Learning to Program with Haiku

Lesson 20

Written by DarkWyrm

Although our focus in the last few lessons has been on the Interface Kit, another kit receives quite a lot of attention when programming in Haiku: the Storage Kit. This kit is devoted to working with files and folders on disk. We'll be doing a fast and furious crash course through it today. Don't worry, though – it's not very difficult.

Overview of the Storage Kit

The kit can be divided into six basic groups of classes.

Class	Description
BFilePanel	A GUI control used to navigate the filesystem and select a file or folder
BRefFilter	A class for filtering `entry_ref` objects. It is almost always used in combination with BFilePanel.

BQuery	Run a search of the filesystem using attributes. Queries work only on BFS volumes.

BVolume	This is a class which represents a disk volume, which can be a hard disk partition, removable drive, disk image, or network share.
BVolumeRoster	This class provides the means to get information on all volumes available on the system.

BDirectory	BDirectory is a very useful class. It gives you ways to read the contents of a directory, create files, folders, and symbolic links, and it can find a specific entry, such as to ascertain if it exists.
BEntry	This is one of the most-used classes in the Storage Kit. It represents an item in a directory, such as a file or folder. A BEntry can tell you if a path exists, delete it, move it to somewhere else on the same volume, and more.
BEntryList	You won't probably ever use a BEntryList directly. It is a class used to define an interface for reading lists of BEntry objects that derived classes must implement. Currently BQuery and BDirectory are the only objects that do this.
BFile	BFile is another commonly-used class. Unsurprisingly, it represents a file on disk and provides functionality to read data from files and write to them. It provides a friendlier interface than `fread()` and friends. Like BDirectory, it is a child class of BNode, inheriting all of its attribute-related functions.
BNode	A BNode represents the data part of a file. The class provides functions for file locking and working with a file's attributes.
BNodeInfo	Although BNode can do everything BNodeInfo does, BNodeInfo provides an easy interface to common attribute tasks, such as getting a file's type. Of particular note is the static function `GetTrackerIcon()`, which is used to get the icon that would be shown in Tracker.
BPath	BPath gives you simple string path manipulation functions. It doesn't do many of the fancy tricks BString does, but it does do path-specific ones, like converting relative paths to absolute ones and appending entry names.

Class	Description
BStatable	This provides a friendly interface to the information provided by the `stat()` function, including entry type (file, directory, etc.), creation time, modification time, file owner, and more. It isn't used directly, though. BStatable is a pure abstract class, intended to create an interface implemented by child classes. BNode and BEntry are both child classes of BStatable.
BSymLink	BSymLink is next to useless. The only useful function is `ReadLink()`, but because BSymLink is a child of BNode, it doesn't know where it is in the filesystem. The link resolution functions provided by BEntry are much more convenient.

BAppFileInfo	This class isn't used very often unless you're developing an IDE, file browser, or something similar. It is used for getting and setting information specific to applications, such as version number, icons, supported types, and so on.
BMimeType	It won't be often that you'll use this class. It's for parsing MIME strings, getting access to the File Type database, and performing related tasks. The main thing that this is used for is setting the preferred application for a file type.
BResources	Like BAppFileInfo, you probably won't ever use this class unless you're writing a specific kind of app, such as a resource editor. It is for programmatically loading, adding, or deleting program resources.

The Node Monitor	Not actually a class, the Node Monitor is a service of the Storage Kit which can notify you of changes in the filesystem, such as when a volume is mounted or unmounted, when a file is created or deleted, or when the contents of a directory changes.

Project

Using the Interface Kit does have one drawback: if you use one part of it, you almost always have to use the whole paradigm. For example, if you want to load a BBitmap from disk, you have to have a valid BApplication, even if you don't use it. The Storage Kit doesn't have this limitation. In fact, today's project is a simple console application which is simple enough for a lesson but is coded in a style that you would find in a regular Haiku application.

This project, called ListDir, is a simple version of the bash command `ls`. Given a path specified on the command line, our program will list the contents of the directory and the size of each entry in the directory. We will display the "size" of any subdirectories by listing how many entries the subdirectory has.

```
#include <Directory.h>
#include <Entry.h>
#include <Path.h>
#include <stdio.h>
#include <String.h>
```

```
// It's better to use constant global integers instead of #defines because
// constants provide strong typing and don't lead to weird errors like #defines
// can.
const uint16 BYTES_PER_KB = 1024;
const uint32 BYTES_PER_MB = 1048576;
const uint64 BYTES_PER_GB = 1099511627776ULL;

int         ListDirectory(const entry_ref &dirRef);
BString     MakeSizeString(const uint64 &size);


int
main(int argc, char **argv)
{
	// We want to require one argument in addition to the program name when
	// invoked from the command line.
	if (argc != 2)
	{
		printf("Usage: listdir <path>\n");
		return 0;
	}

	// Here we'll do some sanity checks to make sure that the path we were given
	// actually exists and it's not a file.

	BEntry entry(argv[1]);
	if (!entry.Exists())
	{
		printf("%s does not exist\n",argv[1]);
		return 1;
	}

	if (!entry.IsDirectory())
	{
		printf("%s is not a directory\n",argv[1]);
		return 1;
	}

	// An entry_ref is a typedef'ed structure which points to a file, directory,
	// or symlink on disk. The entry must actually exist, but unlike a BFile or
	// BEntry, it doesn't use up a file handle.
	entry_ref ref;
	entry.GetRef(&ref);
	return ListDirectory(ref);
}


int
ListDirectory(const entry_ref &dirRef)
{
	// This function does all the real work of the program

	BDirectory dir(&dirRef);
	if (dir.InitCheck() != B_OK)
	{
		printf("Couldn't read directory %s\n",dirRef.name);
		return 1;
	}
```

```
// First thing we'll do is quickly scan the directory to find the length of
// the longest entry name. This makes it possible to left justify the file
// sizes
int32 entryCount = 0;
uint32 maxChars = 0;
entry_ref ref;

// Calling Rewind() moves the BDirectory's index to the beginning of the
// list.
dir.Rewind();

// GetNextRef() will return B_ERROR when the BDirectory has gotten to the
// end of its list of entries.
while (dir.GetNextRef(&ref) == B_OK)
{
	if (ref.name)
		maxChars = MAX(maxChars,strlen(ref.name));
}
maxChars++;
char padding[maxChars];

BEntry entry;
dir.Rewind();

// Here we'll call GetNextEntry() instead of GetNextRef() because a BEntry
// will enable us to get certain information about each entry, such as the
// entry's size. Also, because it inherits from BStatable, we can
// differentiate between directories and files with just one function call.
while (dir.GetNextEntry(&entry) == B_OK)
{
	char name[B_FILE_NAME_LENGTH];
	entry.GetName(name);

	BString formatString;
	formatString << "%s";

	unsigned int length = strlen(name);
	if (length < maxChars)
	{
		uint32 padLength = maxChars - length;
		memset(padding, ' ', padLength);
		padding[padLength - 1] = '\0';
		formatString << padding;
	}

	if (entry.IsDirectory())
	{
		// We'll display the "size" of a directory by listing how many
		// entries it contains
		BDirectory subdir(&entry);
		formatString << "\t" << subdir.CountEntries() << " items";
	}
	else
	{
		off_t fileSize;
		entry.GetSize(&fileSize);
		formatString << "\t" << MakeSizeString(fileSize);
```

```
			}
			formatString << "\n";
			printf(formatString.String(),name);
			entryCount++;
		}
		printf("%ld entries\n",entryCount);
		return 0;
}


BString
MakeSizeString(const uint64 &size)
{
		// This function just converts the raw byte counts provided by BEntry's
		// GetSize() method into something more people-friendly.
		BString sizeString;
		if (size < BYTES_PER_KB)
				sizeString << size << " bytes";
		else if (size < BYTES_PER_MB)
				sizeString << (float(size) / float(BYTES_PER_KB)) << " KB";
		else if (size < BYTES_PER_GB)
				sizeString << (float(size) / float(BYTES_PER_MB)) << " MB";
		else
				sizeString << (float(size) / float(BYTES_PER_GB)) << " GB";
		return sizeString;
}
```

Going Further

With what we now know about the Interface and Storage Kits, there is a lot that is possible. You might want to go back and expand on a previous project and see what you can do with it. If you haven't tried starting a project of your own yet, this might be a time to seriously consider one. If not, very soon we will spend more than one lesson on a project the size of which you might see in the real world to tie in everything we've been learning.