ITP 435 Assignments

RLE Files

For this part, you won't need to write your own unit tests as we've already given you a comprehensive set of tests.

CreateArchive

The code for creating/extracting files archive goes in `RleFile.cpp` in the `RleFile` class. The `CreateArchive` takes in a file name as input and compresses that file into our custom RLE format, and then outputs a file in a format described shortly.

`CreateArchive` needs to open the source file in binary mode. In binary mode, we can load all the contents of the file as one big array. This makes it easy to then compress the data using our existing functions.

Here is one way to open a file in binary mode and copy all its content into a character array:

```
// Requires <fstream>
std::ifstream::pos_type size;
char* memblock = nullptr;
// Open the file for input, in binary mode, and start ATE (at the end)
std::ifstream file (source, std::ios::in|std::ios::binary|std::ios::ate);
if (file.is_open())
{
    size = file.tellg(); // Save the size of the file
    memblock = new char[static_cast<unsigned int>(size)];
    file.seekg(0, std::ios::beg); // Seek back to start of file
    file.read(memblock, size);
    file.close();

    // File data is now in memblock array
    // (Do something with it here...)

    // Make sure to clean up!
```

```
    delete[] memblock;
}
```

After the `file.read` call, all the data will be loaded and ready into `memblock`, and we also have the size of the file (which is the number of bytes in `memblock`). Since `memblock` is a character array, you will first need to copy that data into a `std::vector<int8_t>` in order to use the `Compress` function. Conveniently, there is already an `RleData` member `RleFile`! So, once you load the data into the array you can call `Compress` on the `mRleData` member.

> **WARNING**
> This approach for loading files is ***ONLY*** when you need a file in binary mode. If you just need to read text (such as on a later assignment), please do not use this code and instead open the file as normal and use `std::getline` as you usually would for a text file.

Once you have the compressed RLE data, you must write the data to a file. You should just append `".rl1"` to the source file name. So, if the original file is `"data/rle.bmp"`, the archive's file name should be `"data/rle.bmp.rl1"`.

To open a file for writing in binary mode, use code like this:

```
// Open the file for output, in binary mode, and overwrite an existing file
std::ofstream arc(filename, std::ios::out|std::ios::binary|std::ios::trunc);
if (arc.is_open())
{
    // Use arc.write function to write data here
}
```

The `ios::trunc` flag specifies that we want to overwrite the file if it already exists. But before we can implement the code to write out the file, we need to look at the RLE file format.

This diagram shows what a sample RL1 file might look like. In this case, the source file was `"data/a.out"` (so the output file would be `"data/a.out.rl1"`):

Address	0	1	2	3	4	5	6	7
	File Signature				Original File Size			
0x00	‘R’	‘L’	‘E’	0x01				
File Name Length	File Name (Variable Length)							
0x08	10	‘d’	‘a’	‘t’	‘a’	‘/’	‘a’	‘.’
0x16	‘o’	‘u’	‘t’					
...	RLE Data (Variable Length)							
...								
EOF								

The file starts out with a header that tells us a bit about it. The first four bytes are magic numbers that identify the file type as an RLE version 1 file.

The next part of the header (four more bytes) is the original size of the source file.

Next, there’s one byte for the number of letters in the file name.

Finally, the header has the name of the file. Keep in mind that the length of the file name is variable, it will not always be the same number of bytes.

In `RleFile`, there is a `mHeader` member variable which has all the data fields corresponding to the header. You should save the values into `mHeader` before writing to the final file.

After the header, the rest of the file contains the compressed RLE data. Generally, unless the original file was extremely small, this will be the bulk of the data in the file.

To write binary data into a file, you need to use the write member function of ostream. This function takes two parameters: a pointer to the data you want to write, and the number of bytes to write. This is easy enough to use if you have a character buffer. For instance, to writing the file signature is just this:

```
arc.write(mHeader.mSig, 4);
```

But what about the file size, which is an int? If you just take the address of it with an &, you’ll get an int*, and it won’t work:

```
arc.write(&(mHeader.mFileSize), 4); // Compile error :(
```

Since the function expects a char, *you must reinterpret_cast this int* to a char*. Then the function will be happy:

```
arc.write(reinterpret_cast<char*>(&(mHeader.mFileSize)), 4); // Works!
```

We pass in 4 as the second parameter because the size of an int is `4` bytes.

Also, to write a `std::string` to the file, use `c_str()` to convert it to a C-style string.

Once you write the header, you'll need to write each byte in the mData vector that contains the compressed data.

If you write `CreateArchive` correctly ***and*** your Part 1 compression code is correct, you should pass all the given "File compression" tests in `StudentTests.cpp`. If you pass some of the tests, but not others, it's probably an issue with your Part 1 code. But if you don't pass any, you probably have a problem with `CreateArchive`. Either way, if it doesn't work, check the post about debugging tips for PA1 on Piazza.

If you get very stuck on this part, you can try `ExtractArchive` instead (since decompression is easier to implement).

ExtractArchive

The `ExtractArchive` is roughly the opposite of `CreateArchive`. It loads in an already compressed archive, and then reconstructs the original file based on the data. First, load in the entire file into a memblock like before.

Then, copy the values from `memblock` into the different members of `mHeader`. This requires requires a bit of pointer trickery. You know that the file size starts at the 5th byte, or `memblock[4]`. And you know that `int`s are a total of 4 bytes. So, to grab those 4 bytes and store them into an integer you can use:

```
mHeader.mFileSize = *(reinterpret_cast<int*>(&memblock[4]));
```

What we are doing is getting the pointer to the 5th byte, casting that pointer to an integer pointer, and then dereferencing the integer pointer to get the file size value.

Similarly, grab the length of the file name (though it's just a `char` which is 1 byte), and use that to determine how many characters you need to read in to get the original file name. Once you have the file name, the remaining data in the file is the RLE compressed data, which you should put in a vector to pass into `Decompress`.

Once you have the decompressed data, open a file for writing in binary mode with the original file name. Then just write out all the decompressed data into this file (with no special header or anything), and you should have the original file reconstructed.

If you write `ExtractArchive` correctly and your Part 1 decompression code is correct, you'll pass all the given "File decompression" tests in `StudentTests.cpp`.

Graded Tests Progress

At this point, if everything works, the graded tests on GitHub Actions should give a score of 72/75.

Once you've pushed this code, you're ready to move on to part 3.