Learning to Program with Haiku

Lesson 21

Written by DarkWyrm

All of the projects that we have been working on have been small ones which didn't take very much time. Depending on the complexity, real world programming projects can take months in development or more. Today we will start working on a project which is small in comparison to many Haiku programs out there but will take us more than one lesson to complete.

Project Overview

Our project will be a relatively simple one. For about as long as UNIX has been around there has been a command called `fortune` which displays a witty saying of some sort. Often it would be run when the user logged in. We will be writing a fortune program which displays a window instead of printing to the Terminal. It will also improve upon the original because we will not limit it to one fortune file. Instead, it will randomly choose a file in the fortunes directory and choose a random entry from the file. We will also make it easy for the user to get more than one fortune if he wishes.

We will organize our code into two main parts: the GUI and the code which gets a fortune. The fortune code will be one class which is responsible for choosing the fortune file from the fortune directory, getting a random entry from the file, and returning it as a string. The GUI will consist of a multiple line text box for the fortune, a button to close the program, another to get another fortune, and an About button for the purposes of showing who wrote the program and its version. This kind of separation of code based on task is a good practice which makes for reusable code.

BFile: Make Files Sit Up and Bark

Well, maybe we can't make files act like dogs, but the BFile class is an extremely useful one which gives us one place where we can do just about anything with a file: create files and read, write, append, or even erase data, and because it inherits from BNode, we can also add, remove, or change attributes. For those not familiar, attributes are a BeOS specialty: data about a file that is not part of the file's data, but that is for another lesson. Here is a list of the most commonly-used methods available to us via BFile:

Method	Description
`status_t GetSize(off_t *size) const;`	Gets the size of the file and places the value in `size`.
`ssize_t Read(void *buffer, size_t size);`	Attempts to read `size` bytes into `buffer`. The actual number of bytes read is returned.
`ssize_t Write(const void *buffer, size_t size);`	Attempts to write `size` bytes into `buffer`. The actual number of bytes written is returned.
`off_t Seek(off_t offset, int32 seekMode);`	Moves the file read/write pointer to a different location in the file. seekMode can be `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`. These seek modes move the pointer relative to the beginning of the file, the current position, or the end of the file, respectively.
`off_t Position() const;`	Returns the location of the read/write pointer relative to the beginning of the file.

Method	Description
`status_t SetTo(const char *path, uint32 openMode);`	Set the BFile object to the path specified in `path`. There are other versions of this method, as well. See below for more information on this method.
`void Unset();`	Empties the BFile object, releasing any file handles used. This method is automatically called when the BFile object is destroyed.
`status_t InitCheck() const;`	Returns the current error status of the BFile.

Setting a BFile to a location involves a little more than just giving it a file path. There are actually four different versions of `SetTo()`. They take an open mode and either a string containing the path, a BEntry, an entry_ref, or a BDirectory with a relative path in a string. The open mode is a combination of the following flags which includes a read or write mode:

Mode Flag	Description
`B_READ_ONLY`	The BFile can read from, but not write to, the file.
`B_WRITE_ONLY`	The BFile can write to, but not read from, the file.
`B_READ_WRITE`	The BFile can read from and write to the file.
`B_CREATE_FILE`	Create the file if it doesn't already exist.
`B_FAIL_IF_EXISTS`	Insist on creating a new file and failing if it exists.
`B_ERASE_FILE`	If the file exists, erase its data and attributes.
`B_OPEN_AT_END`	Set the read/write pointer to the file's end.

The constructors for BFile have the same arguments as their respective `SetTo()` calls, but they don't return any error codes. Anything can go wrong when dealing with files, so make sure you check the error status returned from `SetTo()` or call `InitCheck()` after creating a BFile. Also, there are a limited number of file handles available on the system at any given time. Each BEntry, BFile, BNode, or BDirectory uses one when set to a path, so `Unset()` or delete them when you're done with them.

Reading and Writing Files with BFile

BFile makes reading files really easy. `Read()` takes an untyped buffer of bytes to receive data from the file and the amount of data to read. When working with text data, normally you will use either a `char` array or, even better, a BString. Here's the way to read a file using BFile and BString:

```
status_t
ReadFile(const char *path)
{
	if (!path)
		return B_BAD_VALUE;

	// Set up the file to read
	BFile file("/boot/home/Desktop/MyFile.txt", B_READ_ONLY);
	if (file.InitCheck() != B_OK)
	{
		printf("Couldn't read the file\n");
```

```
		return B_ERROR;
	}

	off_t fileSize = 0;
	file.GetSize(&fileSize);
	if (fileSize < 1)
	{
		printf("File is empty, so no data to read\n");
		return B_OK;
	}

	// Create a buffer to hold the file data
	BString fileData;

	// We can't directly pass a BString to Read(), so we'll use the BString
	// method LockBuffer() to get a pointer to its internal storage. While the
	// BString is locked, we can't use any of its methods, but we can make
	// whatever changes we want to the internal string array that it uses.
	// LockBuffer() takes an integer of the maximum size that the array will
	// be expected to be. We'll pad the number just in case so that there are
	// no unexpected crashes.
	char *buffer = fileData.LockBuffer(fileSize + 10);

	// Read() will return the number of bytes actually read, but we're going
	// to ignore the value because we're reading in the entire file.
	file.Read(buffer, fileSize);

	// Unlock the BString so we can use its methods again.
	fileData.UnlockBuffer();

	return B_OK;
}
```

Writing files is even easier. `Write()` has the same parameters as `Read()`, but instead of copying from the file to the buffer, data is copied from the buffer to the file.

```
void
WriteFile(const char *path)
{
	if (!path)
	{
		printf("NULL path sent to WriteFile\n");
		return B_BAD_VALUE;
	}

	// Create a file, if needed, and make it both readable and writable
	BFile file(path,B_READ_WRITE | B_CREATE_FILE);
	if (file.InitCheck() != B_OK)
	{
		printf("Couldn't write file &s\n", path);
		return B_ERROR;
	}

	char testString[] = "This is some file data.\nIt's not really important.\n";
	file.Write(testString,strlen(testString));
	return B_OK;
}
```

Starting Our Project: HaikuFortune

- Open Paladin and create a new project using the GUI with MainWindow template.
- Press Alt+N or choose Add New File from the Project menu and create a file called FortuneFunctions.cpp. Make sure that you check the box to also create a corresponding header file.

The first thing we're going to do is design the class which will get the fortune from the fortune directory.

```
#ifndef FORTUNEFUNCTIONS_H
#define FORTUNEFUNCTIONS_H

#include <List.h>
#include <String.h>

extern BString gFortunePath;

class FortuneAccess
{
public:
                  FortuneAccess(void);
                  FortuneAccess(const char *folder);
                  ~FortuneAccess(void);

      status_t    SetFolder(const char *folder);
      status_t    GetFortune(BString &target);
      int32       CountFiles(void) const;
      status_t    LastFilename(BString &target);

private:
      void        ScanFolder(void);
      void        MakeEmpty(void);

      BString     fPath,
                  fLastFile;
      BList       fRefList;
};

#endif
```

There is a reason for each method that we have in this class. First, both versions of the constructor are for convenience in creating a FortuneAccess object regardless of whether or not we know the folder we want to scan when the object is instantiated. `SetFolder()` allows us to change folders, should we have the desire. `GetFortune()` is the main reason we're creating the class in the first place: a reusable object which randomly gets a fortune from a specified folder. `CountFiles()` tells us how many files are available. `LastFilename()` gives us the name of the file from which the most recent fortune came. `ScanFolder()` runs through a directory and compiles a list of available files that – theoretically – have fortunes in them.

`MakeEmpty()` is a cleanup function which deserves a little extra explanation. The list of filenames in the fortune folder that we set is kept as a collection of `entry_ref` objects in a BList. There are two

problems with BList: we have to `static_cast` any time we access an object it holds, and while the BList takes care of any memory allocation it does internally, any items we give to it are not destroyed when the list is freed. This means that we have to manually go through the list, get each item, and free it ourselves. It's a pain in the neck, but, unfortunately, it's all that we have at the moment. There are better solutions out there, but that's for another time. This will work well enough for our purposes right now.

Below is the skeleton code for our class along with what each function needs to do. Your job is to write the code.

```
#include "FortuneFunctions.h"

#include <Directory.h>
#include <Entry.h>
#include <File.h>
#include <OS.h>
#include <Path.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Initialize the global path to a hardcoded value just in case.
// This happens to be different under Haiku than under previous versions
// of BeOS
BString gFortunePath = "/boot/system/data/fortunes";


FortuneAccess::FortuneAccess(void)
{
}


FortuneAccess::FortuneAccess(const char *folder)
{
	SetFolder(folder);
}


FortuneAccess::~FortuneAccess(void)
{
	// Free all items in our list
}


status_t
FortuneAccess::SetFolder(const char *folder)
{
	// Make sure that folder is valid and return B_BAD_VALUE if it isn't.
	// Set the path variable, scan the folder, and return B_OK
}


status_t
FortuneAccess::GetFortune(BString &target)
{
```

```
	// Here's the meat of this class:
	// 1) Return B_NO_INIT if fPath is empty
	// 2) Return B_ERROR if the ref list is empty

	// 3) This line will randomly choose the index of a file in the ref list
	int32 index = int32(float(rand()) / RAND_MAX * fRefList.CountItems());

	// 4) Get a pointer to the randomly-selected entry_ref
	// 5) Create and initialize a BFile object in read-only mode
	// 6) Check to make sure that the BFile's status is B_OK
	// 7) Set fLastFile to the name property of the ref we just
	// 8) Get the file's size.
	// 9) If the file is empty, return B_ERROR.

	// 10) Create a BString to hold the data in the file
	// 11) Create a char pointer that we'll use in BFile::Read.

	// 12) Initialize the pointer using BString::LockBuffer, passing the file's
	//    size + 10 bytes (for safety) as the size. LockBuffer temporarily gives
	//    you access to the BString's internal char array. We'll need this to
	//    be able to read the file's data into the BString.

	// 13) Use BFile::Read() to read the entire file using our new char pointer.
	// 14) Call BString::UnlockBuffer() to invalidate our char pointer and
	//    allow us to use regular BString methods again.

	// 15) Use a loop to manually count the number of record separators in the
	//    fortune file. The separator is the string "%\n", so use a
	//    combination of BString::FindFirst and offsets in a loop to count them.

	// 16) Use this line to randomly choose an entry.
	int32 entry = int32(float(rand()) / RAND_MAX * (entrycount - 1));

	// 17) Use FindFirst again to find the starting offset of this
	//    randomly-chosen entry in the file.
	// 18) Call FindFirst one last time to find the offset of the next separator
	//    so we know how long the fortune is.
	// 19) Create a BString to hold the fortune.
	// 20) Set this new BString to the String() method plus the starting offset
	//    of the BString holding the file data. This will effectively chop out
	//    everything that is before our fortune in the file. It should look
	//    something like this:
	//    BString fortune = filedata.String() + startingOffset;
	// 21) Chop off everything after our fortune in the fortune BString by
	//    calling its Truncate() method.
	//    Hint: length = endingOffset – startingOffset + 2
	// 22) Set the parameter 'target' to our fortune data and return B_OK
}



void
FortuneAccess::ScanFolder(void)
{
	// Use a BDirectory for this. Make sure that it is initialized from fPath
	// properly. Empty the ref list so that we're not adding to an existing
	// list. Use BDirectory::GetNextEntry to get the entry for each file in the
	// folder. Use the BEntry to check to make sure that the entry is a file,
```

```
	// and, assuming so, make a new entry_ref, send it to BEntry::GetRef,
	// and add it to our ref list.
}


void
FortuneAccess::MakeEmpty(void)
{
	// Iterate through the ref list and delete each entry_ref. After doing
	// this, call BList::MakeEmpty().
}


int32
FortuneAccess::CountFiles(void) const
{
	return fRefList.CountItems();
}


status_t
FortuneAccess::LastFilename(BString &target)
{
	// Return B_NO_INIT if the path variable is empty
	// Set the target parameter to our fLastFile property and return B_OK
}
```

Writing and Testing Our Code

Because we're dealing with non-GUI code, it's probably easiest to test all of the code with a quick-and-dirty Terminal application. In your main() function in App.cpp, comment out all of the code except the return value and write code to quickly make sure that everything works correctly. Here are some development tips that should make writing it a little easier:

- Write the destructor and `MakeEmpty()` first.
- Implement `SetFolder()` next.
- `ScanFolder()` should be written now because `GetFortune()` depends on it. Test code in `main()` should just call `SetFolder()` to some path that you want to use for testing. Using `printf()` to print out what `ScanFolder()` is doing, such as the names of each ref scanned, would be good for debugging here.
- Once `ScanFolder()` has been written, start hacking on `GetFortune()`. Once again, use `printf()` to find out what's going on.
- `LastFileName()` can be implemented whenever you feel like – it's not important until we start implementing the GUI.

Once your FortuneAccess class is complete and debugged, you have all that you need to write a better command-line fortune program than fortune itself!

Going Further

We haven't even touched the GUI yet. Think of some possible ways that we could make a simple interface to show the fortune using graphical controls. We'll make the GUI next time and our project will be complete!