Programming with Haiku

Lesson 3

Written by DarkWyrm

In this final lesson on the C++ language, we will be learning about another way to handle errors using language constructs called exceptions and object-oriented file manipulation using streams.

C++ Input / Output Streams

C developers are quite familiar with using `open()`, `close()`, `fprintf()`, and a host of other functions to read and write files and print text to the screen. C++ developers have a similar set of language constructs which make for clearer file operations. You might remember that the BString class overloads the << operator to append information as a string. That usage comes from how C++ deals with file operations.

The way that C++ interacts with I/O streams is more flexible and looks less cluttered. It uses double arrow operators for interaction with streams. << is used for writing to a stream and >> is used for reading. The standard pipes `stdout`, `stdin`, and `stderr` have been replaced with `cout`, `cin`, and `cerr`, respectively, plus an additional output stream for logging, `clog`. Their capabilities and uses have not changed, however. The collection of functions have been turned into methods which can be overloaded when necessary. In short, the C++ way makes it possible to learn a small set of methods which are much more flexible than their C counterparts.

Let's compare the two ways of working with files. Here is an example of a C++ program which uses C functions to read a file and print it to console.

```
#include <stdio.h>

int
FileExists(const char *path)
{
	// This function tests for the existence of a file by trying
	// to open it. There are better ways of doing this, but this
	// will work well enough for our purposes for the moment.

	// If we were given a NULL pointer, we will return a -1 to
	// indicate an error condition.
	if (!path)
		return -1;

	// Attempt to open the file for reading.
	FILE *file = fopen(path,"r");

	// Our return value will be 1 if the file exists and 0 if
	// it doesn't. ferror() will return a nonzero result if
	// there was a problem opening the file and a 0 if the file
	// opened OK.
	int returnValue;

	// ferror will crash if given a NULL pointer.
	if (!file || ferror(file) != 0)
		returnValue = 0;
	else
	{
		fclose(file);
		returnValue = 1;
	}
```

```
	return returnValue;
}


int
MakeTestFile(const char *path)
{
	// Always check for NULL pointers when dealing with strings.
	if (!path)
		return -1;

	// Open the file and erase the contents if it already exists.
	FILE *file = fopen(path,"w");

	if (!file || ferror(file))
	{
		// We have a different error code if we couldn't create the
		// file. This makes it possible for us to know if we messed
		// up by passing a NULL pointer or if there was a
		// file-related error.
		fprintf(stderr,"Couldn't create the file %s\n",path);
		return 0;
	}

	// The stream handles for stdout, stdin, and stderr are already
	// defined for us, so we can use them without any extra work, like
	// in the if() condition above.
	fprintf(file,"This is a file.\nThis is only a file.\n"
			"Had this been a real emergency, do you think I'd "
			"be around to tell you?\n");
	fclose(file);
	return 1;
}


int
main(void)
{
	int returnValue = 0;

	// Let's use a test file in /boot/home called MyTestFile.txt.
	const char *filePath = "/boot/home/MyTestFile.txt";

	// Make the test file if it doesn't already exist and bail out of
	// our program entirely if there is a problem creating it.
	if (!FileExists(filePath))
	{
		returnValue = MakeTestFile(filePath);
		if (returnValue != 1)
			return returnValue;
	}


	printf("Printing file %s:\n",filePath);

	// We got this far, so it's safe to print the file
	FILE *file = fopen(filePath,"r");

	if (!file || ferror(file))
```

```
	{
		fprintf(stderr,"Couldn't print the file %s\n",filePath);
		return 0;
	}

	char inString[1024];

	// fgets will return a NULL pointer when it reaches the end of the
	// file, so this little loop will print the entire file and quit
	// at its end.
	while (fgets(inString,1024,file))
		fprintf(stdout,"%s",inString);

	fclose(file);

	return 0;
}
```

If this program were to be rewritten in C++, it might look something like this:

```
#include <fstream>
#include <iostream>

using namespace std;

int
FileExists(string path)
{
	// This function tests for the existence of a file by trying
	// to open it. There are better ways of doing this, but this
	// will work well enough for our purposes for the moment.

	if (path.empty())
		return -1;

	// Attempt to open the file for reading
	ifstream file;
	file.open(path.c_str());

	// good() returns true if everything's hunky-dory with our file.
	return file.good();
}


int
MakeTestFile(string path)
{
	// Always check for NULL pointers when dealing with strings.
	if (path.empty())
		return -1;

	// Open the file and erase the contents if it already exists.
	ofstream outFile;
	outFile.open(path.c_str());

	// Check to see if we hit any problems.
	if (!outFile)
	{
		// The endl constant is a special constant which represents
```

```
		// an end-of-line character. By using the endl constant
		// instead of just a '\n' sequence, we are programming for
		// portability without thinking about it.
		cerr << "Couldn't create the file " << path << endl;
		return 0;
	}

	outFile	<< "This is a file." << endl
		<< "This is only a file." << endl
		<< "Had this been a real emergency, do you think I'd "
			"be around to tell you?" << endl;

	outFile.close();
	return 1;
}


int
main(void)
{
	int returnValue = 0;

	// Let's use a test file in /boot/home called MyTestFile.txt.
	string filePath("/boot/home/MyTestFile.txt");

	// Make the test file if it doesn't already exist and bail out of
	// our program entirely if there is a problem creating it.
	if (!FileExists(filePath))
	{
		returnValue = MakeTestFile(filePath);
		if (returnValue != 1)
			return returnValue;
	}

	cout << "Printing file " << filePath << ": " << endl;

	// We got this far, so it's safe to print the file.
	ifstream inFile;
	inFile.open(filePath.c_str());

	if (!inFile)
	{
		cerr << "Couldn't print the file %s" << filePath << endl;
		return 0;
	}

	string inString;

	getline(inFile, inString);
	while (!inString.empty())
	{
		// getline() strips end-of-line characters, so we need to
		// add one.
		cout << inString << endl;
		getline(inFile, inString);
	}

	inFile.close();
```

```
        return 0;
}
```

It looks a little strange, but both versions accomplish the same task. The difference is the potential to do more. The same interface can be used to operate on strings using the `istringstream` and `ostringstream` classes. New `iostream` subclasses can be created to operate in new ways, and the `<<` and `>>` operators can be overloaded to better integrate our own classes with C++ streams. There are many cases where we can utilize methods which take the burden of memory management off our backs, such as `getline()`. Let's take a look at some of the methods available to us from the istream and ostream classes.

istream Methods

```
operator >>
```

Extract information from the stream. This is similar to `fscanf()` or `sscanf()`.

```
streamsize gcount() const;
```

Returns the number of bytes read during the last `get()` or `read()` operation.

```
int get();
int peek();
```

Both methods get one character from the stream and return it. `peek()` does so without actually moving the stream read pointer.

```
istream & get(char &c);
```

Gets one character from the stream and places it in `c`.

```
istream & get(char *string, streamsize count);
istream & get(char *string, streamsize count, char delimiter);
```

Reads characters from the stream until one of the following conditions is met: it has read `count - 1` characters, it encounters the end of the file, or, in the case of the `delimiter` version, it encounters the delimiter character.

```
istream & getline(char *string, streamsize count);
istream & getline(char *string, streamsize count, char delimiter);
```

Reads one line from the stream, up to `count` characters or until `delimiter` is encountered.

```
istream & read(char *buffer, streamsize count);
```

Reads `count` bytes from the stream unless the end of the file is reached.

```
streampos tellg();
istream & seekg(streampos position);
istream & seekg(streamoff offset, ios_base::seekdir direction);
```

These methods get and set the position for the next `get()` call. The position to be set can be either absolute (`position`) or relative to the current one (`offset`, `direction`).

ostream Methods

```
operator <<
```

Write to the stream with formatting, like `fprintf()` or `sprintf()`.

```
ostream & put(char c);
```

Write a character to the stream.

```
ostream & write(char *string, streamsize count);
```

Write a string of `count` length to the stream.

```
streampos tellp();
ostream & seekp(streampos pos);
ostream & seekp(streamoff offset, ios_base::seekdir direction);
```

These methods get and set the position for the next `put()` or `write()` call. The position to be set can be either absolute (`position`) or relative to the current one (`offset`, `direction`).

Methods Common to istream and ostream

```
bool good() const;
bool bad() const;
bool fail() const;
bool operator ! () const;
bool eof() const;
```

These functions deal with the error state of the stream. When a file stream comes to its end, the `eof` flag is set, whose state is returned by the `eof()` method. The `bad` flag for a stream is set when an error has occurred which affects the integrity of the stream. The `fail` flag is set when individual operations fail for one reason or another, while the `bad` flag tends to be set for problems that affect operations in general. `bad()` returns true when the `bad` flag is set, but `fail()` returns true when either `bad` or `fail` is set. The `!` operator does the same thing. `good()` isn't really the opposite of `bad()`. Instead, it returns false whenever any of the error flags are set, `bad`, `fail`, or `eof`. In short, if you're reading a file, looping over `while(myStream.good())` will be sufficient to ensure that you can read the file.

```
streamsize width() const;
streamsize width(streamsize wide);
```

Get or set the width of a field. If you are looking to print with a set alignment or a fixed width, you'll need these methods.

```
char fill() const;
char fill(char c);
```

Set or get the fill character used to pad for alignment or fixed-width formatting. The default is a space.

Wow. That's a lot of methods. Luckily, only a handful of these are necessary for just basic file operations like reading or writing a file. The methods in the list above aren't even the full list of available methods. They are just the you'll likely ever need in day-to-day coding.

Formatting with C++ Streams

One example of the increased flexibility that C++ streams provide is formatting. `printf()` and its brethren provide a wealth of formatting options, but `cin` and `cout` have more. They are used the same way that `endl` is used to add an end-of-line character.

Most stream manipulators, unlike `endl`, actually modify the state of the stream to which they are applied. For example, the `boolalpha` manipulator causes boolean values to be converted to their string equivalents. Considering that they are normally converted to the string equivalents of their numeric values (1 for true, 0 for false), this is a nice convenience. By sending a `boolalpha` manipulator to a stream, all future boolean values are converted to "true" or "false." This mode can be turned off by sending a `noboolalpha` manipulator to the stream.

```
#include <iostream>

using namespace std;

int
main(void)
{
	// The hex manipulator causes integers to be displayed in hexidecimal
	cout << boolalpha << hex;
	cout << true << endl;
	cout << 123 << endl;

	// Manipulators can be sent inline like this or on their own,
	// like the above.
	cout << noboolalpha << true << endl;

	return 0;
}
```

The output from this program looks like this:

```
true
7b
1
```

Here is a list of some of the other available manipulators.

Manipulator	Description
boolalpha, noboolalpha	Turns on/off conversion of boolean values to their string equivalents, i.e. true → "true".
dec, hex, oct	Set the numeric base mode to decimal, hexadecimal, or octal.
flush	Flushes the file buffer. Any data which is waiting to be written to the stream is written.
skipws, noskipws	Toggles whitespace skipping. When enabled, this causes reads to skip over tabs, spaces, and newlines. This mode is a huge time-saver when reading configuration files.
showbase, noshowbase	Prepend numbers based on their numeric base. Hexadecimal numbers begin with 0x and octal numbers begin with 0. Decimal numbers have no prefix.

This is by no means a complete list. A more-complete list may be found in any C++ reference, such as that provided at http://www.cplusplus.org/

C++ Exceptions

Exceptions are a way of building error handling into our programs. Haiku doesn't generally use them because they incur a significant performance hit, but knowing about them is nonetheless important.

Use of exceptions centers around three different C++ language elements: `try` blocks, `throw` statements, and `catch` blocks. When a section of code may run into an exceptional error condition, it is placed into a `try` block. Should it have a problem, it will `throw` an exception. This causes code execution to move up the series of nested function calls, known as the **call stack**, until it encounters a `catch` block which handles the kind of exception raised. If it gets to the top of the call stack and still remains unhandled, your program will abort. Here is an example of how exceptions are used:

```
#include <iostream>

using namespace std;

void
SomeFunction(void)
{
	// Let's say something unexpected happens. We'll throw an exception.
	throw 10;
}


int
main(void)
{
	try
	{
		// We placed a function which might raise an exception inside
		// this try block. Should one occur, it will be handled in
		// the catch block.
```

```
		SomeFunction();
	}
	catch (int error)
	{
		// Should an exception get to the top of the call stack, it
		// causes the program to completely abort, which is a bad
		// thing. If you have a try block, you should definitely
		// have a catch block after it.
		cout  << "An unusual error occurred. Exception number "
			<< error << endl;
	}

	return 0;
}
```

Exceptions are rarely used in Haiku programming because the API provides sufficient error-handling capabilities and, as mentioned before, because exception-handling causes a significant performance hit.

Going Further

- Look over the other manipulators in another reference. What kinds of cool things could be done with them?
- If you were going to design a simple memory database with records having a fixed size, how could you represent different kinds of data? What kind of STL containers could you use? How would you read a record? How would you go about writing one? Deleting one? How about saving and loading the thing?