit may be derived from a class defined in a different kit. The `BWindow` class is one such example. Kits serve as logical groupings of BeOS classes—they make it easier to categorize classes and conceptualize class relationships.

Figure 1-2 shows that the object-oriented concept of inheritance—the ability of one class to inherit the functionality of another class or classes—plays a very large role in the BeOS. So too does multiple inheritance—the ability of a class to inherit from multiple classes. In the figure, you see that almost all of the Interface Kit classes are derived from other classes, and that many of the classes inherit the contents of several classes. As one example, consider the six control classes pictured together in a column at the far right of Figure 1-2. An object of any of these classes (such as a `BButton` object) consists of the member functions defined in that class as well as the member functions defined by all of the classes from which it is directly and indirectly derived: the `BControl`, `BInvoker`, `BView`, `BHandler`, and

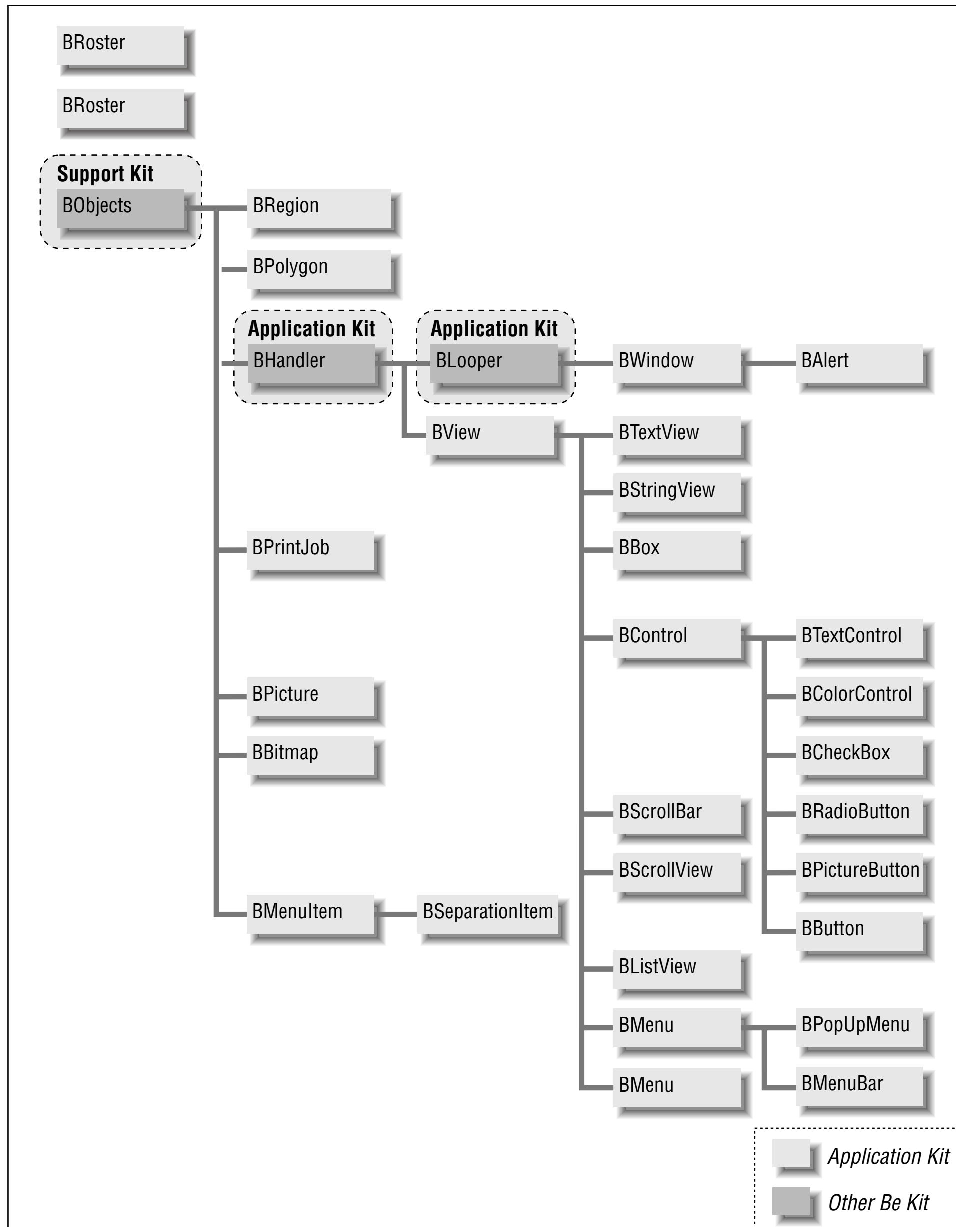

Figure 1-2. The inheritance hierarchy for the Interface Kit

`BArchivable` classes. Figure 1-3 isolates the discussed classes for emphasis of this point. This figure shows that in inheritance hierarchy figures in this book, a class pictured to the left of another class is higher up in the hierarchy. In Figure 1-3, `BView` is derived from `BHandler`, `BControl` is derived from `BView`, and so forth.

Understanding the class hierarchies of the BeOS enables you to quickly determine which class or classes (and thus which member functions) you will need to use to implement whatever behavior you're adding to your program. Obviously, knowledge of the class hierarchies is important. Don't be discouraged, though, if the hierarchies shown in Figures 1-2 and 1-3 don't make complete sense to you. This chapter only provides an overview of the object-oriented nature of the BeOS. The remainder of the book fills in the details of the names, purposes, and uses of the important and commonly used classes.

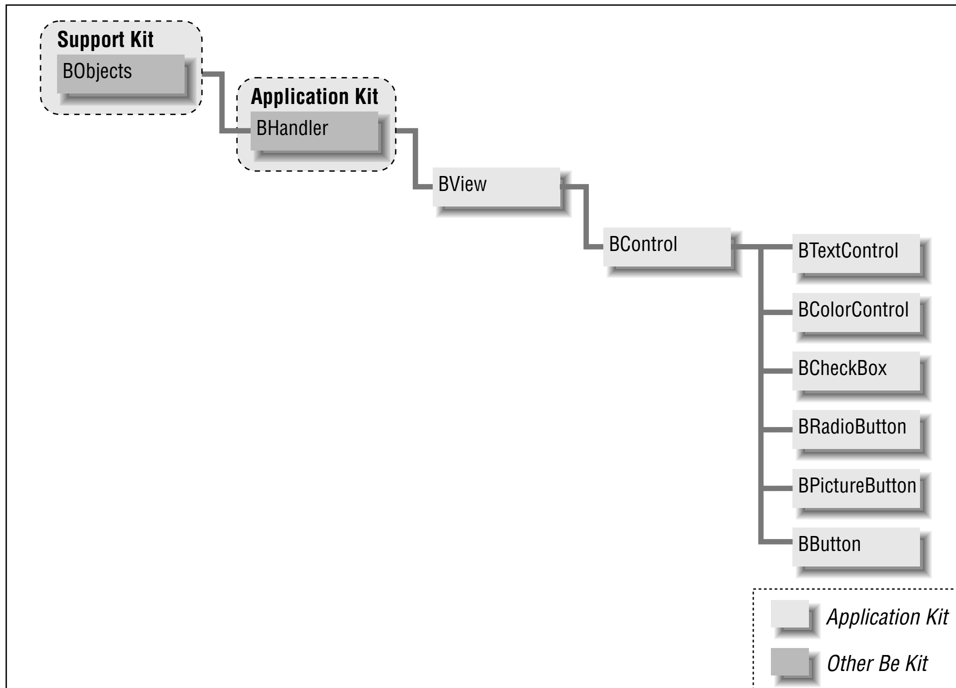

Figure 1-3. The Interface Kit classes that contribute to the various control classes

The `BControl` class defines member functions that handle the needs common to any type of control. For instance, a control should be able to have two states: enabled or disabled. An enabled control is active, or usable by the user. A disabled control is inactive—and has a greyed-out look to let the user know it is unusable. To give controls the ability to implement this behavior, the `BControl` class includes the `SetEnabled()` member function. This routine is used to enable or disable a control—any kind of control. Individual types of controls will have some needs that aren't common to all other types of controls and thus can't be

implemented by the `BControl` class. For example, different controls (such as buttons and checkboxes) have different looks. To make it possible for each control type to be able to draw itself, each control class defines its own constructor to initialize the control and a `Draw()` member function to handle the drawing of the control.

Not all BeOS classes are derived from other classes—there are a few classes that don't rely on inheritance. Two examples, both of which happen to be in the Interface Kit, are the `BRect` and `BPoint` classes. The `BRect` class is used to create objects representing rectangles. A rectangle is an easily defined, two-dimensional shape that's considered a basic datatype. As such, it doesn't need to inherit the functionality of other classes. The `BPoint` class is not a derived class for the same reason.

BeOS Programming Fundamentals

In the previous section, you gained an understanding of how the BeOS is composed of numerous interrelated classes that are defined in software kits. Together these classes form an application framework from which you build your Be applications. Your program will create objects that are based on some of the BeOS classes. These objects will then communicate with one another and with the operating system itself through the use of messages. In this section, you'll look at a few of the most important of these classes, and you'll see how they're used. You'll also see how messages play a role in a BeOS program. To make the transition from the theoretical to the practical, I'll supply you with a few C++ snippets—as well as the code for a complete Be application. In keeping with the introductory nature of this chapter, I'll make this first application a trivial one.

Messages, Threads, and Application Communication

Earlier in this chapter, you read that the BeOS is a multithreaded operating system. You also read that the term *multithreaded* isn't just bandied about by BeOS advocates for no good reason—it does in fact play a key role in why the BeOS is a powerful operating system. Here, you'll get an introduction as to why that's true. In Chapter 4, *Windows, Views, and Messages*, I'll have a lot more to say about multithreading.

Applications and messages

A Be application begins with the creation of an object of a class type derived from the `BApplication` class—a class defined in the Application Kit. Figure 1-4 shows

how the BApplication class fits into the inheritance hierarchy of the Application Kit. Creating an application object establishes the application's main thread, which serves as a connection between the application and the Application Server. Earlier in this chapter, you read that a BeOS server is software that provides services to an application via a software kit. The Application Server takes care of many of the tasks basic to any application. One such task is reporting user actions to applications. For instance, if the user clicks the mouse button or presses a key on the keyboard, the Application Server reports this information to executing applications. This information is passed in the form of a message, and is received by an application in its main thread. A *message* is itself an object—a parcel of data that holds details about the action being reported. The ability of the operating system to determine the user's actions and then use a separate thread to pass detailed information about that action to a program makes your programming job easier.

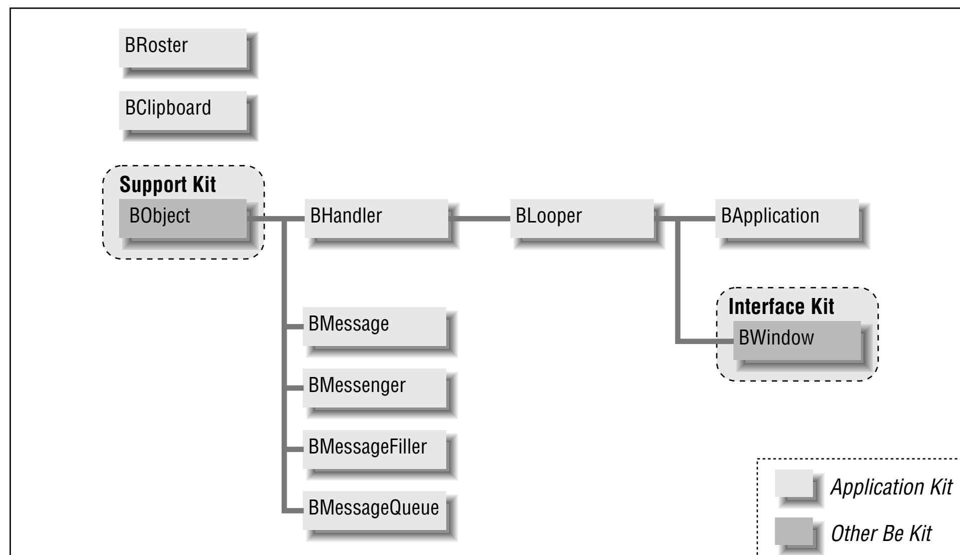

Figure 1-4. The inheritance hierarchy for the Application Kit

An application's code can explicitly define BMessage objects and use them to pass information. What I've discussed above, however, are *system messages* that originate from within the BeOS itself. The movement of the mouse, the pressing of a keyboard key, a mouse button click in a window's close button, and a mouse button click and drag in a window's resize knob are all examples of system messages. Each type of system message has a command constant associated with it. This constant names the type of event the message represents. Examples of command constants are B_KEY_DOWN, B_MOUSE_DOWN, and B_WINDOW_RESIZED.

Message loops and message handling

The BeOS defines classes that allow the creation of objects that can work with messages. The Application Kit defines two such classes: the `BLooper` class and the `BHandler` class. The `BLooper` class is used to create an object that exists in its own thread. The purpose of this thread is to run a *message loop*. As messages reach a message loop thread, they are placed in a queue. From this queue the thread extracts and dispatches messages one after another.

A message is always dispatched to an object of the `BHandler` class. The job of the `BHandler` object is to handle the message it receives. How it handles a message is dependent on the type of message it receives.

As shown back in Figure 1-4, the `BLooper` class is derived from the `BHandler` class. This means that an object of the `BLooper` class (or of a class derived from `BLooper`) can have both a message loop that dispatches messages and can receive these messages itself for handling. Because the `BApplication` class and the `BWindow` class are derived from the `BLooper` class, such is the case for the application itself and any of its windows. Just ahead you'll read a little more on how an application and windows can in fact watch for and respond to messages.

To summarize, a `BLooper` object has a thread that runs a message loop that dispatches messages, and a `BHandler` object receives and handles these dispatched messages. Because the `BLooper` class is derived from the `BHandler` class, a `BLooper` object can dispatch and receive and handle messages. A `BHandler` object can only receive and handle messages. From that description it might seem that all objects that deal with messages might as well be `BLooper` objects. After all, the `BLooper` class provides more functionality. As you read more about messaging, you'll see why that path isn't the one to take. Each `BLooper` object creates a new thread and dominates it with a message loop—the thread shouldn't be used for any other purpose. A `BHandler` object, on the other hand, doesn't create a thread. While having multiple threads in a program can be advantageous, there's no benefit to creating threads that go unused.

Defining and Creating Windows

At the heart of the graphical user interface of the Be operating system is the window. Be applications are window-based—windows are used to accept input from the user by way of menus and controls such as buttons, and to display output to the user in the form of graphics and text. The Interface Kit—the largest of the kits—exists to enable programmers to provide their Be applications with a graphical user interface that includes windows. It is classes of the Interface Kit that you'll be using when you write a program that displays and works with windows.

The BWindow class

Almost all Be applications display at least one window and therefore use the BWindow class—one of the dozens of classes in the Interface Kit. If you look in the *Window.h* header file that is a part of the set of header files used in the compilation of a Be program, you'll find the declaration of the BWindow class. I've included a partial listing (note the ellipses) of this class below. Here you can see the names of a dozen of the roughly one hundred member functions of that class. Looking at the names of some of the member functions of the BWindow class gives you a good indication of the functionality the class supplies to BWindow objects.

```
class BWindow : public BLooper {

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

virtual  void        Quit();
         void        Close();

virtual  void        DispatchMessage(BMessage *message, BHandler *handler);
virtual  void        MessageReceived(BMessage *message);
virtual  void        FrameMoved(BPoint new_position);
...
virtual  void        Minimize(bool minimize);
virtual  void        Zoom(BPoint rec_position, float rec_width, float rec_
height);
...
         void        MoveBy(float dx, float dy);
         void        MoveTo(BPoint);
         void        MoveTo(float x, float y);
         void        ResizeBy(float dx, float dy);
         void        ResizeTo(float width, float height);
virtual  void        Show();
virtual  void        Hide();
         bool        IsHidden() const;
...
         const char  *Title() const;
         void        SetTitle(const char *title);
         bool        IsFront() const;
         bool        IsActive() const;

...
}
```

If you're interested in viewing the entire BWindow class declaration, you can open the *Window.h* header file. The path that leads to the *Window.h* file will most likely be *develop/headers/be/interface*. There's a good chance that your development environment resides in your root directory, so look for the *develop* folder there. You can open any header file from the Edit text editor application or from the BeIDE. The Metrowerks CodeWarrior BeIDE programming environment is introduced later in this chapter and discussed in more detail in Chapter 2, *BeIDE Projects*.

Deriving a class from BWindow

A Be program that uses windows could simply create window objects using the BWindow class. Resulting windows would then have the impressive functionality provided by the many BWindow member functions, but they would be very generic. That is, while they could be moved, resized, and closed (BWindow member functions take care of such tasks), they would have no properties that made them unique from the windows in any other application. Instead of simply creating a BWindow object, programs define a class derived from the BWindow class. This derived class, of course, inherits the member functions of the BWindow class. Additionally, the derived class defines new member functions and possibly overrides some inherited member functions to give the class the properties that windows of the application will need. The following snippet provides an example:

```
class SimpleWindow : public BWindow {

public:
                    SimpleWindow(BRect frame);

virtual  bool       QuitRequested();
};
```

From the BeOS naming conventions section of this chapter, you know that the name of a class that is a part of the BeOS API (such as BWindow) always starts with an uppercase "B." As long as my own classes (such as SimpleWindow) don't start with an uppercase "B," anyone reading my code will be able to quickly spot classes that are of my own creation.

The SimpleWindow constructor

The SimpleWindow class declares a constructor and one member function. The definition of the constructor follows.

```
SimpleWindow::SimpleWindow(BRect frame)
    : BWindow(frame, "A Simple Window", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
}
```

This constructor makes use of a technique common in Be applications: the constructor for the class derived from the `BWindow` class invokes the `BWindow` class constructor. Calling the `BWindow` class constructor is important because the `BWindow` constructor arguments provide important information to the window object that is to be created. In Chapter 5, *Drawing*, I discuss the four `BWindow` constructor parameters in detail. In this introduction, it will suffice for me to say that the four parameters specify the following for a newly created window object:

- The frame, or content area of the window (the size and screen placement of the window)
- The name of the window (as it will appear in the window's tab)
- The type of the window (the look and feel of the window)
- The behavior of the window (whether it has a resize knob, and so forth)

Recall from your C++ background that when the definition of a constructor is followed by a single colon and the base class constructor, the effect is that the base class constructor gets invoked just before the body of the derived class constructor executes.

In this example, the `BWindow` constructor's first argument comes from the sole argument passed to the `SimpleWindow` constructor. A hardcoded string serves as the second argument to the `BWindow` constructor. The third and fourth arguments are constants defined in the *Window.h* header file.

Notice that the body of the `SimpleWindow` constructor is empty. This tells you that the only chore of the `SimpleWindow` constructor is to invoke the `BWindow` constructor. You have to call the `BWindow` constructor; this function creates a new window and spawns a new thread of execution in which the window runs, and starts up a message loop in that same thread. In a Be program, each window exists in its own thread and each window is notified of system messages that involve the window. You'll be pleased to find that the work of maintaining a window's thread and of keeping a window informed of system messages (such as a mouse button click in the window) is taken care of by the operating system. You'll be even more pleased to find that for some system messages, even the window's response to the message is handled by the BeOS. For instance, you needn't write any code that watches for or handles the resizing of a window.

A window can watch for and respond to messages because the BWindow class inherits from both the BLooper and BHandler classes (see Figure 1-4). A window is thus a window (from BWindow), an object that includes a message loop (from BLooper), and an object that responds to messages (from BHandler). This pertains to BWindow objects and, of course, objects created from classes derived from the BWindow class—such as objects of my SimpleWindow class type.

The SimpleWindow QuitRequested() member function

The SimpleWindow class declares one member function. Here's the definition of QuitRequested():

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

QuitRequested() is actually a member function of the BLooper class. Because my SimpleWindow class is derived from the BWindow class, which in turn is derived from the BLooper class, this member function is inherited by the SimpleWindow class. By redeclaring QuitRequested(), SimpleWindow is overriding this function.

If I had opted *not* to override the QuitRequested() member function in the SimpleWindow class, it would be the BLooper version of this function that would execute upon a user mouse button click in a window's close button. Like my SimpleWindow version of QuitRequested(), the version of QuitRequested() defined by the BLooper class returns a value of true. The effect is for an object of BLooper type to kill the thread it is running in and delete itself. That sounds much like what I'd like to do in response to the user's attempt to close a window—kill the thread in which the window is running. And it is. But in my trivial example program, I'll only be allowing a single window to appear on the screen. When the user closes that window, I'll want to terminate the application, not just the window. That's the action I've added to the QuitRequested() function with this line of code:

```
be_app->PostMessage(B_QUIT_REQUESTED);
```

A mouse button click in a window's close button generates a system message that gets passed to the window. The window is a type of BLooper, so it captures messages in its message loop. A window is also a type of BHandler, so it can handle this message (as opposed to having to pass it to some other type of object for handling). It handles the message by invoking QuitRequested(). If my SimpleWindow class didn't override the BLooper version of this function, the BLooper version would be executed and the window would close—but the

application wouldn't quit. That's because the BLooper version only kills its own thread in order to delete itself. Because SimpleWindow does override QuitRequested(), it is the SimpleWindow version of this function that instead gets invoked. The SimpleWindow version posts a B_QUIT_REQUESTED message to the application to tell the application to also quit. The notation used in the above line (be_app->PostMessage()) is new to you, so it's worthy of examination.

You already know that a window is a type of BLooper, but there is another very important type of BLooper: the application itself. An application is always represented by an application object—an object of the BApplication class that is defined in the Application Kit (refer back to Figure 1-4 if you need to verify the relationship between the BLooper class and the BWindow and BApplication classes). The PostMessage() routine is a member function of the BLooper class. A BLooper object can invoke this function to place a message in the queue of its own message loop.

As you'll see ahead, be_app is a global variable that represents the application object. This variable is always available for use by your code. The above line of code invokes the application object's version of the PostMessage() function. The message the application object places in its message loop is one that tells itself to quit.

The variable be_app is a pointer to an object—the use of the membership access operator (->) to invoke PostMessage() tells you that. As is often the case in object-oriented programming, a pointer to an object is simply referred to as the object itself. So in this book, as well as in other Be documentation, you'll read about the "application object" in discussions that include mention of be_app.

After the call to PostMessage() places a request to kill the application thread in the application object's message queue, the SimpleWindow version of QuitRequested() returns a value of true. Remember, QuitRequested() won't be called by my own code—it will be invoked by the system in response to the mouse button click in a window's close button. By returning a value of true, QuitRequested() is telling the system that the requested service should be carried out. The system will then kill the window thread to dispose of the window.

Previously I mentioned that the BeOS took care of system messages involving a window. I gave the example of window resizing being handled by the operating system. Yet here I'm discussing how my own code is being used to handle the system message that gets generated by a click in a window's close button. It's important to restate what I just discussed. It wouldn't be necessary to include any

window-closing code in my `SimpleWindow` class if my goal was only to have a mouse button click in the close button result in the closing of the window. The `QuitRequested()` function defined in `BLooper` would take care of that by killing the window's thread. I, however, also want the program to terminate when a window's close button is clicked. To get that extra action, I need to override `QuitRequested()`.

In summary, a mouse button click in a window's close box automatically causes `QuitRequested()` to execute. If a window class doesn't override this function, the window closes but the application continues to run. If the window class does override this function, what happens is determined by the code in this new version of the function. In my `SimpleWindow` class example, I choose to have this function tell the application to quit and tell the window to close.

Creating a window object

Declaring a class and defining its constructor and member functions only serves to specify how objects of this class type will look and behave—it doesn't actually create any such objects. To create and display a window object you'll first declare a variable that will be used to point to the object:

```
SimpleWindow  *aWindow;
```

Before going ahead and allocating the memory for a new window object, your code should declare and set up a rectangle object that will serve to establish the size and screen placement of the new window:

```
BRect   aRect;

aRect.Set(20, 20, 200, 60);
```

The above snippet first declares and creates a rectangle object. The `BRect` class was briefly mentioned earlier in this chapter—it is discussed at length in Chapter 6, *Controls and Messages*. Next, the `Set()` member function of the `BRect` class is called to establish the dimensions of the rectangle. The `Set()` function's four parameters specify the left, top, right, and bottom coordinates, respectively.

The above call to `Set()` establishes a rectangle that will be used to create a window with a left side 20 pixels in from the left of the screen and a top 20 pixels down from the top of the screen. While the window that will use this rectangle would seem to have a width of 180 pixels (200–20) and a height of 40 pixels (60–20), it will actually have a width of 181 pixels and a height of 41 pixels. This apparent one-pixel discrepancy is explained in the screen and drawing coordinates section of Chapter 5.

With the window's bounding rectangle established, it's time to go ahead and create a new window. This line of code performs that feat:

```
aWindow = new SimpleWindow(aRect);
```

To dynamically allocate an object, use the `new` operator. Follow `new` with the constructor of the class from which the object is to be created. If you glance back at the section that describes the `SimpleWindow` constructor, you'll be reminded that this function has one parameter—a `BRect` object that defines the size of the window and gets passed to the `BWindow` constructor.

After allocating memory for a `SimpleWindow` object, the system returns a pointer to this memory. That pointer is stored in the `aWindow` variable. Until this new window is deleted, it can be accessed via this pointer. This line of code provides an example:

```
aWindow->Show();
```

By default, a newly created window is not visible. To display the window, your code should call the `BWindow` member function `Show()` as I've done in the above line.

Let's end this section by pulling together the code that's just been introduced. Here's a look—with comments—at how a window is typically created in a Be application:

```
SimpleWindow  *aWindow;             // declare a pointer to a SimpleWindow
                                    // object
BRect          aRect;               // declare a rectangle object

aRect.Set(20, 20, 200, 60);         // specify the boundaries of the
                                    // rectangle
aWindow = new SimpleWindow(aRect);  // create a SimpleWindow object

aWindow->Show();                    // display the newly created window
```

You may have noticed that I used the `new` operator to create a window object, but created a rectangle object without a `new` operator. In Be programs, objects can always be, and sometimes are, allocated dynamically. That is, the `new` operator is used to set aside memory in the program's heap—as I've done with the window. Some objects, however, are allocated statically. That is, an object variable (rather than a pointer) is declared in order to set aside memory on the stack—as I chose to do with the rectangle. Creating an object that resides on the stack is typically done for objects that are temporary in nature. In the above snippet, the rectangle object fits that bill—it exists only to provide values for the window's dimensions. After the creation of the window, the rectangle object is unimportant.

If you aren't familiar with the term *heap,* I should explain that it is an area in a program's address space that exists to hold objects that are created dynamically during the execution of a program. An object can be added or deleted from the heap without regard for its placement in the heap, or for the other contents of the heap. The *stack*, on the other hand, is used to store objects in a set order—objects are stacked one atop the other. Objects can only be added and removed from the top of the stack.

Defining an Application

Every Be program must create an object that represents the application itself. This one object is the first one created when a program launches and the last one deleted when the program quits. One of the primary purposes of the application object is to make and maintain a connection with the Application Server. It is the Application Server that takes care of the low-level work such as handling interactions between windows and monitoring input from data entry sources such as the keyboard and mouse.

The BApplication class

To create the application object, you first define a class that is derived from the `BApplication` class and then create a single instance of that class (an *instance* being nothing more than another name for an *object*). From the *Application.h* header file, here's a partial listing of the `BApplication` class:

```
class BApplication : public BLooper {

public:
                        BApplication(const char * signature);
virtual                 ~BApplication();
...
virtual  thread_id      Run();
virtual  void           Quit();
...
         void           ShowCursor();
         void           HideCursor();
...
}
```

Referring back to Figure 1-4, you can see that the `BApplication` class is both a type of `BLooper` and a type of `BHandler`. This means that an application object is capable of having a message loop, and is capable of handling messages in that loop. As it turns out, the application object runs the application's main message loop. It is this loop that receives messages that affect the application.

Deriving a class from BApplication

Every application defines a single class derived from the `BApplication` class. A program that will be communicating with other programs may define a number of member functions to handle this interapplication communication. A simpler application might define nothing more than a constructor, as shown here:

```
class SimpleApplication : public BApplication {

public:
                SimpleApplication();
};
```

The SimpleApplication constructor

When a Be program starts, it's common practice for the program to open a single window without any help from the user. Because the `SimpleApplication()` constructor executes at program launch (that's when the application object is created), it would make sense to let this constructor handle the job of creating and displaying a window. Here's a look at how the constructor does that:

```
SimpleApplication::SimpleApplication()
    : BApplication("application/x-vnd.dps-simpleapp")
{
    SimpleWindow  *aWindow;
    BRect          aRect;

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

    aWindow->Show();
}
```

Just as my `SimpleWindow` class invoked the constructor of the class it was derived from (the `BWindow` class), so does my `SimpleApplication` class invoke the constructor of the class it is derived from (the `BApplication` class). Invoking the `BApplication` constructor is necessary for a few reasons. The `BApplication` constructor:

- Connects the application to the Application Server
- Provides the application with a unique identifying signature for the program
- Sets the global variable `be_app` to point to the new application object

The connecting of an application to the Application Server has already been mentioned. This connection allows the server to send messages to the application. The application signature is a MIME (Multipurpose Internet Mail Extensions) string. The phrase *application/x-vnd.* should lead off the signature. Any characters you want can follow the period, but convention states that this part of the MIME string consist of an abbreviation of your company's name, a hyphen, and then part or all of

the program name. In the above example, I've used my initials (*dps*) as the company name. I've elected to name my program SimpleApp, so the MIME string ends with *simpleapp*. The assignment of an application's signature is described at greater length in Chapter 2. The global variable `be_app` was introduced in the earlier discussion of windows. This variable, which is always available for your program's use, always points to your program's `BApplication` object.

In the "Creating a window object" section that appears earlier, you saw five lines of code that demonstrated how a window object could be created and how its window could be displayed on the screen. If you compare the body of the `SimpleApplication()` constructor to those five lines, you'll see that they are identical.

Creating an application object

After defining a class derived from the `BApplication` class, it's time to create an application object of that class type. You can create such an object dynamically by declaring a pointer to the class type, then using the `new` operator to do the following: allocate memory, invoke a constructor, and return a pointer to the allocated memory. Here's how that's done:

```
SimpleApplication  *myApplication;

myApplication = new SimpleApplication();
```

After the above code executes, `myApplication` can be used to invoke any of the member functions of the `BApplication` class (from which the `SimpleApplication` class is derived). In particular, soon after creating an application object, your Be program will invoke the `BApplication Run()` member function:

```
myApplication->Run();
```

The `Run()` function kicks off the message loop in the application's main thread, and then it begins processing messages. Not only is a call to this function important, it's necessary; an application won't start running until `Run()` is invoked.

A program's application object is typically declared in `main()`, and is accessed by the global variable `be_app` outside of `main()`. So there's really no need to have the application object reside in the heap—it can be on the stack. Here's how the creation of the application object looks when done using a variable local to `main()`:

```
SimpleApplication  myApplication;

myApplication.Run ();
```

This second technique is the way the application object will be created in this book, but you should be aware that you may encounter code that uses the first technique.

The main() Routine

The preceding pages introduced the C++ code you'll write to create an application object and start an application running. One important question remains to be answered, though: where does this code go? Because the application object must be created immediately upon application launch (to establish a connection to the Application Server), it should be obvious that this code must appear very early in the program. A C++ program always begins its execution at the start of a routine named `main()`, so it shouldn't come as a surprise that it is in `main()` that you'll find the above code. Here's a look at a `main()` routine that is typical of a simple Be application:

```
int  main()
{
    SimpleApplication   myApplication;

    myApplication.Run();

    return(0);
}
```

To start a program, call `Run()`. When the user quits the program, `Run()` completes executing and the program ends. You'll notice in the above snippet that between `Run()` and `return`, there is no code. Yet the program won't start and then immediately end. Here's why. The creation of the application object (via the declaration of the `BApplication`-derived object `myApplication`) initiates the program's main thread. When `Run()` is then called, the `Run()` function takes control of this main application thread. `Run()` sets up the main message loop in this main thread, and controls the loop and thread until the program terminates. That is, once called, `Run()` executes until the program ends.

The SimpleApp Example Program

The preceding pages have supplied you with all the code you need to write a Be application—albeit a very simple one. Because this same code (or slight variations of it) will appear as a part of the source code for every Be program you write, I've gone to great lengths to explain its purpose. In trying to make things perfectly clear, I'll admit that I've been a bit verbose—I've managed to take a relatively small amount of starter code and spread it out over several pages. To return your focus to just how little code is needed to get a Be program started, I've packaged the preceding snippets into a single source code listing. When compiled and

linked, this source code becomes an executable named SimpleApp. When launched, the SimpleApp program displays a single, empty window like the one shown in Figure 1-5.

Figure 1-5. The window that results from running the SimpleApp program

The SimpleApp source code listing

Presented next, in its entirety, is the source code for a Be application named SimpleApp. As mentioned, all of the code you're about to see has been presented and discussed earlier in this chapter.

```
#include <Window.h>
#include <Application.h>

class SimpleWindow : public BWindow {

public:
                    SimpleWindow(BRect frame);
    virtual  bool   QuitRequested();
};


SimpleWindow::SimpleWindow(BRect frame)
    : BWindow(frame, "A Simple Window", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
}


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


class SimpleApplication : public BApplication {

public:
                SimpleApplication();
};


SimpleApplication::SimpleApplication()
    : BApplication("application/x-vnd.dps-simpleapp")
{
    SimpleWindow  *aWindow;
    BRect          aRect;
```

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

    aWindow->Show();
}


main()
{
    SimpleApplication   myApplication;

    myApplication.Run();

    return(0);
}
```

What the SimpleApp program does

When you launch SimpleApp you'll see the window pictured in Figure 1-5. You can click the mouse button while the cursor is positioned over the zoom button in the window's tab to expand the window to a size that fills most of your monitor. Click the mouse button with the cursor again positioned over the window's zoom button and the window will return to its previous, much smaller, size. If you click and hold the mouse button while the cursor is positioned over the window's tab, you can drag the window about the monitor. Most important to this discussion is that the SimpleApp source code includes no code to handle such tasks. The zooming and moving of windows is handled by the system, not by the SimpleApp code. This simple demonstration emphasizes the power of the BeOS system software—it is the system software code (rather than the application code) that supplies much of the functionality of a program.

What the SimpleApp program doesn't do

There are a number of things SimpleApp doesn't do—things you'd expect a "real" Be application to do. Most notable of these omissions are menus, support of input by way of controls in the window, and support of output via drawing or writing to the window. Of course these omissions will be rectified in the balance of this book. Starting, in fact, with the next chapter.

BeOS Programming Environment

The programming tool you'll be using to create your Be applications is the BeIDE. This piece of software is an integrated development environment (IDE) and it runs on the Be operating system (so the origin of the name BeIDE is pretty evident!).

The development of a new program entails the creation of a number of files which, collectively, are often referred to as a *project*. Taking a look at an existing

project is a good way to get an overview of the files that make up a project, and is also of benefit in understanding how these same files integrate with one another. In Chapter 2 I do just that. There you'll see the HelloWorld example that's the mainstay of getting introduced to a new programming language or platform. In that chapter, you'll also see how an existing project (such as HelloWorld) can be used as the basis for an entirely new program. As a prelude to Chapter 2's in-depth coverage of this project, take a look at Figures 1-6 and 1-7.

Figure 1-6. The files used in the development of the HelloWorld application

The */boot/apps/Metrowerks* folder holds the BeIDE itself, along with other folders that hold supporting files and projects. Figure 1-6 shows the contents of a folder that holds two projects, both of which are used for building a standalone HelloWorld application. The project named *HelloWorld_ppc.proj* is used to build a Be application that executes on a PowerPC-based machine running the BeOS, while the project named *HelloWorld_x86.proj* is used to build a Be application that executes on an Intel-based PC. In Figure 1-6 you see that a project consists of a number of files. The filename extensions provide a hint of the types of files that make up any one project. A file with an extension of:

.cpp
: Is a C++ source code file

.h
: Is a header file that holds definitions used by certain C++ source code files

.rsrc

Is a resource file that holds resources that get merged with compiled source code

.proj

Is a project file that is used to organize the files used by the project

Also shown in the HelloWorld folder in Figure 1-6 is a makefile—appropriately named *makefile*. The BeIDE programming environment supports creation of programs from the command line. That is, you can supply the BeIDE compiler and linker with information by editing a makefile and then running that file from the BeOS Terminal application. In this book I'll forego the command-line approach to application development and instead rely on the BeIDE's project-based model. As you'll see in Chapter 2, creating a project file to serve as a means of organizing the files used in a project takes full advantage of the Be graphical user interface. Figure 1-7 shows the window that appears when you use the BeIDE to open the project file for the HelloWorld project.

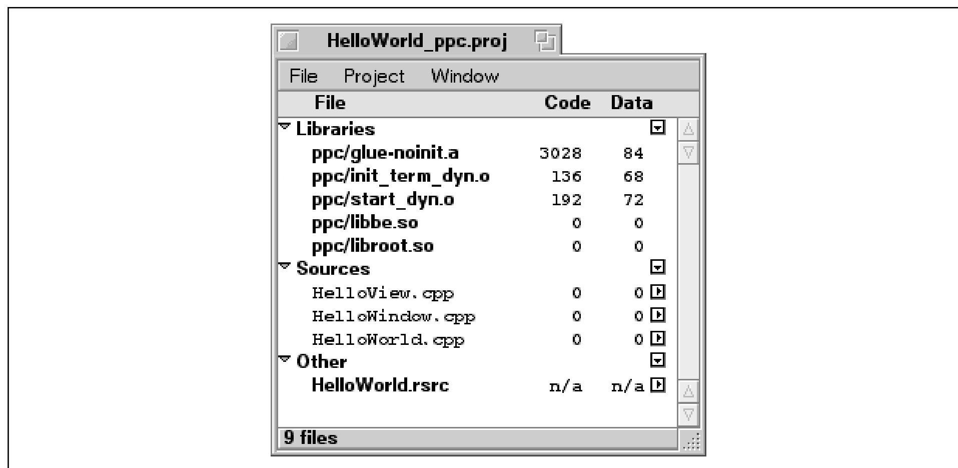

Figure 1-7. The project window for the HelloWorld project

You use a project file as a sort of command center for one project. From this one window, you add and remove source code files, libraries, and resource files from the project. You can also double-click on a source code filename in the project window to open, and then edit, that file. Using menu items from the File, Project, and Window menus in the menubar of the project window, you can perform a myriad of commands—including compiling source code and building an application.