Sample Code

Compressing and decompressing files with stream compression

Perform compression for all files and decompression for files with supported extension types.

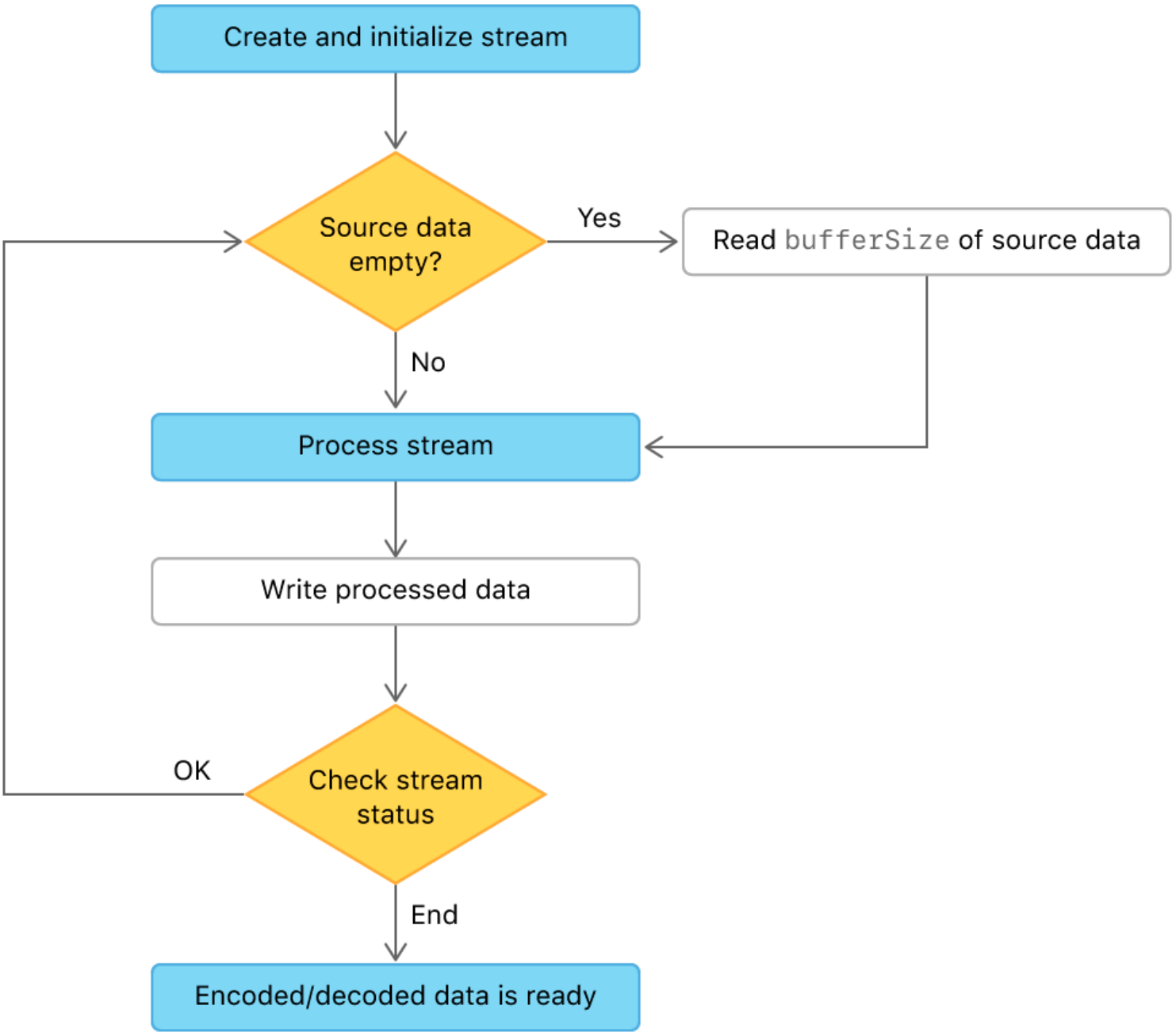
Download

macOS 13.0+ | Xcode 14.3+

Overview

This sample code project uses the Compression framework to encode (compress) and decode (decompress) files when the user drags and drops them onto the app window. The app decompresses files with extensions that match one of four supported compression algorithms: `.lz4`, `.zlib`, `lzma`, or `.lzfse`, and compresses all other files, regardless of their extension. The app writes the encoded or decoded result to the temporary directory that the `NSTemporaryDirectory` function returns.

The code in this sample is useful in applications that store or transmit files, such as PDF or text, where saving or sending smaller files can improve performance and reduce storage overhead. This sample apps implements *stream compression*, where it reads chunks of data from a source buffer repeatedly to compress or decompress data, and appends each chunk to a destination buffer.



Because the sample app performs the encoding and decoding in a background thread, it's able to remain interactive and update the user interface with progress of the operation with a `ProgressView`. Stream compression also enables tasks such as:

- Decoding a compressed stream into a buffer, with the ability to grow that buffer and resume decoding if the expanded stream is too large to fit, without repeating any work.
- Encoding a stream as pieces of it become available, without ever needing to create a buffer large enough to hold all the uncompressed data at one time.

This sample code project includes implementations that use the Compression Swift API and C API. The Compression C API is suitable for use with Swift and Objective-C and for older operating systems that's don't support the Swift API.

Select a compression algorithm

The app uses the `Algorithm.lzfse` algorithm, which provides the compression ratio of zlib level 5, but with much higher energy efficiency and speed (between 2x and 3x) for both encode and decode operations.

```
let encodeAlgorithm = Algorithm.lzfse
```

For apps that require interoperability with non-Apple devices, use `Algorithm.zlib` instead. For more information about other compression algorithms, see `compression_algorithm`.

Distinguish between compressed and uncompressed files

The sample code uses a file's path extension to infer whether a file is already compressed, or if the file needs to be compressed. To simplify this process, the project includes a failable initializer in an extension to the Compression framework's `Algorithm` enumeration.

```
extension Algorithm {
    init?(name: String) {
        switch name.lowercased() {
            case "lz4":
                self = .lz4
            case "zlib":
                self = .zlib
            case "lzma":
                self = .lzma
            case "lzfse":
                self = .lzfse
            default:
                return nil
        }
    }
}
```

The following code uses the new initializer to define the `algorithm` and `operation` constants:

```
let algorithm: Algorithm
let operation: FilterOperation

if let decodeAlgorithm = Algorithm(name: url.pathExtension) {
    algorithm = decodeAlgorithm
    operation = .decompress
```

```
    message = "Decompressing \(url.lastPathComponent)"
} else {
    algorithm = self.encodeAlgorithm
    operation = .compress
    message = "Compressing \(url.lastPathComponent)"
}
```

Define the source and destination file handles

The sample uses `FileHandle` instances to read from the source file and write to the destination file. Use optional binding to define the required file handles.

```
if
    let sourceFileHandle = try? FileHandle(forReadingFrom: url),
    let sourceLength = FileHelper.fileSize(atURL: url),
    let fileName = url.pathComponents.last,
    let fileNameDeletingPathExtension = url.deletingPathExtension().pathComponents.l
    let destinationFileHandle = FileHandle.makeFileHandle(forWritingToFileNameInTemp
        operation == .compress
            ? fileName + self.encodeAlgorithm.pathExtension
            : fileNameDeletingPathExtension) {
```

If the optional binding succeeded, the destination file handle points to the source filename by appending the respective compression algorithm extension, or removing the extension in the case of decompression.

For example, the compressed source file `MyCompressedFile.PDF.lzfse` would have a decompressed destination of `MyCompressedFile.PDF`; and the uncompressed source file, `MyRawFile.PDF`, would have a compressed destination of `MyRawFile.PDF.lzfse`.

Perform streaming compression

The `streamingCompression(operation:sourceFileHandle:destinationFileHandle:algorithm:progressUpdateFunction:)` functions iterate over the source data and encodes or decodes data in blocks based on the length that `bufferSize` defines. The methods write the result into the destination buffer, and write the destination buffer data to the destination file handle. The Swift and C API functions have slightly different signatures.

The following is the function declaration for the Swift version of the streaming compression function:

```
static func streamingCompression(operation: FilterOperation,
                                 sourceFileHandle: FileHandle,
                                 destinationFileHandle: FileHandle,
                                 algorithm: Algorithm,
                                 progressUpdateFunction: (UInt64) -> Void) {
```

The following is the function declaration for the C API version of the streaming compression function:

```
static func streamingCompression(operation: compression_stream_operation,
                                 sourceFileHandle: FileHandle,
                                 destinationFileHandle: FileHandle,
                                 algorithm: compression_algorithm,
                                 progressUpdateFunction: (UInt64) -> Void) {
```

The sample code passes the source and destination file handles, with the operation and algorithm values to either `streamingCompression` function based on the value of the Boolean constant, `useSwiftAPI`:

```
if useSwiftAPI {
    Compressor.streamingCompression(operation: operation,
                                    sourceFileHandle: sourceFileHandle,
                                    destinationFileHandle: destinationFileHandle,
                                    algorithm: algorithm) { progress in
        DispatchQueue.main.async {
            self.progress = Double(progress)
        }
    }
} else {
    Compressor.streamingCompression(operation: operation.rawValue,
                                    sourceFileHandle: sourceFileHandle,
                                    destinationFileHandle: destinationFileHandle,
                                    algorithm: algorithm.rawValue) { progress in

        DispatchQueue.main.async {
            self.progress = Double(progress)
        }
    }
}
```

Create the output filter using the Swift API

The `OutputFilter` instance specifies the operation and the compression algorithm. The final initializer parameter is a closure the instance calls as it writes each encoded or decoded block of data to the destination file handler.

```
do {
    let outputFilter = try OutputFilter(operation,
                                        using: algorithm) {
        (data: Data?) -> Void in
        if let data = data {
            destinationFileHandle.write(data)
        }
    }
```

Compress or decompress the dropped file using the Swift API

The Swift streaming compression function iterates over the source data and calls the `readData(ofLength:)` method to copy `bufferSize` chunks to `subdata`.

```
while true {
    let subdata = sourceFileHandle.readData(ofLength: bufferSize)

    progressUpdateFunction(sourceFileHandle.offsetInFile)

    try outputFilter.write(subdata)
    if subdata.count < bufferSize {
        break
    }
}
```

Create a destination buffer using the C API

The C API streaming compression function allocates the destination buffer based on the `bufferSize` constant.

```
let destinationBufferPointer = UnsafeMutablePointer<UInt8>.allocate(capacity: buffer
defer {
    destinationBufferPointer.deallocate()
```

```
}
```

Create a compression stream using the C API

The `compression_stream` structure defines the source and destination pointers and sizes. The following code declares and initializes the compression stream:

```
let streamPointer = UnsafeMutablePointer<compression_stream>.allocate(capacity: 1)
var status = compression_stream_init(streamPointer, operation, algorithm)
guard status != COMPRESSION_STATUS_ERROR else {
    fatalError("Unable to initialize the compression stream.")
}
```

To prevent memory leaks, the following code calls `compression_stream_destroy(_:)` to free the memory that the stream initialization function allocated. A defer block frees the memory even if the `streamingCompression` method exits early:

```
defer {
    compression_stream_destroy(streamPointer)
    streamPointer.deallocate()
}
```

The sample code sets up the initialized stream by defining its source and destination sizes and destination pointer:

```
streamPointer.pointee.src_size = 0
streamPointer.pointee.dst_ptr = destinationBufferPointer
streamPointer.pointee.dst_size = bufferSize
```

Read the source file data iteratively using the C API

A `repeat-while` loop manages the read-encode/decode-write process. If the stream's source size is zero, the code reads a block of data from the source file handle and points the stream's source pointer to that data. If the read data is shorter than the buffer size, the code infers that it's reading the last block of the source file and sets the stream's status to `COMPRESSION_STREAM_FINALIZE`:

```
var sourceData: Data?
repeat {
```

```
    var flags = Int32(0)

    // If this iteration has consumed all of the source data,
    // read a new buffer from the input file.
    if streamPointer.pointee.src_size == 0 {
        sourceData = sourceFileHandle.readData(ofLength: bufferSize)

        streamPointer.pointee.src_size = sourceData!.count
        if sourceData!.count < bufferSize {
            flags = Int32(COMPRESSION_STREAM_FINALIZE.rawValue)
        }
    }
```

Compress or decompress the dropped file using the C API

The `compression_stream_process(_:_:)` function encodes or decodes the current block.

```
if let sourceData = sourceData {
    let count = sourceData.count

    sourceData.withUnsafeBytes {
        let baseAddress = $0.bindMemory(to: UInt8.self).baseAddress!

        streamPointer.pointee.src_ptr = baseAddress.advanced(by: count - streamPoint
        status = compression_stream_process(streamPointer, flags)
    }
}
```

On return, `destinationBufferPointer` points to the encoded or decoded data.

Write encoded or decoded data to a destination file

The following code checks the status that `compression_stream_process` returns. If the status is either `COMPRESSION_STATUS_OK` or `COMPRESSION_STATUS_END`, the code writes the destination data to the destination file handler:

```
switch status {
    case COMPRESSION_STATUS_OK, COMPRESSION_STATUS_END:

        // Get the number of bytes put in the destination buffer.
```

```
        // This is the difference between `stream.dst_size` before the
        // call (`bufferSize`), and `stream.dst_size` after the call.
        let count = bufferSize - streamPointer.pointee.dst_size

        let outputData = Data(bytesNoCopy: destinationBufferPointer,
                              count: count,
                              deallocator: .none)

        // Write all produced bytes to the output file.
        destinationFileHandle.write(outputData)

        // Reset the stream to receive the next batch of output.
        streamPointer.pointee.dst_ptr = destinationBufferPointer
        streamPointer.pointee.dst_size = bufferSize
```

This read-encode/decode-write loop continues while `status` equals `COMPRESSION_STATUS_OK`.

Close the source and destination files

After the app has finished working with the source and destination file handles, it calls the `closeFile()` method to close them.

```
sourceFileHandle.closeFile()
destinationFileHandle.closeFile()
```

See Also

Compression

- Compressing and decompressing data with buffer compression

  Compress a string, write it to the file system, and decompress the same file using buffer compression.

- Compressing and decompressing data with input and output filters

  Compress and decompress streamed or from-memory data, using input and output filters.