# The BinaryFile Class

For Project 4, we will provide you with a class named BinaryFile that you can use to write data to and read data from a file on disk. The data will be stored on disk in binary form as opposed to a newline-delimited text form.

You can think of a binary disk file as an analog of standard C++ vector<char>, except that the data is stored on disk instead of in RAM. The disk file starts out empty (just like a vector), but can be expanded to any size up to the capacity of your hard drive when you write data into the file.

Here's how you might create a new binary file:

```
#include "BinaryFile.h"

int main()
{
	BinaryFile bf;                                  // create a BinaryFile object

	bool success = bf.createNew("myfile.dat");  // create a new file
	if (!success)
		cout << "Error! Unable to create myfile.dat\n";
	else
		cout << "Successfully created file myfile.dat\n";
	...
}  // bf's destructor closes the file
```

Note: If a file of the given name already exists, *createNew()* will wipe out the contents of that file, leaving it empty (0 bytes long) and opened ready for use.

Here's how you would open an existing binary data file that was created earlier (without wiping out its contents upon opening it):

```
#include "BinaryFile.h"

int main()
{
	BinaryFile bf;

	bool success = bf.openExisting("myfile.dat");
	if (!success)
		cout << "Error! Unable to find myfile.dat\n";
	else
		cout << "Successfully opened existing file myfile.dat\n";
	...
}  // bf's destructor closes the file
```

Once you have opened a binary data file, you can write data into the file at any offset you want. (The offset is the number of bytes from the beginning of the file, which is at offset 0.) To write some data, you use one of the two forms of the *BinaryFile*::*write()* method:

```
	bool write(const char* s, size_t length, BinaryFile::Offset toOffset);
	bool write(const SomeType& x, BinaryFile::Offset toOffset);
```

In both forms, the last argument is an integer, the number of bytes from the start of the file at which to start writing data. (BinaryFile::Offset is a typedef for a large integer type.) In the first form, the data written will be `length` number of characters starting with the character pointed to by `s`. In the second form, *`SomeType`* may be one of many possible types (e.g., int, double, BinaryFile::Offset, a struct consisting of an int and two character arrays, and many others); this call writes the value of `x` to the disk file (in internal binary form) .

For example, the following code writes a 5-byte string "David" into offsets 0-4 of the binary file and then writes an int whose value is 987654321 into offsets 7-10 of the binary file (7-10 because on our machine, an int is 4 bytes long in internal binary form:

```
int main()
{
	BinaryFile bf;
	if (bf.createNew("myfile.dat"))
	{
		bool success;
		success = bf.write("David", 5, 0);  // write 5 chars from "David"
		if (!success)
			cout << "Error writing 'David' to slots 0-4 of file!\n";

		int i = 987654321;
		success = bf.write(i, 7);
		if (!success)
			cout << "Error writing integer i to slot 7-10 of file!\n";
	}
}
```

The resulting binary data file would be exactly 11 bytes long and contain the following data. For clarity, the top row shows the offset where each piece of data can be found in the binary data file (but this is only for illustration; the offsets would not be stored in the file, of course).

| Offsets | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Values | 'D' | 'a' | 'v' | 'i' | 'd' | ?? | ?? | 987654321 | | | |

Notice that the code above wrote five bytes of data to the start of the data file (between offsets 0-4), then wrote the value of the integer variable i at offsets 7-10. Since the program did not explicitly write any data to offsets 5 and 6 of the binary data file, these bytes in the file are unknown and their values could be anything (this is shown by the ??s). Notice that we did not ask to write a sixth character from the string, so no zero byte was written to offset 5. We are not showing exactly what byte values are at each of offsets 7 through 10, because the internal binary form of a 4-byte int may be different on different machines (if you're curious about the details, which you don't need to know for this project, see http://en.wikipedia.org/wiki/Endianness). Just know that if we read back an int starting from offset 7, that int's value will be 987654321.

You may write over existing data or append new data to an existing data file as well. For example, suppose we ran the following code after running the example just above.

```
int main()
{
	BinaryFile bf;
	if (bf.openExisting("myfile.dat")) // open previously-created data file
	{
		if ( ! bf.write("Carey", 5, 2))
			cout << "Error writing string to file!\n";

		if ( ! bf.write("Dog", 3, 11))
			cout << "Error writing string to file!\n";
	}
}
```

Upon completion of this second piece of code, the resulting binary data file would contain the following data:

| Offsets | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Values | 'D' | 'a' | 'C' | 'a' | 'r' | 'e' | 'y' | 987654321 | | | | 'D' | 'o' | 'g' |

As you can see, the string "Carey" overwrites the last three characters of "David" and replaced the two unknown characters in byte slots 5 and 6 of the file. Moreover, we expanded the file by three bytes by writing "Dog" in slots 11-13.

In addition to writing simple C strings and integers to the data file, you can also write objects of other basic types (e.g., bool, char, int, long, unsigned int, float, double, BinaryFile::Offset). In addition, you can write arrays and structs/classes consisting of these types, which can themselves contain arrays or structs/classes of these types, etc., subject to the following restrictions:

- You must not write pointers or arrays or structs/classes containing pointers.
- You must not write objects of a struct/class type containing any virtual functions.
- You must not write objects of a struct/class type containing **both** at least one public and at least one private data member. It's OK if **all** data members are public (as is typically done with a C-like struct) or **all** data members are private (as is typically done with a C++ class with interesting behavior).
- You must not write objects of a struct/class type with a destructor, copy constructor, or assignment operator that was declared and implemented by the author of the class, not the compiler. In particular, you must not write C++ strings, vectors, lists, etc.; they do not have compiler-generated destructors, for example.

Simple C-like structs not containing pointers can be written. For example, the following code saves a Student struct to a binary data file:

```
struct Student
{
	char first[7+1];	// first name up to 7 chars long; 1 char for '\0'
	int studentID;
	float GPA;
};

int main()
```

```
{
	BinaryFile bf;

	if (bf.createNew("student.dat"))
	{
		Student s;
		strcpy(s.first, "Carey"); // this is the way to copy a C string
		s.studentID = 989105343;
		s.GPA = 3.62;

		if ( ! bf.write(s, 0))
			cout << "Error writing Student struct to file!\n";
		else
			assert(bf.fileLength() == sizeof(Student));
	}
}
```

The assertion should never fail; the number of bytes in the binary data file, which is what the *fileLength* method returns, should be the number of bytes in a Student object, because all we ever wrote was one Student object starting at offset 0. On most machines, this length would be 16, and the file would contain the following data. For clarity, the top row shows the offset where each piece of data can be found in the binary data file, and the second row shows which field in the struct each piece of data is associated with. Neither of the top two rows would actually be stored in the binary data file; they are for illustration only. Only the third row, labeled Values, would be stored in the data file. Note that the strcpy did not touch element 6 and 7 of first. The exact values of each byte at offsets 8 through 15 depend on our machine's internal representation of ints and floats.

| Offsets | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Field | first | | | | | | | | studentID | | | | GPA | | | |
| Values | 'C' | 'a' | 'r' | 'e' | 'y' | '\0' | ?? | ?? | 989105343 | | | | 3.62 | | | |

Once you have written data to a binary data file, you can read data from the file. To read previously-written data, you use the *BinaryFile::read()* method, which takes one of two forms:

```
bool read(char* s, size_t length, BinaryFile::Offset toOffset);
bool read(SomeType& x, BinaryFile::Offset toOffset);
```

In both forms, the last argument is an integer, the number of bytes from the start of the file at which to start reading data. In the first form, `length` number of characters will be read into storage starting at the character pointed to by `s`. In the second form, *`SomeType`* may be any type that is allowed to be written); this call reads a value from the disk file (in internal binary form) and stores it in `x`.

Here's an example program that opens the data file created in the previous example and reads several of the previously-saved values:

```
int main()
{
	BinaryFile bf;
	if (bf.openExisting("student.dat"))
	{
```

```
                Student s;
                if ( ! bf.read(s, 0))
                        cout << "Error reading Student struct from file!\n";
                else
                {
                        cout << "First name: " << s.first << endl;
                        cout << "Student ID: " << s.studentID << endl;
                        cout << "GPA       : " << s.GPA << endl;
                }
        }
}
```

And here's another example that just reads in the *studentID* value that had been previously stored at offsets 8-11:

```
int main()
{
        BinaryFile bf;
        if (bf.openExisting("student.dat"))
        {
                int studID;
                if ( ! bf.read(studID, 8))
                        cout << "Error reading integer from file!\n";
                else
                        cout << "Student ID: " << studID << endl;
        }
}
```

Notice how we are able to read in just a piece of the larger struct (just the integer studentID value) as long as we knew its offset (8) and its type (int). As far as the binary data file is concerned, it just holds a bunch of bytes, so you can read any data from any offset in the file you like (but the results might be nonsense if you don't pay attention to offest and type; bf.read(studID, 5) would put an unusual int value in studID, both because the bytes starting at offset 5 were not written as an int, and because of the unknown values at offsets 6 and 7).

If you attempt to read data that does not yet exist in the file (i.e., past the end of the file), the *read()* method will return false. For example, what if we tried to read the student's GPA from offsets 13-16 rather than from 12-15 where it is stored:

```
int main()
{
        BinaryFile bf;
        if (bf.openExisting("student.dat"))
        {
                float GPA;
                if ( ! bf.read(GPA, 13))
                        cout << "Error reading float from file!\n";
                else
                        cout << "GPA: " << GPA << endl;
        }
}
```

The above would write the error message, since it attempts to read 4 bytes of data from locations 13-16 in the data file. However, our previously-created data file only has a total of 16 bytes (numbered 0 through 15):

| Offsets | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Fields | First | | | | | | | | studentID | | | | GPA | | | |
| Values | 'C' | 'a' | 'r' | 'e' | 'y' | '\0' | ?? | ?? | 989105343 | | | | 3.62 | | | |

Therefore, the *read()* method would fail, since it would only be able to read 3 of the 4 bytes it requested (bytes 13,14, and 15). Byte 16 does not exist in the current data file.

When a BinaryFile is destroyed, the file is closed, which ensures that the data written to the file is indeed safely saved. If you wish to close a BinaryFile earlier, perhaps to use the same object when opening the file later, you can call *BinaryFile::close()*, as in this silly example:

```
void f(BinaryFile& bf, string s)
{
         // c_str() is a string method that returns a const char*
       assert(bf.write(s.c_str(), s.size(), 10));  //  offsets 0 to 9 unknown
       if (s.size() >= 4)
       {
              char buffer[4+1];
              assert(bf.read(buffer, 4, 11);
              buffer[4] = '\0';
              if (strcmp(buffer, "avid") == 0))
                     bf.close();
       }
}

int main()
{
       BinaryFile testf;
       if (testf.createNew("test.dat"))
              f(testf, "David");
       if (testf.isOpen())
              assert(testf.write("Hello", 5, 0));
       else
       {
              if (testf.openExisting("test2.dat"))
                     cout << testf.fileLength() << endl;
       }
}
```

Notice that the method *BinaryFile::isOpen()* lets you test whether the file is open.

## *What can you do with binary files?*

Using the *BinaryFile* class you can implement a disk-based version of virtually any data structure that can be stored in RAM. There are two differences between RAM-based and disk-based data structures:

1. ***You*** must explicitly decide where to store each data structure in the binary data file. In contrast, when you create a new variable (or use a *new* expression to

allocate a dynamic object) in RAM, C++ decides where in memory to put the object.

2. You can't use pointers in a binary data file. Instead, you must use offset values (of type BinaryFile::Offset) to specify where in the data file a piece of data is located.

So, for example, here's how we might implement a simple linked list of nodes (called *DiskNode*s) in a binary data file. In this example, we assume we're using a machine where a BinaryFile::Offset value is 4 bytes long and a DiskNode is 8 bytes long. We'll store our head pointer as a BinaryFile::Offset variable in bytes 0-3 of the binary file. We'll store our first DiskNode in bytes 4-11 of the binary file. Finally, we'll store our second DiskNode in bytes 12-19 of the binary file.

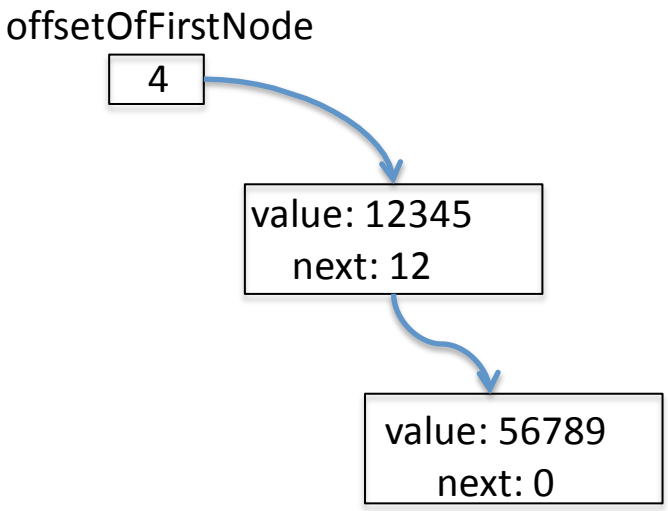
Normally we wouldn't use hard-coded offsets like 0, 4 and 12 to specify locations in the file. However, for the purposes of this example, we'll do so. In more carefully written code, `sizeof(BinaryFile::Offset)` and `sizeof(DiskNode)` would be the way we'd talk about the number of bytes occupied by objects of the indicated types.

```
struct DiskNode
{
	DiskNode(int v, BinaryFile::Offset n) : value(v), next(n) {}
	int value;
	BinaryFile::Offset next;   // instead of a DiskNode *
};

int main()
{
	BinaryFile bf;
	if (bf.createNew("linkedlist.dat"))
	{
		  // First, we write the head "pointer" at the start of the file.
		  // this indicates that our first node is at offset 4 in the
		  // binary data file
		BinaryFile::Offset offsetOfFirstNode = 4;   // like a head pointer
		bf.write(offsetOfFirstNode, 0);

		  // Save the first node at offset 4 in the binary file.
		  // Note that the head "pointer" at the start of the file
		  // will "point" to this node.  Also note that we specify that
		  // the next node is at offset 12 in the file.
		DiskNode firstNode(12345, 12);
		bf.write(firstNode, 4);

		// Write the next node at offset 12 in the file.  Note that
		// we use a next value of zero.  This is our choice for the
		// equivalent of nullptr to indicate the end of the linked list.
		// The value zero as our choice implies we never will put
		// a DiskNode itself at offset 0 if it's part of a linked list.
		DiskNode secondNode(56789, 0);
		bf.write(secondNode, 12);
	}
}
```

This would result in the following being saved to the data file:

| Offsets | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Fields | offsetOfFirstNode | | | | firstNode.value | | | | firstNode.next | | | | secondNode.value | | | | secondNode.next | | | |
| Values | 4 | | | | 12345 | | | | 12 | | | | 56789 | | | | 0 | | | |

And would represent the following logical data structure:




So as you can see, you can implement complex data structures directly within disk files. You'll need to use a technique like this to implement your own disk-based open hash table for this project!