In this chapter:
- ***Development Environment File Organization***
- ***Examining an Existing BeIDE Project***
- ***Setting Up a New BeIDE Project***
- ***HelloWorld Source Code***

2
BeIDE Projects

The BeOS CD-ROM includes the BeIDE—Be's integrated development environment (IDE) that's used for creating Be applications. This programming environment consists of a number of folders and files, the most important of which are mentioned in this chapter. In the early stages of your Be programming studies, the folder of perhaps the most interest is the one that holds sample code. Within this folder are a number of other folders, each holding a Be-supplied project. A project is a collection of files that, when compiled, results in a single Be application. The best way to understand just what a project consists of is to take a long look at an existing Be project. That's exactly what I do in this chapter.

After examining an existing project, you'll of course want to create your own. A large part of this chapter is devoted to the steps involved in doing that. Here you'll see how to organize classes into header files and source code files, and how the resource file fits into the scheme of things.

Development Environment File Organization

You'll find that an overview of how the many BeIDE items are organized will be beneficial as you look at existing BeIDE example projects and as you then start to write your own BeOS program.

The BeIDE Folders

When the BeIDE is installed on your hard drive, the folders and files that make up this programming environment end up in a pair of folders named *develop* and *apps* on your boot drive.

The /boot/develop folder

In the *develop* folder you'll find folders that hold header files, libraries, and developer tools. Figure 2-1 shows the contents of the *develop* folder (on a PowerPC-based machine—a BeOS installation on an Intel-based machine results in one additional folder, the *tools* folder). This figure also shows the *apps* folder. The *apps* folder holds over a dozen items, though in Figure 2-1 you just see a single item (the *Metrowerks* folder, discussed later).

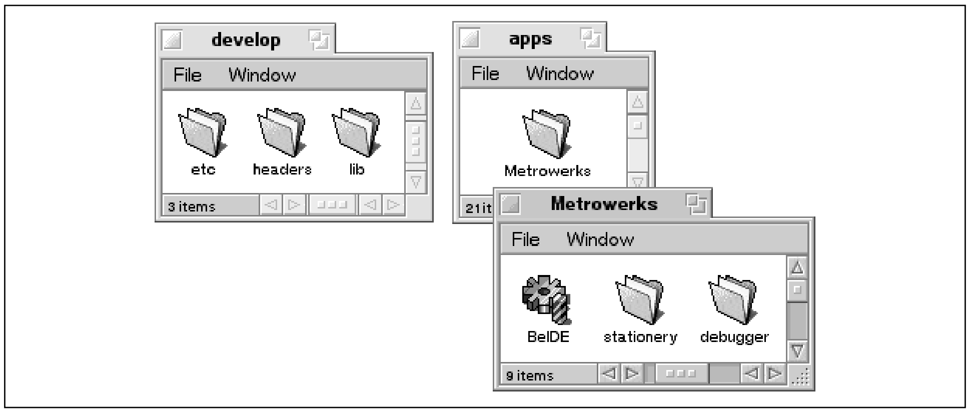


Figure 2-1. Some of the key folders and files used in BeOS programming

In the *develop* folder the *lib* folder holds a number of library files that can be linked to your own compiled code. The act of creating a BeIDE project (discussed later) automatically handles the adding of the basic libraries (*libroot.so* and *libbe.so* at this writing) to the project. As a novice Be programmer, this automatic adding of libraries to a new project is beneficial—it shields you from having to know the details of the purpose of each library. As you become proficient at programming for the BeOS, though, you'll be writing code that makes use of classes not included in the basic libraries—so you'll want to know more about the libraries included in the *develop/lib* folder. Of course you could simply add libraries wholesale to a project to "play it safe," but that tack would be a poor one—especially for programmers developing BeOS applications that are to run on Intel machines. On Intel, all libraries in a project will likely be linked during the building of an application—even if the program uses no code from one or more of the project's libraries. The resulting application will then be unnecessarily large, or will include dependencies on libraries that are not needed.

The *develop* folder *headers* holds the header files that provide the BeIDE compiler with an interface to the software kits. Within the *headers* folder is a folder named *be*. Within that folder you'll find one folder for each software kit. In any one of these folders are individual header files, each defining a class that is a part of one kit. For instance, the `BWindow` class is declared in the *Window.h* header file in the *interface* folder. The complete path to that file is */boot/develop/headers/be/interface/Window.h*.

The *etc* folder in the *develop* folder contains additional developer tools. As of this writing, the primary component in this folder is files used by programmers who prefer a makefile alternative to BeIDE projects. To build an application without creating a BeIDE project, copy the *makefile* template file from this folder to the folder that holds your source code files. Then edit the copied makefile to include the names of the files to compile and link. In this book, I'll focus on the BeIDE project model, rather than the makefile approach, for creating an application.

The *tools* folder in the *develop* folder is found only on Intel versions of the BeOS. This folder contains the x86 (Intel) compiling and linking tools and the debugger.

The /boot/apps/Metrowerks folder

Of most interest in the */boot/apps* folder is the *Metrowerks* folder. The BeIDE was originally an integrated development environment that was created and distributed by a company named Metrowerks. Be, Inc. has since taken over development and distribution of the BeIDE. Though Be now owns the BeIDE, installation of the environment still ends up in a folder bearing Metrowerks' name.

In the *Metrowerks* folder can be found the BeIDE application itself. The BeIDE is the Be integrated development environment—to develop an application, you launch the BeIDE and then create a new project or open an existing one.

Also in the *Metrowerks* folder are a number of subdirectories that hold various supporting files and tools. The *plugins* folder holds BeIDE plugins that enhance the capabilities of the BeIDE. The *stationery* folder contains the basic stationery used in the creation of a new BeIDE project (stationery being a file that tells the BeIDE which files (such as which libraries) to include, and what compiler and linker settings to use in a new project). The *tools* folder contains the compiler and linker (on the PowerPC version of the BeOS) or links to the compiler and linker (on the Intel version of the BeOS). On the PowerPC version of the BeOS, you'll find a couple of other folders in the Metrowerks folder: the *debugger* folder (which holds the PowerPC debugger, of course) and the *profiling* folder (which holds some PowerPC profiling tools).

The sample-code folder

Included on the BeOS CD-ROM, but not automatically placed on your hard drive during the installation of the BeOS, is the *sample-code* folder. If you elected to have optional items included during the BeOS installation, this folder may be on your hard drive. Otherwise, look in the *optional* folder on the BeOS CD-ROM for the *sample-code* folder and manually copy it to your hard drive.

The *sample-code* folder holds a number of Be-provided projects. Each project, along with the associated project files, is kept in its own folder. A Be application starts out as a number of files, including source code files, header files, and a resource file (I have much more to say about each of these file types throughout this chapter).

Examining an Existing BeIDE Project

The development of a new program entails the creation of a number of files collectively called 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. Because my intent here is to provide an overview of what a project consists of (as opposed to exploring the useful and exciting things that can be accomplished via the code within the files of a project), I'll stick to staid and familiar ground. On the next several pages I look at the HelloWorld project.

You've certainly encountered a version of the HelloWorld program—regardless of your programming background. The Be incarnation of the HelloWorld application performs as expected—the phrase "Hello, World!" is written to a window. Figure 2-2 shows what is displayed on your screen when the HelloWorld program is launched.

Figure 2-2. The window displayed by the HelloWorld program

You may encounter a number of versions of the HelloWorld project—there's one in the *sample-code* folder, and you may uncover other incarnations on Be CD-ROMs or on the Internet. So that you can follow along with me, you might want to use the version I selected—it's located in its own folder in the Chapter 2 folder of example projects. Figure 2-3 shows the contents of this book's version of the *HelloWorld* folder.

As shown in Figure 2-3, when developing a new application, the general practice is to keep all of the project's files in a single folder. To organize your own projects, you may want to create a new folder with a catchy name such as *myProjects* and store it in the */boot/home* folder—as I've done in Figure 2-3. To

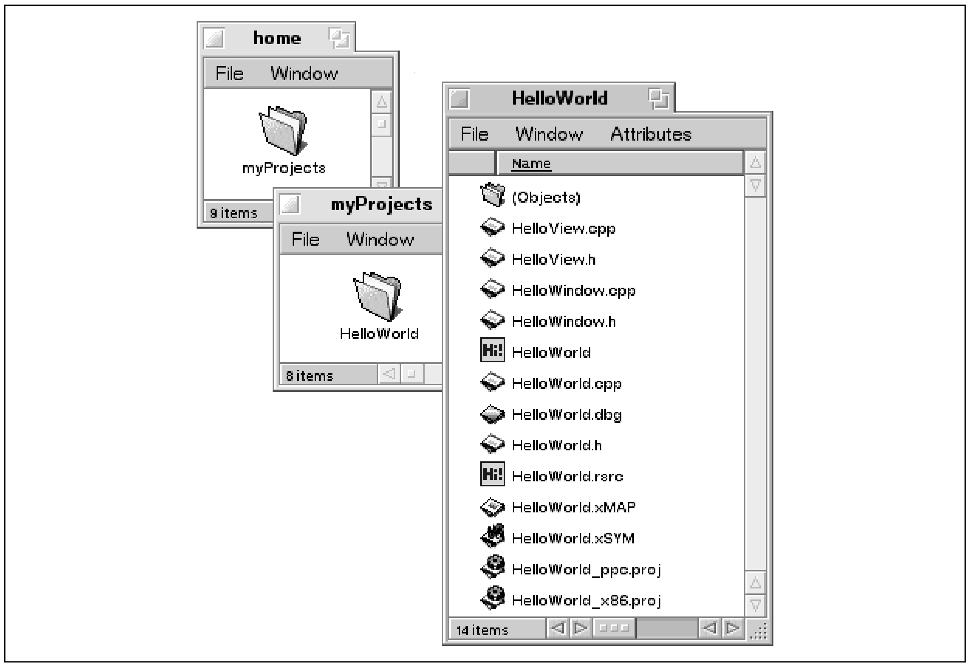


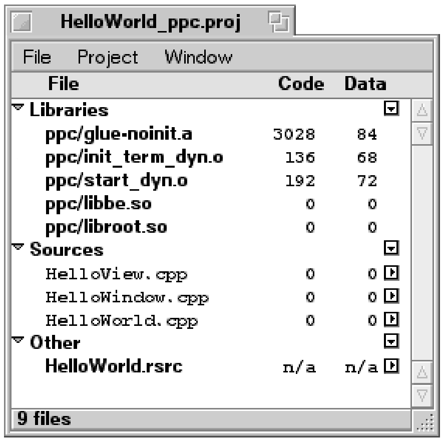
Figure 2-3. The files used in the development of a Be application

begin experimenting, you can copy this book's *HelloWorld* example folder to your own project folder. That way you're sure to preserve the original, working version of this example.

Project File

A Be application developed using the BeIDE starts out as a *project file*. A project file groups and organizes the files that hold the code used for one project. By convention, a project file's name has an extension of *.proj*. It's general practice to give the project file the same name the application will have, with the addition of an underscore and then *ppc* for a PowerPC-based project or an underscore and then *x86* for an Intel-based project. In Figure 2-3, you can see that for the HelloWorld project there are two versions of the project file: *HelloWorld_ppc.proj* and *HelloWorld_x86.proj*.

To open a project file, you can either double-click on its icon or start the BeIDE application and choose Open from the File menu. In either case, select the project that's appropriate for the platform you're working on. When a project file is opened, its contents are displayed in a *project window*. As shown in Figure 2-4, a project window's contents consist of a list of files.



File	Code	Data
Libraries		
ppc/glue-noinit.a	3028	84
ppc/init_term_dyn.o	136	68
ppc/start_dyn.o	192	72
ppc/libbe.so	0	0
ppc/libroot.so	0	0
Sources		
HelloView.cpp	0	0
HelloWindow.cpp	0	0
HelloWorld.cpp	0	0
Other		
HelloWorld.rsrc	n/a	n/a

9 files

Figure 2-4. The project window for the PowerPC version of the HelloWorld project

The files listed in a project window are the files to be compiled and linked together to form a single executable. This can be a combination of any number of source code, resource, and library files. The HelloWorld project window holds three source code files and one resource file, each of which is discussed in this chapter. The project window also lists one or more libraries. The number of libraries varies depending on whether you're working on a PowerPC version or an Intel version of a project. Figure 2-4 shows the PowerPC version of the HelloWorld project. In this project, the *glue-noinit.a*, *init_term_dyn.o*, and *start_dyn.o* libraries collectively make up the Be runtime support library that handles the dynamic linking code used by any Be application. An Intel project doesn't list these libraries—they're linked in automatically. The *libroot.so* library handles library management, all of the Kernel Kit, and the standard C library. The *libnet.so* library handles networking, while the *libbe.so* library is a shared library that contains the C++ classes and the global C functions that encompass many of the other kits. An Intel project lists only the *libbe.so* library—the other two libraries are always automatically linked in. The Be kits hold the software that make up much of the BeOS, so this library is a part of the Be operating system rather than a file included with the BeIDE environment.

Library filenames will be prefaced with an indicator as to the project's target platform (the platform on which the resulting application is to run). Figure 2-4 shows a project targeted for the PowerPC (Power Macintosh or BeBox) platform.

Project activity is controlled from the Project menu located in the project window menubar. In Figure 2-5, you see that this menu is used to add files to and remove files from a project. From this menu, you can compile a single file, build an application, and give a built application a test run. In the "Setting Up a New BeIDE Project" section, you'll make use of several of the menu items in the Project menu.

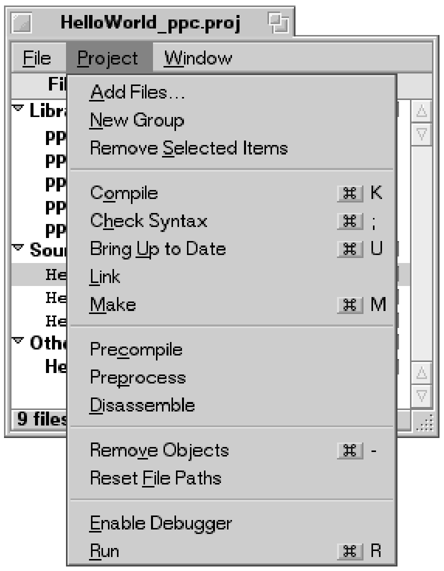


Figure 2-5. The Project menu in the menubar of a BeIDE project window

Of the many items in the Project menu, the Run/Debug item is the most important. Figure 2-5 shows that this bottom menu item is named Run—but this same item can instead take on the name Debug. When the menu item just above this one says Enable Debugger, then the Run/Debug item is in the Run mode. When the menu item just above instead says Disable Debugger, then the Run/Debug item is in the Debug mode. In either case, choosing Run or Debug causes all of the following to happen:

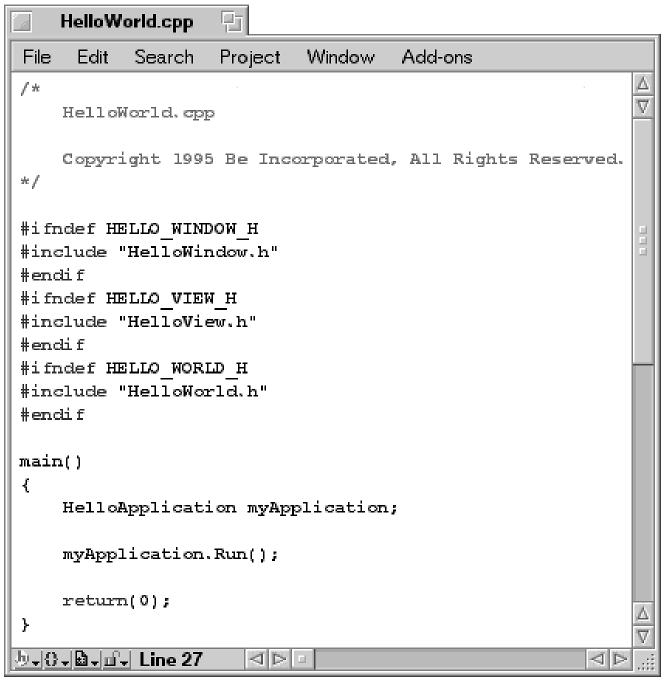
- Compile all project files that have been changed since the last compilation (which may be none, some, or all of the files in the project)
- Link together the resulting object code
- Merge the resource code from any resource files to the linked object code to make (build) an executable (an application)
- Launch the resulting application in order for you to test it (if no compile or link errors occurred)

If the Run/Debug menu is in Debug mode, then the last step in the above list takes place in the debugger. That is, the application is launched in the appropriate debugger (MWDebug-Be for PowerPC projects and bdb for Intel projects). Many of the other items in the Project menu carry out a subset of the operations that are collectively performed by the Run/Debug item.

If you haven't compiled any Be source code yet, go ahead and give it a try now. Open the HelloWorld project file. To avoid the debugger during this first test, make sure the Project menu item just above the Run/Debug item says Enable Debugger (select the item if it doesn't say that). Now choose Run from the Project menu to compile the project's code and run the resulting HelloWorld application.

Source Code and Header Files

The BeOS is a C++ application framework, so your source code will be written in C++ and saved in source code files that have an extension of *.cpp*. To open an existing source code file that is a part of a project, double-click on the file's name in the project window. That's what I did for the *HelloWorld.cpp* file that's part of the HelloWorld project—the result is shown in Figure 2-6.



```
/*
    HelloWorld.cpp

    Copyright 1995 Be Incorporated, All Rights Reserved.
*/

#ifndef HELLO_WINDOW_H
#include "HelloWindow.h"
#endif
#ifndef HELLO_VIEW_H
#include "HelloView.h"
#endif
#ifndef HELLO_WORLD_H
#include "HelloWorld.h"
#endif

main()
{
    HelloApplication myApplication;

    myApplication.Run();

    return(0);
}
```

Figure 2-6. The source code window for the HelloWorld.cpp source code file

Most of your code will be kept in source code files. Code that might be common to more than one file may be saved to a header file with an extension of *.h*. While you can keep a project's code in as few or as many source code and header files as desired, you'll want to follow the way Be does things in its examples.

Project file organization convention

Be example projects organize source code into files corresponding to a convention that's common in object-oriented programming. The declaration, or specifier, of an application-defined class exists in its own header file. The definitions, or implementations, of the member functions of this class are saved together in a single source code file. Both the header file and the source code file have the same name as the class, with respective extensions of *.h* and *.cpp*.

There's one notable exception to this naming convention. A project usually includes a header file and source code file with the same name as the project (and thus the same name as the application that will be built from the project). The header file holds the definition of the class derived from the `BApplication` class. The source code file holds the implementations of the member functions of this `BApplication`-derived class, as well as the `main()` function.

File organization and the HelloWorld project

Now let's take a look at the HelloWorld project to see if it follows the above convention. Because this example is based on a project from Be, Inc., you can guess that it does, but you'll want to bear with me just the same. The point here isn't to see if the HelloWorld project follows the described system of organizing files, it's to examine an existing project to clarify the class/file relationship.

Back in Figure 2-4 you saw that the HelloWorld project window displays the names of three source code files: *HelloView.cpp*, *HelloWindow.cpp*, and *HelloWorld.cpp*. While it's not obvious from the project window, there is also a header file that corresponds to each of these source code files (opening a source code file and looking at its `#include` directives reveals that information). According to the previous discussion, you'd expect that the *HelloView.h* file holds a listing for a class named `HelloView`. Here's the code from that file:

```
// -----------------------------------------------------------
// HelloView.h

class HelloView: public BView {

public:
                HelloView(BRect frame, char *name);
virtual  void   AttachedToWindow();
virtual  void   Draw(BRect updateRect);
};
```

Looking at the code in the *HelloView.cpp* file, we'd expect to see the implementations of the three member functions declared in the `HelloView` class definition. And we do:

```
// -----------------------------------------------------------
// HelloView.cpp

#include "HelloView.h"

HelloView::HelloView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
    ...
}


void HelloView::AttachedToWindow()
{
    ...
}


void HelloView::Draw(BRect updateRect)
{
    ...
}
```

As you can see from the *HelloView.cpp* listing, I'm saving a little ink by not showing all of the code in the project's files. Nor do I describe the code I do show. Here I'm only interested in demonstrating the relationship between a project's files and the classes defined by that project. I do, however, take care of both of those omissions at the end of this chapter in the "HelloWorld Source Code" section.

I said I wouldn't discuss the HelloWorld source code here. Out of decency to the very curious, though, I will make a few quick comments. You're familiar with the SimpleApp example that was introduced in Chapter 1, *BeOS Programming Overview*. That example defined two classes. One was named `SimpleWindow` and was derived from the `BWindow` class. It was used to display a window. The second class was named `SimpleApplication` and was derived from the `BApplication` class. Every Be program needs to define such a class. The HelloWorld example discussed here defines similar classes named `HelloWindow` and `HelloApplication`. It also defines a third class named `HelloView`, which is derived from the `BView` class. Before writing or drawing to a window, a program must define a view—an area in the window to which drawing should be directed. The SimpleApp program didn't draw to its window, so it didn't need a class derived from the `BView` class.

The second source code file shown in the project window in Figure 2-4 is *HelloWindow.cpp*. This file has a corresponding header file named *HelloWindow.h*. In this file we expect to find the declaration of a class named `HelloWindow`—and we do:

```
// ----------------------------------------------------------
// HelloWindow.h

class HelloWindow : public BWindow {

public:
                HelloWindow(BRect frame);
virtual  bool  QuitRequested();
};
```

The *HelloWindow.cpp* file contains the source code for the two `HelloWindow` member functions, `HelloWindow()` and `QuitRequested()`:

```
// ----------------------------------------------------------
// HelloWindow.cpp

#include "HelloWindow.h"

HelloWindow::HelloWindow(BRect frame)
    : BWindow(frame, "Hello", B_TITLED_WINDOW,
              B_NOT_RESIZABLE | B_NOT_ZOOMABLE)
{
    ...
}

bool HelloWindow::QuitRequested()
{
    ...
}
```

Earlier I stated that the header file that bears the name of the project should hold the declaration of the project's application class—the class derived from the `BApplication` class. Here you see that the *HelloWorld.h* header file does indeed hold this declaration:

```
// ----------------------------------------------------------
// HelloWorld.h

class HelloApplication : public BApplication {

public:
          HelloApplication();
};
```

The source code file with the name of the project should hold the code for the implementation of the member functions of the application class as well as the `main()` function. *HelloWorld.cpp* does hold the following code.

```
// -----------------------------------------------------------
// HelloWorld.cpp

#include "HelloWindow.h"
#include "HelloView.h"
#include "HelloWorld.h"

int  main()
{
    ...
    ...
}

HelloApplication::HelloApplication()
    : BApplication("application/x-vnd.Be-HLWD")
{
    ...
    ...
}
```

While I omitted the code that makes up the body of each member function of each class in the HelloWorld project, you may still have picked up on similarities between the HelloWorld source code and the source code of the Chapter 1 example, SimpleApp. In the section "HelloWorld Source Code" I point out all of the similarities and discuss the differences.

In looking at existing source code, you may encounter a `BApplication` constructor argument that's four characters between single quotes rather than a long, double-quoted MIME string. The four-character method is the old way of supplying a signature to an application, and is dated. The newer MIME string format is discussed in more detail later in this chapter.

Resources and the Resource File

It's nice to sum up a programming concept in a single sentence, as in "a pointer is a reference to a specific area in memory." Unfortunately, such conciseness isn't always possible. Such is the case with the subject of resources. I'll begin with a short summation—"a resource is code that represents one element of a program"—but adding clarity to that vague explanation necessitates a couple of paragraphs.

The "element of a program" I speak of is usually thought of as one part, or entity, of a program's graphical user interface. For instance, some operating systems make it easy to represent a window or menu as a resource. But a resource doesn't have to represent something graphical. For instance, an application's signature—a short, unique string that helps the operating system differentiate the application from all other applications—is anything but graphical. Yet it can be a resource. While an application's signature isn't graphical in nature, the way in which it can be created and edited can be thought of as graphical. For instance, one could imagine a simple editor that had a Create Application Signature menu item which, when selected, displayed a text box in which a short string was typed. The editor would then be responsible for saving these typed characters as a resource. So it turns out that rather than *representing* something that is itself graphical, a resource is usually something that can be *created and edited* graphically.

Being graphically editable is typically one trait that makes an element a candidate to be represented by a resource. Since some programmer will have to design a special editor that is capable of graphically editing a resource, another requirement is that the element be something common to most or all programs.

You've just read that different program elements exist as resources for a variety of reasons. An application's icon is a good example. First, an icon is a small picture, so it of course is an entity that lends itself to being easily edited graphically. Second, all applications have an icon that is used to represent the application on the desktop, so it made sense for someone to expend the effort to create an editor capable of editing icons. Finally, the BeOS needs the data that defines an application's icon even when the application isn't running, so that it can display the icon on the desktop at all times.

There are different types of resources, and the BeOS keeps track of these different types by using a different 32-bit integer for each resource type. As a convenience to programmers, a four-character constant is often used to define this integer. Consider the icon that represents an application on the desktop. The data that defines this icon exists as a resource, and its type is 'ICON.' Most programmers find it easier to remember the four-character constant 'ICON' than the numerical value this constant represents.

While a resource type is surrounded in single quotes in this book and in Be documentation as well, the quotes aren't a part of the resource type—a resource type is simply the four characters (or an actual 32-bit numerical value). The quotes are used only to make it obvious that a resource type is being discussed. This is important because a resource type can be in lowercase, and it can include a space or spaces. Placing an icon type in quotes sets it apart from the rest of the text that appears with it.

Application-information resource

There's one other reason that a certain part of a program will exist as a resource—a reason unrelated to the graphical nature of the element or its ability to be edited graphically. Because of the way in which resources are stored in an executable, resource information is available to the BeOS even when the application isn't running. The BeOS needs some information about an application in order to be able to effectively communicate with it. This information can be kept together in a single resource of type 'APPI' (for "application information") in the application. An 'APPI' resource consists of the following pieces of information about an application:

Launch Behavior
: The launch characteristics of a Be application can fall into one of three categories. *Single launch* is the typical behavior—no matter how many times a user double-clicks on the application's icon, only one instance of the executable is loaded into memory and executed (that is, double-clicking on an application's icon a second time has no effect). It's possible for two versions of a single launch application to end up in memory if the user makes a duplicate of the original executable and then double-clicks on each. *Exclusive launch* is a behavior that restricts this from occurring. Under no circumstance can two versions of a program execute at the same time. *Multiple launch* is a behavior that allows any number of instances of a single copy of a program to execute simultaneously.

Background App
: An application can forego a user interface and run in the background only. If an application is marked as a background app, it behaves in this way and won't be named in the Deskbar.

Argv Only
: An application can be excluded from receiving messages from the BeOS (refer to Chapter 1 for an introduction to messages). Marking an application as *argv only* means that the only information the application receives comes from the `argc` and `argv` command-line arguments that can be optionally passed to the program's `main()` routine.

Signature
: Each application has a string that lets the BeOS view the application as unique from all others. Obviously, no two applications should share the same signature. For your own test programs, the signature you choose isn't too important. Should you decide to distribute one of your applications to the Be community, though, you'll want to put a little effort into selecting a signature. Be's recommended format is "application/x-vnd.VendorName-ApplicationName".

Replacing VendorName with your company's name should provide a unique signature for your application.

Here I'll look at 'APPI' information for an existing project that already includes a resource file. In this chapter's "Setting Up a New BeIDE Project" section you'll find information on creating a resource file and editing its 'APPI' information. To view the 'APPI' information in a project's resource file, double-click on its name in the project window. That launches the FileTypes application (which can also be launched by choosing it from the preferences folder in the Be menu) and opens two windows. Figure 2-7 shows the main window of FileTypes.

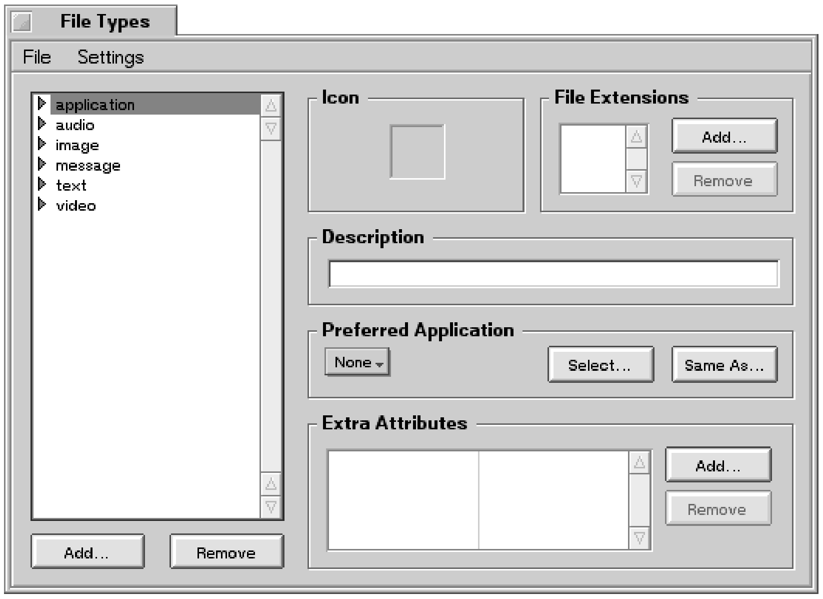
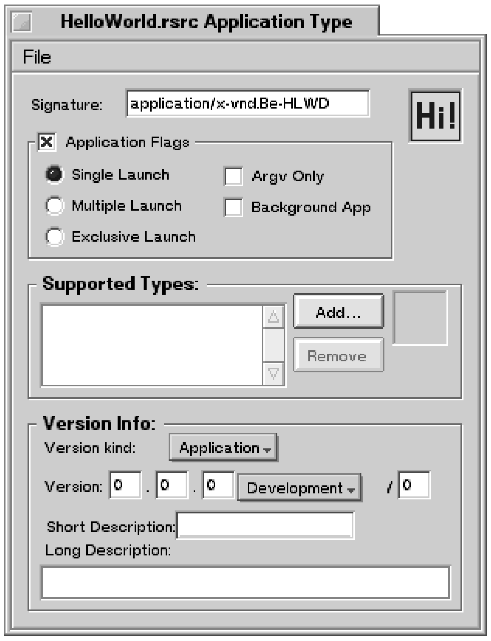


Figure 2-7. The main FileTypes window

To view or edit an application's 'APPI' resource information, work in the second of FileTypes' two windows. Figure 2-8 shows this window for the HelloWorld application.

The application's launch behavior is determined based on which of the three Launch radio buttons is on—Single Launch, Multiple Launch, or Exclusive Launch (only one can be on at any given time).

Whether or not the application is a background app is determined by the status of the Background App checkbox. Whether or not the application is capable of receiving messages is determined by the status of the Argv Only checkbox. While these two items appear grouped together in the Application Flags area, they aren't related. Neither, either, or both can be checked at the same time.



HelloWorld.rsrc Application Type

File

Signature: application/x-vnd.Be-HLWD

Hi!

[✓ Application Flags]

[● Single Launch] ◯ Multiple Launch ◯ Exclusive Launch

☐ Argv Only ☐ Background App

Supported Types:

Add... Remove

Version Info:

Version kind: Application

Version: 0 . 0 . 0 Development / 0

Short Description:

Long Description:

Figure 2-8. Viewing the 'APPI' information for the HelloWorld application

An application's signature is based on the MIME string you enter in the signature edit box of the FileTypes window. If a signature appears here, the string passed to the `BApplication` constructor will be ignored (refer to Chapter 1). If no signature appears here, the string passed to the `BApplication` constructor will be used. Thus, by entering a string in the FileTypes window, you're making the `BApplication` constructor argument immaterial. Figure 2-8 shows the signature for the HelloWorld application used throughout this chapter.

If you make any changes to a project's resource file, save them by choosing Save from the File menu of FileTypes (the File menu's other item, Save into Resource File, is discussed in the "Setting Up a New BeIDE Project" section of this chapter).

Icon resource

An icon could be described within source code (and, in the "old days," that was in fact how icons were described), but the specification of individual pixel colors in source code is difficult and tedious work. Rather than attempting to specify the colors of each pixel of an icon from within source code, a BeOS application's icon can be created using a special graphical editor built into the FileTypes application.

The graphical editor in FileTypes is used in a manner similar to the way you use a graphics paint program—you select a color from a palette of colors and then use a pencil tool to click on individual pixels to designate that they take on that color. See Figure 2-9 to get an idea of what the graphical editor in FileTypes looks like.

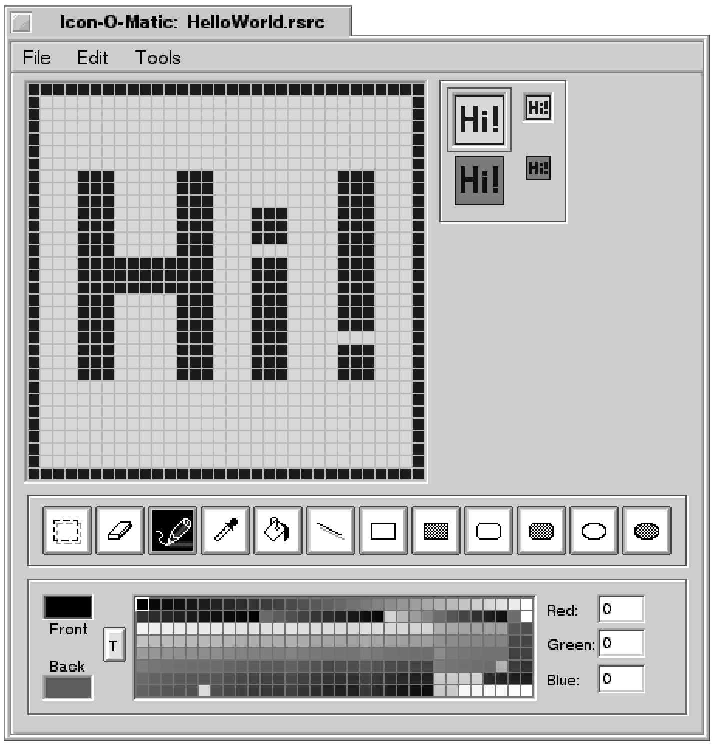


Figure 2-9. The icon editing window and palettes displayed by FileTypes

In FileTypes you simply draw the icon. You can then save the icon to a project's resource file so that each time an application is built from the project, your icon is merged with the application (and becomes the icon viewed on the desktop by the user). To view or edit the icon stored in the resource file of an existing project, you first double-click on the resource filename in the project window to open the resource file. After FileTypes opens the resource file, double-click on the small icon box located at the upper right of the FileTypes window; you'll see the window shown in Figure 2-9.

Setting Up a New BeIDE Project

In the previous section, you read that an application starts as a Be project. The Be project consists of a project file, source code files, header files, libraries, and a resource file. The project file itself doesn't hold any code; it serves as a means to organize all the other files in the project. The project file also serves as the project "command center" from which you compile code and build and test the executable. A close look at the HelloWorld project clarified many of these concepts.

When you set out to develop your own application, you'll find that you may be able to save some effort if you don't start from scratch, but instead duplicate a folder that holds the files of an existing project. Consider this scenario: I want to create a very simple children's game—perhaps a tic-tac-toe game. I know that the HelloWorld project results in a program that displays a single window and draws to it. That represents a good part of what my game will need to do, so it makes sense for me to base my game on the HelloWorld project, and then modify and add to the HelloWorld source code as needed. If your program will be a complex one, or one for which you can't find a similar "base" program to start with, this approach won't be as fruitful. In such cases you'll want to start with a new project.

In this section, I'll discuss each step of the process of setting up a new project first in general terms. I'll also carry out each step using the HelloWorld project to provide a specific example. So you see, I had good reason for devoting the previous several pages to a look at the HelloWorld project. While I use the small HelloWorld project for simplicity, the steps discussed on the following pages apply just as well to projects of a much larger scale.

In the above paragraphs, I refer to using code written by others. Before doing that you'll of course want to make sure that you're allowed to do so! The BeOS CD-ROM comes with a number of example projects that fall into the category of projects that are available for your own personal use. The source code that makes up the example projects is copyright Be, Inc., but Be, Inc. has granted unrestricted permission for anyone to use and alter any of this source code. I've taken advantage of this fact and used these projects as the basis for the numerous examples that appear in this book. In turn, you're free to use without restrictions the example code in this book for your own projects.

The following is an overview of the major steps you'll carry out each time you create a new project. While on the surface it may appear that performing these steps involves a lot of work, you'll find that after you've set up a few new projects the process becomes quite routine, and doesn't take much time at all. All of the steps are discussed in the sections that follow this list.

1. Find an existing project that is used to build a program that has similarities to the program you're to develop.
2. Duplicate the existing project folder and its contents.

3. Open the new project folder and change the names of the project, source code, header, and resource files to names that reflect the nature of the project you're working on.
4. Open the newly renamed project and drag the renamed source code files and resource file from the folder and drop them into the project window.
5. Remove the now obsolete source code and resource filenames from the project window.
6. Edit the name of the constants in the `#ifndef` directive in the header files and the `#includes` in the source files.
7. Test the project's code by building an application (here you're verifying that the original source code is error-free before you start modifying it)
8. If there are library-related errors, create a new project (which will automatically include the most recent versions of each library) and add the source code files and resource file to the new project.
9. If there are compilation errors, correct the source code that caused the errors.
10. Open the header files and change application-defined class names in the header files to names that make sense for the project you're working on.
11. Change all usage of application-defined class names in the source files to match the changes you made in the header files.
12. Open the resource file using FileTypes and modify any of the 'APPI' resource information and the icon.
13. Set the name for the executable to be built.
14. Build a new application from the modified BeIDE project.

No new functionality will have been added to the program that gets built from the new project—it will behave identically to the program that results from the original project. So why go through the above busy-work? Executing the above steps results in a new project that includes source code files that define and use classes with new names—names that make sense to you. This will be beneficial when you start the real work—implementing the functionality your new program requires.

The above list isn't an iron-clad set of steps you must follow. Other programmers have their own slightly (or, perhaps, very) different guidelines they follow when starting a new project. If you're new to the BeIDE, the BeOS, or both, though, you might want to follow my steps now. As you get comfortable with working in a project-based programming environment, you can vary the steps to match your preferred way of doing things.

Selecting and Setting Up a Base Project

As mentioned, you'll get off to the best start in your programming endeavor by finding a project that matches the following criteria:

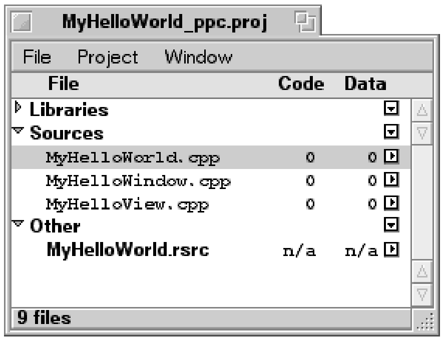
- The application that is built from the original project has several features common to the program you're to develop.
- You have access to the project and all its source code and resource files.
- It's made clear that the project's source code can be modified and redistributed, or you have the developer's permission to do so.

Once you've found a project that meets the above conditions, you've performed Step 1 from the previous section's numbered list.

Step 2 involves creating a copy of the project folder and its contents. After doing that, rename the new folder to something appropriate for the project you're embarking upon. Usually a project folder has the same name that the final application that gets built from the project will have. Here I'm making a new project based on the HelloWorld project only for the sake of providing a specific example, so after duplicating the HelloWorld folder, I'll simply change the name of the folder from *HelloWorld copy* to *MyHelloWorld* (each of these folders can be found in the Chapter 2 examples folder available on the O'Reilly web site).

Step 3 is the renaming of the project-related files. Double-click on the new folder to reveal its contents. Click on the name of any one of the header files and type a new name for the file. For my new MyHelloWorld project I'll rename the *HelloView.h*, *HelloWindow.h*, and *HelloWorld.h* header files to *MyHelloView.h*, *MyHelloWindow.h*, and *MyHelloWorld.h*, respectively. Next, rename the source code files (so, for example, *HelloWorld.cpp* becomes *MyHelloWorld.cpp*) and the resource file (here, from *HelloWorld.rsrc* to *MyHelloWorld.rsrc*). Now rename the project file. Again, choose a name appropriate to the project. Typically, the project file has the same name the application will have, with an extension of *x86.proj* or *ppc.proj* added. I'll change the PowerPC version of the HelloWorld project by updating the project filename from *HelloWorld_ppc.proj* to *MyHelloWorld_ppc.proj*.

Steps 4 and 5 are performed to get the project to recognize the newly named files. After the name changes are made, double-click on the project file to open it. The project window will appear on your screen. If you renamed a file from the desktop, the project file that includes that file will list it by its original, now invalid, name. That necessitates adding the file by its new name and removing the original file from the project. To add the newly named files, select them from the desktop (click on each) and drag and drop them into the project window. In the project window, drag each to its appropriate group (for instance, place *MyHelloWorld.cpp* in the Sources group). To remove the original files from the project, select each and choose Remove Selected Items from the Project menu. For the MyHelloWorld project, the resulting project window looks like the one shown in Figure 2-10.



File	Code	Data
Libraries		
Sources		
MyHelloWorld.cpp	0	0
MyHelloWindow.cpp	0	0
MyHelloView.cpp	0	0
Other		
MyHelloWorld.rsrc	n/a	n/a

Figure 2-10. The MyHelloWorld project window with renamed files in it

Testing the Base Project

The new project now has newly named files in it, but these files hold the code from the original project. Before adding new functionality to the code, verify that it compiles without error (this is Step 7 from the previous list). Before compiling the code, though, perform Step 6—update the `#includes` at the top of each source code file so that they match the new names of the header files. For example, near the top of *MyHelloView.cpp* is this `#include`:

```
#ifndef HELLO_VIEW_H
#include "HelloView.h"
#endif
```

I've changed the *HelloView.h* header file to *MyHelloView.h*, so this `#include` needs to be edited. While I'm doing that, I'll change `HELLO_VIEW_H` to the more appropriate `MY_HELLO_VIEW_H` (though I could leave this as is—it's simply a constant I define in the *MyHelloView.h* header file).

```
#ifndef MY_HELLO_VIEW_H
#include "MyHelloView.h"
#endif
```

Because I changed the name of the constant `HELLO_VIEW_H` in the *MyHelloView.cpp* source file, I need to change this same constant in the *MyHelloView.h* header file. Originally the header file contained this code:

```
#ifndef HELLO_VIEW_H
#define HELLO_VIEW_H
```

Now that code should look like this:

```
#ifndef MY_HELLO_VIEW_H
#define MY_HELLO_VIEW_H
```

Finally, test the code by choosing Run from the Project menu. Later, if you experience compilation errors after you've introduced changes to the original code, you'll know that the errors are a direct result of your changes and not related to problems with the original code.

If the building of an application is successful, Steps 8 and 9 are skipped. If the attempt to build an application results in library-related errors (such as a library "file not found" type of error), you're probably working with a project created under a previous version of the BeIDE. The easiest way to get the proper libraries into a project is to follow Step 8—create a new project based on one of the Be-supplied project stationeries. A new BeIDE project file can be created by choosing New Project from the File menu of an existing project. When you do that, the New Project dialog box appears to let you choose a project stationery from which the new project is to be based. The project stationery is nothing more than a template that specifies which libraries and project settings are best suited for the type of project you're creating. Here are definitions of the more important stationeries:

BeApp
Stationery that links against the standard libraries applications need.

EverythingApp
Stationery that links against all of the Be libraries.

KernelDriver
A basic template for writing a kernel driver.

SharedLib
Stationery used to create a basic library or add-on, this links against the basic Be libraries.

BeSTL
Stationery used to create an application that includes the C++ standard libraries (including STL and the basic iostream functions).

In Figure 2-11, I'm choosing the *BeApp* project stationery under the *ppc* heading in order to create a project that's set up to generate a Be application targeted for the PowerPC platform (the list of project stationeries you see may differ depending on the version of the BeOS you're using and the processor (PowerPC or Intel) in your machine). Note that when creating a new project you'll want to uncheck the Create Folder checkbox in the New Project dialog box and specify that the new project end up in the folder that holds all the renamed files.

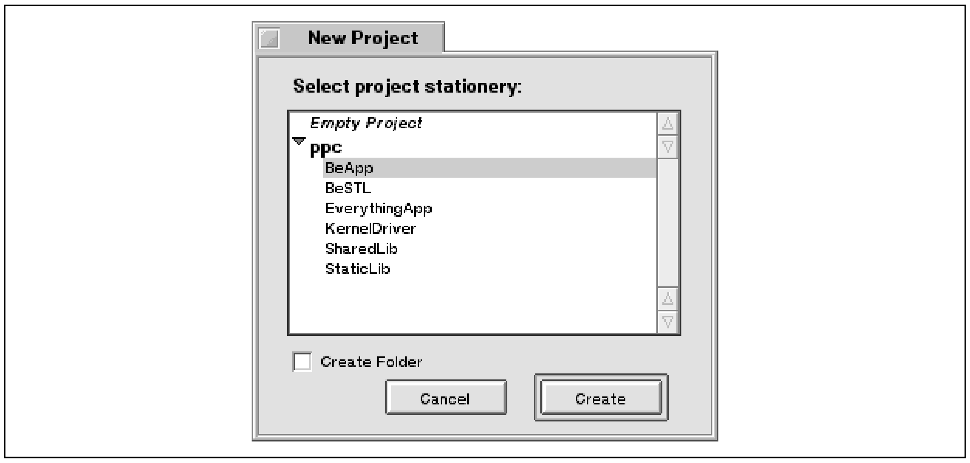


Figure 2-11. Choosing a stationery on which to base a new project

The act of creating a new project doesn't provide you with any source code files or a resource file—you'll need to repeat Step 4. That is, drag and drop the necessary files from the desktop folder to the new project window.

If the building of an application results in compilation errors, now's the time to correct them. This is Step 9. Only after you successfully build an application does it make sense to start making changes to the project's source code.

Preliminary Code Changes

You'll of course be making changes to the source code in the source code files. The most interesting of these changes will be the ones that turn the original code into code that results in an application that is distinctly your own. First, however, you need to make some preliminary source code changes. You've changed the names of the files, including the header files, so you'll need to search for and change header file references in the source code files. You'll also want to change the names of the classes already defined to match the names you've given to the header files.

Header file code changes

Step 10 in the list of new project setup tasks is the changing of application-defined class names in the header files. Begin the changes by opening any one of the header files. The quickest way to do that is to click on the small arrow icon to the right of one of the source code filenames in the project window. Doing that displays a menu that lists the header files included in the selected source code file. To open a header file, simply select it from this popup menu. With a header file open, make the following changes to the code.

- Add the new name of the file to the file's description comment section.
- If you haven't already done so, rename the application-defined constant that is used in the `#ifndef` and `#define` preprocessor directives.
- Rename the file's application-defined class.
- Rename the class constructor to match the new name you've give to the application-defined class.

The above steps are carried out in the same way regardless of the project you start with. To provide a specific example of how these changes are implemented, I'll change the *HelloView.h* header file from the HelloWorld project. The following listing shows the original version of the *HelloView.h* file. Refer to it in the discussion that follows the listing.

```
// ----------------------------------------------------------
    HelloView.h
    Copyright 1995 Be Incorporated, All Rights Reserved.

#ifndef HELLO_VIEW_H
#define HELLO_VIEW_H

#ifndef _VIEW_H
#include <View.h>
#endif

class HelloView : public BView {

public:
               HelloView(BRect frame, char *name);
virtual  void  AttachedToWindow();
virtual  void  Draw(BRect updateRect);
};

#endif

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

The first change to the header file, the altering of the file's descriptive comment, needs little discussion. You may want to leave the original name intact as a courtesy to the original author. I've done that in the *MyHelloView.h* file (the listing of which follows this discussion).

The `HELLO_VIEW_H` constant is defined to eliminate the possibility of the code in this header file being included more than once in the same source code file. Earlier I changed its name to `MY_HELLO_VIEW_H` to reflect the new name I've given to this header file (*MyHelloView.h*):

```
#ifndef MY_HELLO_VIEW_H
#define MY_HELLO_VIEW_H
```

The `_VIEW_H` constant is a BeOS-defined constant (it's used to ensure that the BeOS header file *View.h* doesn't get included multiple times) so it can be left as is. If you aren't clear on the usage of the `#ifndef` preprocessor directive in Be files, refer to the "Header Files and Preprocessor Directives" sidebar.

The original class defined in the *HelloView.h* file was named `HelloView`. I've renamed that class to `MyHelloView` to match the name I've given the header file. I also changed the name of the class constructor from `HelloView()` to `MyHelloView()`. All of lines of code that include changes are shown in bold in the following listing of *MyHelloView.h*:

```
// -----------------------------------------------------------
    MyHelloView.h
    Dan Parks Sydow
    1999

    Based on source code from:
    HelloView.h
    Copyright 1995 Be Incorporated, All Rights Reserved.
// -----------------------------------------------------------

#ifndef MY_HELLO_VIEW_H
#define MY_HELLO_VIEW_H

#ifndef _VIEW_H
#include <View.h>
#endif

class MyHelloView : public BView {

public:
                MyHelloView(BRect frame, char *name);
virtual  void   AttachedToWindow();
virtual  void   Draw(BRect updateRect);
};

#endif

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

Notice that I've stopped short of making any substantial changes to the application-defined class. While you may be tempted to "get going" and start adding new member functions to a class, it's best to wait. First, make all the name changes in the header files. Then update the source code files so that any usage of application-defined classes reflects the new names (that's discussed next). Finally, verify that everything works by compiling the code. Only after you're satisfied that all the preliminaries are taken care of will you want to start making significant changes to the code.

For the MyHelloWorld project, I'd repeat the above process on the *MyHelloWindow.h* and *MyHelloWorld.h* header files.

Header Files and Preprocessor Directives

In a large project that consists of numerous source code files and header files, the potential for a source code file to include the same header file more than once exists. Consider three files from a large project: source file *S1.cpp* and header files *H1.h* and *H2.h*. If *S1.cpp* includes both *H1.h* and *H2.h*, and *H1.h* includes *H2.h*, then *S1.cpp* will include the code from *H2.h* twice (once directly and once indirectly via *H1.h*). Such an event results in a compilation error that, given the dependency of the files, can be difficult to remedy.

To coordinate the inclusion of header file code in source code files, Be projects typically use the "if not defined" (`#ifndef`) preprocessor conditional directive to define a constant in a header file and to check for the definition of that constant in a source code file. This is a standard technique used by many C and C++ programmers—if you're familiar with it, you can feel free to skip the remainder of this sidebar.

A header file in a Be project begins by checking to see if a particular constant is defined. If it isn't defined, the header file defines it. Because all the remaining code in the header file lies between the `#ifndef` and the `#endif` directives, if the constant is already defined, the entire contents of the header file are skipped. Here's that code from the HelloWorld project's *HelloView.h* header file:

```
#ifndef HELLO_VIEW_H
#define HELLO_VIEW_H
// class declaration
#endif
```

Before including a header file, a source code file checks to see if the constant defined in the header file is in fact defined. If it isn't defined, the source code file does include the header file. Here's that code from the *HelloView.cpp* source code file:

```
#ifndef HELLO_VIEW_H
#include "HelloView.h"
```

#endif source code file code changes

After you make the changes to the header files, it's time to update the source code files—this is Step 10 in the list of new project setup steps. Begin by double-clicking on one of the source code filenames in the project window. Then make the following changes to the code:

- Add the new name of the file to the file's description comment section.
- Rename the application-defined constant that is used in the `#ifndef` and `#define` preprocessor directives to match the renamed constant in the corresponding header file.
- Rename any header filenames that follow `#include` directives to match the renamed header files.
- Rename all occurrences of any application-defined classes to match the renamed classes in the other project header files.

As with header files changes, the source code file changes listed above apply to any project. Again, I'll use a file from the HelloWorld project to provide a specific example. The following listing shows the original version of the *HelloView.cpp* file:

```
// ------------------------------------------------------------
    HelloView.cpp
    Copyright 1995 Be Incorporated, All Rights Reserved.
// ------------------------------------------------------------

#ifndef HELLO_VIEW_H
#include "HelloView.h"
#endif

HelloView::HelloView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
    ...
}


void HelloView::AttachedToWindow()
{
    ...
}


void HelloView::Draw(BRect updateRect)
{
    ...
}

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

Next you'll see the edited version of *HelloView.cpp*—it's now the code for a file named *MyHelloView.cpp*. Because there are no occurrences of any application-defined class names in the code in the bodies of any of the member functions, I've omitted that code for brevity.

```
// ------------------------------------------------------------
    MyHelloView.cpp
    Dan Parks Sydow
    1999

    Based on source code from:
    HelloView.cpp
    Copyright 1995 Be Incorporated, All Rights Reserved.
// -----------------------------------------------------------

#ifndef MY_HELLO_VIEW_H
#include "MyHelloView.h"
#endif

MyHelloView::MyHelloView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
    ...
}


void MyHelloView::AttachedToWindow()
{
    ...
}


void MyHelloView::Draw(BRect updateRect)
{
    ...
}

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

These steps need to be repeated for each source code file in the project. As you do that, note that a source code file may include occurrences of application-defined classes other than the class defined in the corresponding header file. Consider this snippet from the original *HelloWorld.cpp* file:

```
HelloApplication::HelloApplication()
    : BApplication("application/x-vnd.Be-HLWD")
{
    HelloWindow  *aWindow;
    HelloView    *aView;
    BRect         aRect;
```

Here you see that the constructor for the application-defined `HelloApplication` class declares variables of two different application-defined classes: `HelloWindow` and `HelloView`. After renaming the *HelloWorld.cpp* file to *MyHelloWorld.cpp*, the changes to the above snippet would turn it into code that looks like this:

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

The astute reader will have noticed that I slipped an extra change in the above snippet. If you didn't spot it, look at the parameter to the `BApplication()` constructor. Later in this chapter you'll see that I also use the FileTypes application to place this same signature in the project's resource file.

A quicker way to make the changes

You may have found all this talk about manually editing header files and source code files a little disconcerting. Before your level of frustration rises too high, it's time for me to make a confession. There is a shortcut to poring over page after page of code to track down occurrences of class names that need to be changed—a shortcut I didn't mention at the onset of the "Preliminary Code Changes" section. The BeIDE has a search and replace utility that makes finding and replacing names (such as constants and class names) a snap.

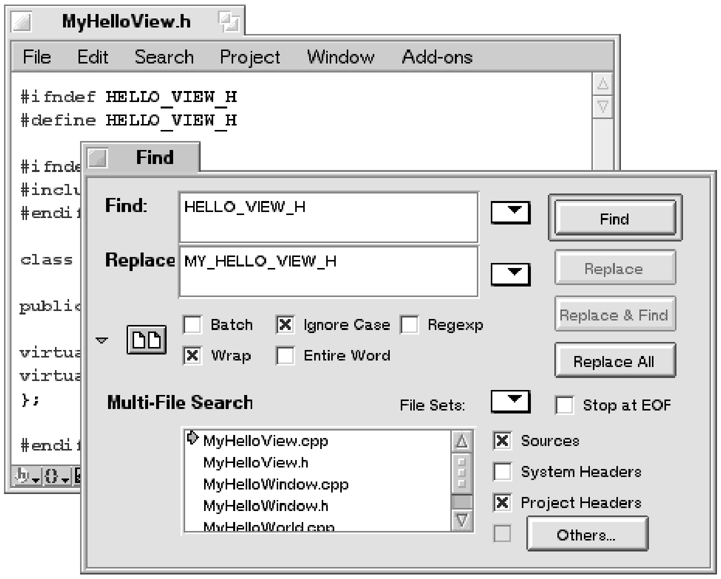
Before discussing how to quickly make global changes to all of the files in a project, I'll answer the one question that's on your mind: Why did I bother with the drawn-out explanation of exactly what changes to make if everything could be done with a few mouse clicks? I went laboriously through all the changes so that you'd know what you're changing and why.

Now, here's a revised method for changing the names of constants, files, and classes that appear in the header files and source code files of a project:

1. Choose Open from the BeIDE main menu in the dock and open a header file.
2. Choose Find from the Search menu of the opened header file.
3. Set up the Find window to search the project's source code files and header files.
4. Enter a name to change in the Find box and the name to replace it with in the Replace box of the Find window.
5. Click on the Find button in the Find window.
6. After verifying that the found text should indeed be replaced, click on the Replace & Find button in the Find window.
7. Repeat Step 6 until all occurrences of the name have been found and changed.
8. Repeat Steps 4 though 8 for each name to be changed.

You can speed things up by using the Replace All button once in place of the Replace & Find button several times. However, it's safer to use the Replace & Find button so that you can take a quick look at each name the BeIDE is changing.

If you want to perform a multiple-file search, the file icon located on the left side of the Find window must be displaying two files and the Find window must be expanded as shown in Figure 2-12. If the file icon is displaying a single file, click on the icon (it toggles between one and two files). If the Find window isn't expanded (the Multi-File Search section won't be visible), click on the small blue arrow located to the left of the file icon (the arrow expands and collapses the Find window).



```
#ifndef HELLO_VIEW_H
#define HELLO_VIEW_H

#ifnde
#inclu
#endi

class

publi

virtua
virtua
};

#endi
```

Find: HELLO_VIEW_H
Replace: MY_HELLO_VIEW_H

- [ ] Batch
- [x] Ignore Case
- [ ] Regexp
- [x] Wrap
- [ ] Entire Word
- [ ] Stop at EOF

MyHelloView.cpp
MyHelloView.h
MyHelloWindow.cpp
MyHelloWindow.h
MyHelloWorld.cpp

- [x] Sources
- [ ] System Headers
- [x] Project Headers

Figure 2-12. The BeIDE Find window set up to search all of a project's files

With the Find window set for multiple-file searching, specify which files are to be included in a search. Check the Sources checkbox to designate that all of the source code files listed in the project window are to be searched. To specify that the header files also be included in the search, you'll be tempted to check the Project Headers checkbox. At this point, however, the BeIDE doesn't know that the project source code files will be including the renamed header files—so it won't know to add the desired files to the Multi-File Search list. Instead, click on the Others button. In the Select a File to Search window that appears, Shift-click on the names of each of the renamed header files. That is, hold down the Shift key and successively click the mouse button while the cursor is over each header filename in the window's list of files. Click the Add button, then the Done button.

To quickly change a name, enter the name to change in the Find box, enter the new name in the Replace box, and click the Replace All button. The BeIDE will search all the project files for the name to find and replace each occurrence with the new name. For my MyHelloWorld project I began by searching for the constant `HELLO_VIEW_H` and replacing it with the new constant `MY_HELLO_VIEW_H`.

You can see in Figure 2-12 that the Find window is set up to perform that act. When done (it will only take a moment), enter another name and another new name in the Find and Replace boxes and again click on the Replace All button. Repeat these steps until all of the necessary changes have been made.

A click on the Replace All button quickly searches all of the files in a project and replaces all hits with new text. That makes the BeIDE Find facility a very powerful utility—*too* powerful if you aren't sure what you're changing. Hence my reasoning for describing this technique after the discussion on just what needs to be changed in the files of a project. Now, if you make a mistake that results in an error when you compile the project's code, you'll know where to look and what to look for to remedy the errors.

Testing the changes

After making all the changes to a project, compile the code by choosing Make from the Project menu in the project window's menubar. This menu item compiles all *touched* files and then builds a new version of the application. A touched file is one that has been edited since the last time the file was compiled. Before moving on to the "real" code changes—the possibly extensive changes and additions you'll be making to implement your program's functionality—you'll want to verify that the "cosmetic" changes you've just made didn't introduce any errors.

Editing the Resource File

You'll want to give your application a unique signature and its own icon. The 'APPI' and 'ICON' resources are both located in the project's resource file, so the FileTypes application is involved in both these acts.

Changing the signature

To edit the signature to be merged with the application built from the new project, double-click on project's resource file. This launches the FileTypes application and opens a window that holds 'APPI' information (refer back to Figure 2-8 for an example). To complete Step 12 of the new project setup process, modify the 'APPI' information as needed. Most likely this will involve nothing more than entering a new, unique signature to be assigned to the application that makes use of the resource file. For the MyHelloWorld project, I changed the signature in the *MyHelloWorld.rsrc* file to `application/x-vnd.dps-myworld`.

After changing the signature, you can choose Save from the File menu of the FileTypes window to save the change—but don't close the window just yet.

Changing the icon

To modify the icon housed in the resource file, double-click on the icon box located in the upper right corner of the Application Type window. Then draw away. That, in brief, is the rest of Step 12.

The Tracker—the software that allows you to work with files and launch applications in the BeOS—displays icons in one of two sizes: 32 pixels by 32 pixels or 16 pixels by 16 pixels. The larger size is the more commonly used, but you've seen the smaller size on occasion. For example, the active application displays its smaller icon in the area you click on to reveal the items in the main menu. For each icon resource, you'll draw both the larger and smaller variants. At the top right of the window back in Figure 2-9, you'll see the editing area for both variants of the icon in the *HelloWorld.rsrc* file. Clicking on either icon displays an enlarged, editable view of that icon in the editing area.

In the top right area of the window, you'll see a total of four actual-size icons. You can refer to these views to see how the 32-by-32 version and the 16-by-16 version will look on the desktop. There are two views of each sized icon to demonstrate how each looks normally and when selected (a selected icon has extra shading added by the BeOS).

You'll treat the large and small versions of an icon separately—editing one version has no effect on the other version. IconWorld will assign the larger version a resource type of 'ICON' and the smaller version a resource type of 'MICN' (for "mini-icon"). Because each icon has both a large and small version, IconWorld will save the two versions of the icon as a single unit.

You'll of course want to replace the original icon with one of your own creation. To do that, use the drawing tools and color palette to erase the old icon and draw a new one. Creating an icon in FileTypes is a lot like creating a small picture in a paint program, so you're on familiar ground here. If you aren't artistically inclined, at least draw a simple icon that distinguishes your program from the other program icons on your desktop. When done, click on the smaller editing area and then draw a smaller version of the icon. That's what I did for the MyHelloWorld example I'm working on. I show off my creative abilities back in Figure 2-9.

If you're developing a program that will be distributed to others (rather than one that will be for your personal use only), you can find or hire someone else to draw a 32-pixel-by-32-pixel and a 16-pixel-by-16-pixel picture for you at a later time.

When you're done with the new icon, choose Save from the File menu of the Icon-O-Matic window, then close the Icon-O-Matic window. Now choose Save from the File menu of the FileTypes window. Close the FileTypes window. This completes Step 12.

Setting Project Preferences

Besides keeping track of the files that hold what will become a program's executable code, a project also keeps track of a variety of project settings. The font in which source code is displayed and the types of compilation errors to be displayed are just two examples of project settings. There's a good chance that most of the copied project's settings will be fine just as they are. You will, however, want to make one quick change. Step 13 specifies that you need to set the name of the application about to be built—you do that in the Settings window.

To display the project Settings window, choose Settings from the Window menu in the project file. On the left side of the window that appears is a list that holds a variety of settings categories. Click on the xxx Project item under the Project heading in this list (where xxx specifies the target, such as PPC for PowerPC). The Settings window responds by displaying several project-related items on the right side of the window.

From the displayed items you'll set the name that will be assigned to the application that gets built from the project. The MyHelloWorld project is used to build a program that gets the name MyHelloWorld. Note that while the file type may look like a program signature (they're both MIME strings), the two aren't the same. The file type specifies the general type of file being created, such as a Be application. The file signature (assigned in the FileTypes window, as discussed earlier) gives an application a unique signature that differentiates the file from all other applications.

After changing the information in the File Name field, click the Save button. This is the only setting change you need to make, though you're of course free to explore the Settings window by clicking on any of the other setting topics in the list.

Testing the Changes

After making all your changes, you'll want to test things out—that's the last step in my list of how to create a new project. Choose Run from the Project menu in the project window's menubar. This will cause the BeIDE to:

1. Compile any files that have been touched (altered) since the last build.
2. Link the compiled code to build an application.
3. Merge resources from the resource file with the built application.
4. Run the application.

To verify that your changes to the resource file were noted by the BeIDE, return to the desktop and look in your project folder. There you should find your new application, complete with its own icon. Figure 2-13 shows the *MyHelloWorld* folder as it looks after my test build of a PowerPC version of the program.

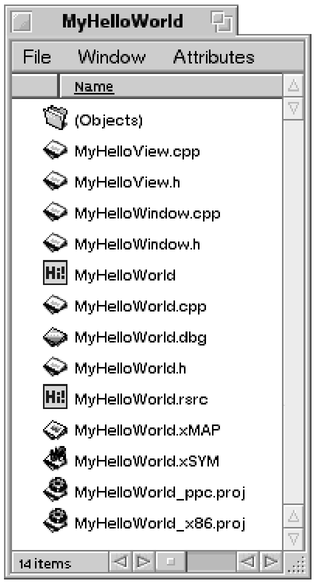


Figure 2-13. The MyHelloWorld project folder after building an application

The *MyHelloWorld* folder in Figure 2-13 shows one folder and two files that I had no hand in creating. The *(Objects)* folder holds, obviously enough, the object code that the BeIDE generated when compiling the project's source code. The *.xMAP* and *.xSYM* files were generated by the BeIDE during the building of the project's application. The files in the *(Objects)* folder and the *.xMAP* and *.xSYM* files aren't directly of use to you—they're used by the BeIDE during linking and debugging. If I were working on a PC and did a build of an Intel version of the *MyHelloWorld* program, my project folder would look a little different. There'd be no *.xMAP* or *.xSYM* files, as these are PowerPC-only. Instead, the information in these files (namely the symbol table and the debugging information for the application) would be contained in the ELF binary itself. There might also be a number of *.d* files, which are temporary dependency files the compilers create (and which may be better disposed of in a future release of the BeOS).

What's Next?

After reading this section you know how to create a project that's ready for your own use. Now you need to make the source code changes and additions that will cut the ties from the original project on which you're basing your new program. So you need to know a lot more about writing BeOS code. The next section starts you in that direction, and the remainder of this book takes you the rest of the way.

HelloWorld Source Code

In the previous section, you saw that a new BeIDE project is usually based on an existing project. Once you have a mastery of BeOS programming, you'll be able to look at existing projects and recognize which one or ones result in a program that bears some similarity to the program you intend to develop. Until that time, it makes sense to use a small project such as the HelloWorld project as your starting point. If you follow my advice and do that, your new project will hold the HelloWorld source code. You got a glimpse of some of that code earlier in this chapter. Because you'll be modifying the HelloWorld source code, you'll find it beneficial to have a good understanding of that code. This section provides you with that.

The HelloWorld/SimpleApp/MyHelloWorld Connection

The HelloWorld project defines three classes: `HelloView`, `HelloWindow`, and `HelloApplication`. Two of these classes, `HelloWindow` and `Hello-Application`, bear a strong resemblance to the `SimpleWindow` and `Simple-Application` classes from the SimpleApp example that was introduced in Chapter 1. Actually, the opposite is true—the `SimpleApp` classes are based on the `HelloWorld` classes.

To create the SimpleApp project, I started with the HelloWorld project. I then followed this chapter's steps for renaming the project files and renaming the application-defined classes. If you think that my following a Chapter 2 process while writing Chapter 1 was a good trick, just wait—there's more! I then wrote SimpleApp such that it became a simplified version of the HelloWorld program. Not many books can lay claim to simplifying what is traditionally the most basic program that can be written for an operating system.

In order to keep this book's first example small and simple, I deleted the *HelloView.cpp* and *HelloView.h* files and removed any references to the class that was defined and implemented in those two files—the `HelloView` class. I also stripped out the `#ifndef` and `#define` preprocessor directives to further simplify things.

Now that the relationship between the Chapter 1 SimpleApp example project and the Be-supplied HelloWorld project has been established, where does the MyHelloWorld project fit in? As you saw in this chapter, the MyHelloWorld project also came about by duplicating the HelloWorld project. I renamed the files and the application-defined classes, but I left all the other code intact. Building an application from the MyHelloWorld project results in a program indistinguishable from the application that gets built from the HelloWorld project (except for the look of the application's icon).

In this section I describe the code that makes up the HelloWorld project. At the end of this section I make a few minor changes to this code. When I do that I'll use the MyHelloWorld project so that I can leave the original HelloWorld project untouched, and so that I can justify the time I invested in making the MyHelloWorld project!

HelloWorld View Class

A view is a rectangular area in which drawing takes place. Drawing is an important part of almost all Be programs, so views are important too. Views are discussed at length in Chapter 4, *Windows, Views, and Messages*, so I'll spare you the details here. I will, however, provide a summary of views so that you aren't in the dark until you reach the fourth chapter.

A view can encompass the entire content area of a window—but it doesn't have to. That is, a window's content area can consist of a single view or it can be divided into two or more views. Note that a view has no visible frame, so when I say that a window can be divided, I'm referring to a conceptual division—the user won't be aware of the areas occupied by views in a window.

A view serves as an independent graphics environment, or state. A view has its own coordinate grid so that the location at which something is drawn in the view can be kept independent of other views that may be present in a window. A view has a large set of drawing characteristics associated with it, and keeps track of the current state of these characteristics. The font used to draw text in the view and the width of lines that are to be drawn in the view are two examples of these characteristics.

The information about a single view is stored in a view object, which is of the `BView` class (or a class derived from the `BView` class). The `BView` class is a part of the Interface Kit.

When a window is created, it doesn't initially have any views attached to (associated with) it. Because drawing always takes place in a view and never directly in a window, any program that draws must define a class derived from the `BView` class. When the program creates a new window object, it will also create a new `BView`-derived object and attach this view object to the Window object.

A class derived from the `BView` class will typically define a minimum of three virtual member functions. That is, such a class will override at least three of the many `BView` member functions. These functions are the constructor function, the `AttachedToWindow()` function, and the `Draw()` function. The `HelloView` class defined in the HelloWorld project does just that:

```
class HelloView : public BView {

public:
                HelloView(BRect frame, char *name);
virtual  void   AttachedToWindow();
virtual  void   Draw(BRect updateRect);
};
```

These member functions, along with several other `BView` member functions, are discussed at length in Chapter 4. Here I'll provide only a brief overview of each.

The purpose of the constructor is to establish the size of the view and, optionally, provide a name for the view (`NULL` can be passed as the second parameter). If a window is to have only a single view, then the first parameter to the constructor is a rectangle of the same size as the content area of the window the view is to be attached to.

The `HelloView` constructor does nothing more than invoke the `BView` class constructor. When a `HelloView` view object is created by the HelloWorld program, the program passes to the `HelloView` constructor the size and name the view is to have. The `HelloView` constructor will in turn pass that information on to the `BView` constructor (as the `rect` and `name` parameters). The third `BView` constructor parameter describes how the view is to be resized as its window is resized. The Be constant `B_FOLLOW_ALL` sets the view to be resized in tandem with any resizing of the window. The final `BView` constructor parameter determines the types of notifications the view is to receive from the system. The Be constant `B_WILL_DRAW` means that the view should be notified when the visible portions of the view change (and an update is thus necessary).

```
HelloView::HelloView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
}
```

You might not be able to fully initialize a view object when creating it—some characteristics of the view may be dependent on the window the view becomes attached to. Thus the existence of the `BView` member function `AttachedToView()`. This function is automatically called by the operating system when the view is attached to a window (covered ahead in the discussion of the

HelloWorld's application class). The characteristics of the view should be included in the implementation of `AttachedToWindow()`.

The HelloView version of `AttachedToWindow()` sets the font and font size that are to be used for any text drawing that takes place within the view:

```
void HelloView::AttachedToWindow()
{
    SetFont(be_bold_font);
    SetFontSize(24);
}
```

The `SetFont()` and `SetFontSize()` functions are member functions of the `BView` class. The function names make the purpose of each obvious, so I won't offer any more information here. You will, however, find plenty of information on drawing strings in Chapter 5, *Drawing*.

In Chapter 1 you saw member functions invoked via an object—as in the `SimpleWindow` object invoking the `BWindow` member function `Show()`:

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

In the `AttachedToWindow()` member function you see that the `BView` member functions `SetFont()` and `SetFontSize()` are invoked without the use of an object.

As a C++ programmer, it should be obvious how a member function can be invoked in this way. If it isn't, read on. The implementation of a member function results in a routine that can be invoked by any object of the type of class to which the function belongs. Therefore, the code that makes up a member function can operate on any number of objects. When a member function includes code that invokes a different member function (as the `HelloView` member function `AttachedToWindow()` invokes the `BView` member function `SetFont()`), it is implicit that the invocation is acting on the current object. So no object prefaces the invocation.

The `BView` class includes a `Draw()` member function that automatically gets called when the window a view is attached to needs updating. The system keeps track of the views that are attached to a window, and if more than one view object is attached to a window, the `Draw()` function for each view object is invoked. The `Draw()` function should implement the code that does the actual drawing in the view.

The HelloView version of `Draw()` simply establishes the starting position for drawing a string of text and then draws that string.

```
void HelloView::Draw(BRect updateRect)
{
    MovePenTo(BPoint(10, 30));
    DrawString("Hello, World!");
}
```

HelloWorld Window Class

In Chapter 1, you received an overview of the `BWindow` class and classes derived from it, so I won't go on at length about those topics here. In fact, I'll barely discuss the `HelloWindow` class at all. If you need a refresher on defining a class derived from `BWindow`, refer back to the SimpleApp example in Chapter 1—there you'll find that the `SimpleWindow` class consists of the same two member functions (a constructor and the `QuitRequested()` function) as the `HelloWindow` class.

```
class HelloWindow : public BWindow {

public:
                HelloWindow(BRect frame);
virtual  bool  QuitRequested();
};
```

In the SimpleApp project, the only thing the `SimpleWindow` class constructor does is invoke the `BWindow` class constructor. That's true for the `HelloWindow` class constructor as well. The first parameter to the `BWindow` constructor, which comes from the `HelloWindow` constructor, sets the size of the window. The second parameter is a string that is used as the window's title. The `SimpleWindow` class passes the string "A Simple Window," while the `HelloWindow` class passes the more appropriate string "Hello." The third parameter is a Be-defined constant that specifies the type of window to be displayed. The final parameter defines the behavior of the window. The `SimpleWindow` class passes the constant `B_NOT_RESIZABLE`, while the `HelloWindow` class passes the result of combining two constants, `B_NOT_RESIZABLE` and `B_NOT_ZOOMABLE`.

```
HelloWindow::HelloWindow(BRect frame)
    : BWindow(frame, "Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE |
                              B_NOT_ZOOMABLE)
{
}
```

You've already seen the code that makes up the `QuitRequested()` function. In the Chapter 1 SimpleApp project, I left this routine from the HelloWorld project unchanged. Recall from that chapter that the purpose of overriding this `BWindow` member function is to cause a mouse click on a window's close button to not only close the window, but to quit the application as well:

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

HelloWorld Application Class

All programs must define a class derived from the `BApplication` class. Like the SimpleApp's `SimpleApplication` class, the HelloWorld's `HelloApplication` class consists of just a constructor:

```
class HelloApplication : public BApplication {

public:
           HelloApplication();
};
```

For the implementation of the `SimpleApplication` constructor, I started with the `HelloApplication` constructor, then stripped out the view-related code. As you look at the `HelloApplication` constructor you should recognize much of the code:

```
HelloWindow  *aWindow;
...
BRect         aRect;

aRect.Set(20, 30, 180, 70);
aWindow = new HelloWindow(aRect);
...
...
aWindow->Show();
```

This snippet begins by declaring a window variable and a rectangle variable. Next, the boundaries of the rectangle that sets the window's size are established. Then a new window object (an object of the `HelloWindow` class) is created. Finally, the `BWindow` member function `Show()` is invoked to display the new window. The code I omitted from the above snippet is the `HelloApplication` constructor's view-related code:

```
HelloView    *aView;

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

aWindow->AddChild(aView);
```

In the above snippet, the size of the rectangle object isn't changed, but its existing boundaries (as set by the prior call to the `BRect` member function `Set()`) are offset. The one parameter in the `BRect` member function `OffsetTo()` is the location to move the upper-left corner of the rectangle to. The constant `B_ORIGIN` tells `OffsetTo()` to shift the rectangle `aRect` from its present location such that its upper-left corner aligns with the origin—the point (0, 0). This has the effect of changing the `aRect` coordinates from (20, 30, 180, 70) to (0, 0, 160, 40). The details of the coordinate system, by the way, are presented in Chapter 4. Next, a new `HelloView` object is created. The `HelloView` constructor sets the size of the view to `aRect` and the name of the view to "HelloView." Finally, the window object `aWindow` invokes the `BWindow` member function `AddChild()` to attach the view to itself. Recall that in order for a view to be of use, it must be attached to a window; in order for drawing to take place in a window, it must have a view attached to it.

You've seen the implementation of the `HelloApplication` constructor in bits and pieces. Here it is in its entirety:

```
HelloApplication::HelloApplication()
    : BApplication("application/x-vnd.Be-HLWD")
{
    HelloWindow  *aWindow;
    HelloView    *aView;
    BRect         aRect;

    aRect.Set(20, 30, 180, 70);
    aWindow = new HelloWindow(aRect);

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

    aWindow->AddChild(aView);

    aWindow->Show();
}
```

You may have noticed that after the window object is created, no drawing appears to take place. Yet you know that the string "Hello, World!" gets drawn in the window. Recall that the updating of a window has the effect of calling the `Draw()` member function of any views attached to that window. When `aWindow` invokes the `BWindow` `Show()` member function, the window is displayed. That forces an update to occur. That update, in turn, causes the `aView` object's `Draw()` method to be invoked. Look back at the implementation of the `HelloView` class `Draw()` member function to see that it is indeed this routine that draws the string.

HelloWorld main() Function

When I wrote the `main()` routine for SimpleApp, the only changes I made to the HelloWorld version of `main()` involved changing `HelloApplication` references to `SimpleApplication` references. For the details of what `main()` does, refer back to Chapter 1. I'll summarize by saying that `main()` performs the mandatory creation of a new application object, then starts the program running by invoking the `BApplication` member function `Run()`:

```
int  main()
{
    HelloApplication  myApplication;

    myApplication.Run();

    return(0);
}
```

Altering the Source Code

To implement the functionality required of your own program, you'll no doubt make some changes and substantial additions to the code of whatever project you start with. After reading a few more chapters, you'll know the details of how to draw just about anything to a window and how to add controls (such as buttons) to a window. At that point you'll be ready to make large-scale revisions of existing code. While at this early point in the book you may not feel ready to make sweeping changes to Be code, you should be ready to make at least minor revisions. In this section, I'll do that to the HelloWorld project.

I'll leave the HelloWorld project intact and instead make the changes to a duplicate project—a project I've name MyHelloWorld. If you haven't followed the steps in this chapter's "Setting Up a New BeIDE Project" section, do so now. Besides getting experience at starting a new project, you'll bring yourself to the point where you can follow along with the code changes I'm about to make. Here they are:

- Change the text that is drawn in the window
- Change the window's title
- Add a zoom button to the window's tab
- Change the size of the window

At the top of Figure 2-14, you see the original window from the HelloWorld program, while at the bottom of the figure you see the new window from the MyHelloWorld program.

The drawing that takes place in the MyHelloWorld window is done by the `MyHelloView` `Draw()` member function. A call to the `BView` member function `DrawString()` writes the string "Hello, World!" to the window. Changing this one `DrawString()` parameter to "Hello, My World!" will cause this new string to be drawn to the window instead. The affected line is in the *MyHelloView.cpp* file, and is shown here in bold type:

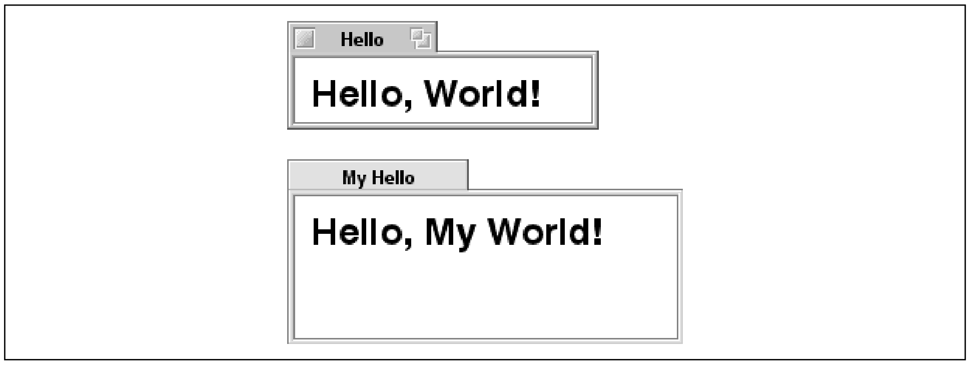


Figure 2-14. The window from HelloWindow (top) and MyHelloWindow (bottom)

```
void MyHelloView::Draw(BRect updateRect)
{
    MovePenTo(BPoint(10, 30));
    DrawString("Hello, My World!");
}
```

The change to the window's title and the addition of a zoom button to the window are both taken care of in the `MyHelloWindow` class constructor, so I'll open the *MyHelloWindow.cpp* file. `MyHelloWindow()` invokes the `BWindow` constructor. The second parameter in the `BWindow` constructor defines the window's title, so changing that parameter from "Hello" to "My Hello" handles the window title change. The last parameter in `BWindow()` is a Be-defined constant, or combination of Be-defined constants, that defines the behavior of the window. In the original `HelloWindow` class, this parameter was a combination of the constants that specified that the window be drawn without a resize knob and without a zoom button (`B_NOT_RESIZABLE | B_NOT_ZOOMABLE`). I'll change this parameter to the one constant `B_NOT_RESIZABLE` so that a window created from the `MyHelloWindow` class will be without a resize knob, but will now have a zoom button:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
}
```

The final change I'll make to the code that came from the HelloWorld project will change the size of the program's window. The `MyHelloApplication` constructor uses a `BRect` object to define the size of the window, so I'll go to this function in the *MyHelloWorld.cpp* file to make the change. In the following version of `MyHelloApplication()`, I've changed the coordinates of the rectangle object `aRect` from (20, 30, 180, 70) to (20, 30, 230, 100) in order to set up a window that's a little larger than the one used in the original HelloWorld program.

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

    aRect.Set(20, 20, 240, 90);
    aWindow = new MyHelloWindow(aRect);

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

    aWindow->AddChild(aView);

    aWindow->Show();
}
```

I can compile all the MyHelloWorld project files, build a new version of the MyHelloWorld program, and run that program by choosing Run from the Project menu in the menubar of the MyHelloWorld project window. If I've done everything correctly, the program will display a window that looks like the one at the bottom of Figure 2-14.