Clzip

LZMA lossless data compressor
for Clzip version 1.12, 4 January 2021

by Antonio Diaz Diaz

Table of Contents

1 Introduction

Clzip is a C language version of lzip, fully compatible with lzip 1.4 or newer. As clzip is written in C, it may be easier to integrate in applications like package managers, embedded devices, or systems lacking a C++ compiler.

Lzip is a lossless data compressor with a user interface similar to the one of gzip or bzip2. Lzip uses a simplified form of the 'Lempel-Ziv-Markov chain-Algorithm' (LZMA) stream format, chosen to maximize safety and interoperability. Lzip can compress about as fast as gzip (lzip -0) or compress most files more than bzip2 (lzip -9). Decompression speed is intermediate between gzip and bzip2. Lzip is better than gzip and bzip2 from a data recovery perspective. Lzip has been designed, written, and tested with great care to replace gzip and bzip2 as the standard general-purpose compressed format for unix-like systems.

For compressing/decompressing large files on multiprocessor machines plzip can be much faster than lzip at the cost of a slightly reduced compression ratio. See `plzip`.

For creation and manipulation of compressed tar archives tarlz can be more efficient than using tar and plzip because tarlz is able to keep the alignment between tar members and lzip members. See `tarlz`.

The lzip file format is designed for data sharing and long-term archiving, taking into account both data integrity and decoder availability:

- The lzip format provides very safe integrity checking and some data recovery means. The program lziprecover can repair bit flip errors (one of the most common forms of data corruption) in lzip files, and provides data recovery capabilities, including error-checked merging of damaged copies of a file. See Section "Data safety" in `lziprecover`.
- The lzip format is as simple as possible (but not simpler). The lzip manual provides the source code of a simple decompressor along with a detailed explanation of how it works, so that with the only help of the lzip manual it would be possible for a digital archaeologist to extract the data from a lzip file long after quantum computers eventually render LZMA obsolete.
- Additionally the lzip reference implementation is copylefted, which guarantees that it will remain free forever.

A nice feature of the lzip format is that a corrupt byte is easier to repair the nearer it is from the beginning of the file. Therefore, with the help of lziprecover, losing an entire archive just because of a corrupt byte near the beginning is a thing of the past.

The member trailer stores the 32-bit CRC of the original data, the size of the original data, and the size of the member. These values, together with the end-of-stream marker, provide a 3 factor integrity checking which guarantees that the decompressed version of the data is identical to the original. This guards against corruption of the compressed data, and against undetected bugs in clzip (hopefully very unlikely). The chances of data corruption going undetected are microscopic. Be aware, though, that the check occurs upon decompression, so it can only tell you that something is wrong. It can't help you recover the original uncompressed data.

Clzip uses the same well-defined exit status values used by bzip2, which makes it safer than compressors returning ambiguous warning values (like gzip) when it is used as a back end for other programs like tar or zutils.

Clzip will automatically use for each file the largest dictionary size that does not exceed neither the file size nor the limit given. Keep in mind that the decompression memory requirement is affected at compression time by the choice of dictionary size limit.

The amount of memory required for compression is about 1 or 2 times the dictionary size limit (1 if input file size is less than dictionary size limit, else 2) plus 9 times the dictionary size really used. The option '-0' is special and only requires about 1.5 MiB at most. The amount of memory required for decompression is about 46 kB larger than the dictionary size really used.

When compressing, clzip replaces every file given in the command line with a compressed version of itself, with the name "original_name.lz". When decompressing, clzip attempts to guess the name for the decompressed file from that of the compressed file as follows:

filename.lz	becomes	filename
filename.tlz	becomes	filename.tar
anyothername	becomes	anyothername.out

The DJGPP port of clzip checks if the used file system supports LFNs or only SFNs. If only SFN support is available, the file name and its extension must be truncated and modified to encode the compressor extension and the numbers to identify a multi volume archive. The program does not store the original file name in the header of the archive thus there is no chance to regenerate them when uncompressiong the archive in a SFN environment. Here the same naming schema is applied than the one used in the bzip2 and xz ports. The archiver extension is ".lz". On SFN systems, the ".lz" extension will consume one or two characters of the original file name extension.

For SFN systems the following rules apply when decompressing:

filename.exl	becomes	filename.ex
filename.elz	becomes	filename.e
filename.lz	becomes	filename
filename.tlz	becomes	filename.tar
anyothername.ext	becomes	anyothername.out

For SFN systems the following rules apply when compressing:

filename.ext	becomes	filename.exl
filename.ex	becomes	filename.exl
filename.e	becomes	filename.elz
filename	becomes	filename.lz
filename.tar	becomes	filename.tlz

If only SFN support is available, the DJGPP port of clzip will always consume the last 5 characters of the 8 possible name characters to encode the volume number. E.g.: when compressing a file with the file name '*filename.ext*' and splitting it into multiple volumes, the different volumes will become '*fil*00001.exl', '*fil*00002.exl', etc.

(De)compressing a file is much like copying or moving it; therefore clzip preserves the access and modification dates, permissions, and, when possible, ownership of the file just as 'cp -p' does. (If the user ID or the group ID can't be duplicated, the file permission bits S_ISUID and S_ISGID are cleared).

Clzip is able to read from some types of non-regular files if either the option '-c' or the option '-o' is specified.

Clzip will refuse to read compressed data from a terminal or write compressed data to a terminal, as this would be entirely incomprehensible and might leave the terminal in an abnormal state.

Clzip will correctly decompress a file which is the concatenation of two or more compressed files. The result is the concatenation of the corresponding decompressed files. Integrity testing of concatenated compressed files is also supported.

Clzip can produce multimember files, and lziprecover can safely recover the undamaged members in case of file damage. Clzip can also split the compressed output in volumes of a given size, even when reading from standard input. This allows the direct creation of multivolume compressed tar archives.

Clzip is able to compress and decompress streams of unlimited size by automatically creating multimember output. The members so created are large, about 2 PiB each.

2 Meaning of clzip's output

The output of clzip looks like this:

```
clzip -v foo
  foo:  6.676:1, 14.98% ratio, 85.02% saved, 450560 in, 67493 out.

clzip -tvvv foo.lz
  foo.lz:  6.676:1, 14.98% ratio, 85.02% saved.  450560 out,  67493 in. ok
```

The meaning of each field is as follows:

`N:1` — The compression ratio (uncompressed_size / compressed_size), shown as N to 1.

`ratio` — The inverse compression ratio (compressed_size / uncompressed_size), shown as a percentage. A decimal ratio is easily obtained by moving the decimal point two places to the left; 14.98% = 0.1498.

`saved` — The space saved by compression (1 - ratio), shown as a percentage.

`in` — Size of the input data. This is the uncompressed size when compressing, or the compressed size when decompressing or testing. Note that clzip always prints the uncompressed size before the compressed size when compressing, decompressing, testing, or listing.

`out` — Size of the output data. This is the compressed size when compressing, or the decompressed size when decompressing or testing.

When decompressing or testing at verbosity level 4 (-vvvv), the dictionary size used to compress the file and the CRC32 of the uncompressed data are also shown.

LANGUAGE NOTE: Uncompressed = not compressed = plain data; it may never have been compressed. Decompressed is used to refer to data which have undergone the process of decompression.

3 Invoking clzip

The format for running clzip is:

```
clzip [options] [files]
```

If no file names are specified, clzip compresses (or decompresses) from standard input to standard output. A hyphen '`-`' used as a *file* argument means standard input. It can be mixed with other *files* and is read just once, the first time it appears in the command line.

clzip supports the following options: See Section "Argument syntax" in `arg_parser`.

`-h`
`--help` Print an informative help message describing the options and exit.

`-V`
`--version`
Print the version number of clzip on the standard output and exit. This version number should be included in all bug reports.

`-a`
`--trailing-error`
Exit with error status 2 if any remaining input is detected after decompressing the last member. Such remaining input is usually trailing garbage that can be safely ignored. See [concat-example], page 21.

`-b` *`bytes`*
`--member-size=`*`bytes`*
When compressing, set the member size limit to *bytes*. It is advisable to keep members smaller than RAM size so that they can be repaired with lziprecover in case of corruption. A small member size may degrade compression ratio, so use it only when needed. Valid values range from 100 kB to 2 PiB. Defaults to 2 PiB.

`-c`
`--stdout` Compress or decompress to standard output; keep input files unchanged. If compressing several files, each file is compressed independently. (The output consists of a sequence of independently compressed members). This option (or '`-o`') is needed when reading from a named pipe (fifo) or from a device. Use it also to recover as much of the decompressed data as possible when decompressing a corrupt file. '`-c`' overrides '`-o`' and '`-S`'. '`-c`' has no effect when testing or listing.

`-d`
`--decompress`
Decompress the files specified. If a file does not exist or can't be opened, clzip continues decompressing the rest of the files. If a file fails to decompress, or is a terminal, clzip exits immediately without decompressing the rest of the files.

`-f`
`--force` Force overwrite of output files.

`-F`
`--recompress`
: When compressing, force re-compression of files whose name already has the '`.lz`' or '`.tlz`' suffix.

`-k`
`--keep`
: Keep (don't delete) input files during compression or decompression.

`-l`
`--list`
: Print the uncompressed size, compressed size, and percentage saved of the files specified. Trailing data are ignored. The values produced are correct even for multimember files. If more than one file is given, a final line containing the cumulative sizes is printed. With '`-v`', the dictionary size, the number of members in the file, and the amount of trailing data (if any) are also printed. With '`-vv`', the positions and sizes of each member in multimember files are also printed.

  '`-lq`' can be used to verify quickly (without decompressing) the structural integrity of the files specified. (Use '`--test`' to verify the data integrity). '`-alq`' additionally verifies that none of the files specified contain trailing data.

`-m` *`bytes`*
`--match-length=`*`bytes`*
: When compressing, set the match length limit in bytes. After a match this long is found, the search is finished. Valid values range from 5 to 273. Larger values usually give better compression ratios but longer compression times.

`-o` *`file`*
`--output=`*`file`*
: If '`-c`' has not been also specified, write the (de)compressed output to *file*; keep input files unchanged. If compressing several files, each file is compressed independently. (The output consists of a sequence of independently compressed members). This option (or '`-c`') is needed when reading from a named pipe (fifo) or from a device. '`-o -`' is equivalent to '`-c`'. '`-o`' has no effect when testing or listing.

  In order to keep backward compatibility with clzip versions prior to 1.12, when compressing from standard input and no other file names are given, the extension '`.lz`' is appended to *file* unless it already ends in '`.lz`' or '`.tlz`'. This feature will be removed in a future version of clzip. Meanwhile, redirection may be used instead of '`-o`' to write the compressed output to a file without the extension '`.lz`' in its name: '`clzip < file > foo`'.

  When compressing and splitting the output in volumes, *file* is used as a prefix, and several files named '*`file`*`00001.lz`', '*`file`*`00002.lz`', etc, are created. In this case, only one input file is allowed.

`-q`
`--quiet`
: Quiet operation. Suppress all messages.

`-s` *bytes*
`--dictionary-size=`*bytes*

When compressing, set the dictionary size limit in bytes. Clzip will use for each file the largest dictionary size that does not exceed neither the file size nor this limit. Valid values range from 4 KiB to 512 MiB. Values 12 to 29 are interpreted as powers of two, meaning 2^12 to 2^29 bytes. Dictionary sizes are quantized so that they can be coded in just one byte (see [coded-dict-size], page 12). If the size specified does not match one of the valid sizes, it will be rounded upwards by adding up to (*bytes* / 8) to it.

For maximum compression you should use a dictionary size limit as large as possible, but keep in mind that the decompression memory requirement is affected at compression time by the choice of dictionary size limit.

`-S` *bytes*
`--volume-size=`*bytes*

When compressing, and '`-c`' has not been also specified, split the compressed output into several volume files with names '`original_name00001.lz`', '`original_name00002.lz`', etc, and set the volume size limit to *bytes*. Input files are kept unchanged. Each volume is a complete, maybe multimember, lzip file. A small volume size may degrade compression ratio, so use it only when needed. Valid values range from 100 kB to 4 EiB.

`-t`
`--test`

Check integrity of the files specified, but don't decompress them. This really performs a trial decompression and throws away the result. Use it together with '`-v`' to see information about the files. If a file fails the test, does not exist, can't be opened, or is a terminal, clzip continues checking the rest of the files. A final diagnostic is shown at verbosity level 1 or higher if any file fails the test when testing multiple files.

`-v`
`--verbose`

Verbose mode.
When compressing, show the compression ratio and size for each file processed.
When decompressing or testing, further -v's (up to 4) increase the verbosity level, showing status, compression ratio, dictionary size, trailer contents (CRC, data size, member size), and up to 6 bytes of trailing data (if any) both in hexadecimal and as a string of printable ASCII characters.
Two or more '`-v`' options show the progress of (de)compression.

`-0 .. -9`

Compression level. Set the compression parameters (dictionary size and match length limit) as shown in the table below. The default compression level is '`-6`', equivalent to '`-s8MiB -m36`'. Note that '`-9`' can be much slower than '`-0`'. These options have no effect when decompressing, testing, or listing.

The bidimensional parameter space of LZMA can't be mapped to a linear scale optimal for all files. If your files are large, very repetitive, etc, you may need to use the options '`--dictionary-size`' and '`--match-length`' directly to achieve optimal performance.

If several compression levels or '-s' or '-m' options are given, the last setting is used. For example '-9 -s64MiB' is equivalent to '-s64MiB -m273'

Level	Dictionary size (-s)	Match length limit (-m)
-0	64 KiB	16 bytes
-1	1 MiB	5 bytes
-2	1.5 MiB	6 bytes
-3	2 MiB	8 bytes
-4	3 MiB	12 bytes
-5	4 MiB	20 bytes
-6	8 MiB	36 bytes
-7	16 MiB	68 bytes
-8	24 MiB	132 bytes
-9	32 MiB	273 bytes

`--fast`
`--best` Aliases for GNU gzip compatibility.

`--loose-trailing`
When decompressing, testing, or listing, allow trailing data whose first bytes are so similar to the magic bytes of a lzip header that they can be confused with a corrupt header. Use this option if a file triggers a "corrupt header" error and the cause is not indeed a corrupt header.

Numbers given as arguments to options may be followed by a multiplier and an optional 'B' for "byte".

Table of SI and binary prefixes (unit multipliers):

Prefix	Value	|	Prefix	Value
k	kilobyte (10^3 = 1000)	|	Ki	kibibyte (2^10 = 1024)
M	megabyte (10^6)	|	Mi	mebibyte (2^20)
G	gigabyte (10^9)	|	Gi	gibibyte (2^30)
T	terabyte (10^12)	|	Ti	tebibyte (2^40)
P	petabyte (10^15)	|	Pi	pebibyte (2^50)
E	exabyte (10^18)	|	Ei	exbibyte (2^60)
Z	zettabyte (10^21)	|	Zi	zebibyte (2^70)
Y	yottabyte (10^24)	|	Yi	yobibyte (2^80)

Exit status: 0 for a normal exit, 1 for environmental problems (file not found, invalid flags, I/O errors, etc), 2 to indicate a corrupt or invalid input file, 3 for an internal consistency error (eg, bug) which caused clzip to panic.

4 Design, development, and testing of lzip

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.
— C.A.R. Hoare

Lzip is developed by volunteers who lack the resources required for extensive testing in all circumstances. It is up to you to test lzip before using it in mission-critical applications. However, a compressor like lzip is not a toy, and maintaining it is not a hobby. Many people's data depend on it. Therefore the lzip file format has been reviewed carefully and is believed to be free from negligent design errors.

Lzip has been designed, written, and tested with great care to replace gzip and bzip2 as the standard general-purpose compressed format for unix-like systems. This chapter describes the lessons learned from these previous formats, and their application to the design of lzip.

4.1 Format design

When gzip was designed in 1992, computers and operating systems were much less capable than they are today. The designers of gzip tried to work around some of those limitations, like 8.3 file names, with additional fields in the file format.

Today those limitations have mostly disappeared, and the format of gzip has proved to be unnecessarily complicated. It includes fields that were never used, others that have lost their usefulness, and finally others that have become too limited.

Bzip2 was designed 5 years later, and its format is simpler than the one of gzip.

Probably the worst defect of the gzip format from the point of view of data safety is the variable size of its header. If the byte at offset 3 (flags) of a gzip member gets corrupted, it may become difficult to recover the data, even if the compressed blocks are intact, because it can't be known with certainty where the compressed blocks begin.

By contrast, the header of a lzip member has a fixed length of 6. The LZMA stream in a lzip member always starts at offset 6, making it trivial to recover the data even if the whole header becomes corrupt.

Bzip2 also provides a header of fixed length and marks the begin and end of each compressed block with six magic bytes, making it possible to find the compressed blocks even in case of file damage. But bzip2 does not store the size of each compressed block, as lzip does.

Lziprecover is able to provide unique data recovery capabilities because the lzip format is extraordinarily safe. The simple and safe design of the file format complements the embedded error detection provided by the LZMA data stream. Any distance larger than the dictionary size acts as a forbidden symbol, allowing the decompressor to detect the approximate position of errors, and leaving very little work for the check sequence (CRC and data sizes) in the detection of errors. Lzip is usually able to detect all possible bit flips in the compressed data without resorting to the check sequence. It would be difficult to write an automatic recovery tool like lziprecover for the gzip format. And, as far as I know, it has never been written.

Lzip, like gzip and bzip2, uses a CRC32 to check the integrity of the decompressed data because it provides optimal accuracy in the detection of errors up to a compressed size of about 16 GiB, a size larger than that of most files. In the case of lzip, the additional detection capability of the decompressor reduces the probability of undetected errors several million times more, resulting in a combined integrity checking optimally accurate for any member size produced by lzip. Preliminary results suggest that the lzip format is safe enough to be used in critical safety avionics systems.

The lzip format is designed for long-term archiving. Therefore it excludes any unneeded features that may interfere with the future extraction of the decompressed data.

4.1.1 Gzip format (mis)features not present in lzip

'`Multiple algorithms`'
: Gzip provides a CM (Compression Method) field that has never been used because it is a bad idea to begin with. New compression methods may require additional fields, making it impossible to implement new methods and, at the same time, keep the same format. This field does not solve the problem of format proliferation; it just makes the problem less obvious.

'`Optional fields in header`'
: Unless special precautions are taken, optional fields are generally a bad idea because they produce a header of variable size. The gzip header has 2 fields that, in addition to being optional, are zero-terminated. This means that if any byte inside the field gets zeroed, or if the terminating zero gets altered, gzip won't be able to find neither the header CRC nor the compressed blocks.

'`Optional CRC for the header`'
: Using an optional CRC for the header is not only a bad idea, it is an error; it circumvents the Hamming distance (HD) of the CRC and may prevent the extraction of perfectly good data. For example, if the CRC is used and the bit enabling it is reset by a bit flip, the header will appear to be intact (in spite of being corrupt) while the compressed blocks will appear to be totally unrecoverable (in spite of being intact). Very misleading indeed.

'`Metadata`'
: The gzip format stores some metadata, like the modification time of the original file or the operating system on which compression took place. This complicates reproducible compression (obtaining identical compressed output from identical input).

4.1.2 Lzip format improvements over gzip and bzip2

'`64-bit size field`'
: Probably the most frequently reported shortcoming of the gzip format is that it only stores the least significant 32 bits of the uncompressed size. The size of any file larger than 4 GiB gets truncated.

  Bzip2 does not store the uncompressed size of the file.

The lzip format provides a 64-bit field for the uncompressed size. Additionally, lzip produces multimember output automatically when the size is too large for a single member, allowing for an unlimited uncompressed size.

'`Distributed index`'
: The lzip format provides a distributed index that, among other things, helps plzip to decompress several times faster than pigz and helps lziprecover do its job. Neither the gzip format nor the bzip2 format do provide an index.

  A distributed index is safer and more scalable than a monolithic index. The monolithic index introduces a single point of failure in the compressed file and may limit the number of members or the total uncompressed size.

4.2 Quality of implementation

'`Accurate and robust error detection`'
: The lzip format provides 3 factor integrity checking and the decompressors report mismatches in each factor separately. This way if just one byte in one factor fails but the other two factors match the data, it probably means that the data are intact and the corruption just affects the mismatching factor (CRC or data size) in the check sequence.

'`Multiple implementations`'
: Just like the lzip format provides 3 factor protection against undetected data corruption, the development methodology of the lzip family of compressors provides 3 factor protection against undetected programming errors.

  Three related but independent compressor implementations, lzip, clzip, and minilzip/lzlib, are developed concurrently. Every stable release of any of them is tested to verify that it produces identical output to the other two. This guarantees that all three implement the same algorithm, and makes it unlikely that any of them may contain serious undiscovered errors. In fact, no errors have been discovered in lzip since 2009.

  Additionally, the three implementations have been extensively tested with unzcrash, valgrind, and '`american fuzzy lop`' without finding a single vulnerability or false negative. See Section "Unzcrash" in `lziprecover`.

'`Dictionary size`'
: Lzip automatically adapts the dictionary size to the size of each file. In addition to reducing the amount of memory required for decompression, this feature also minimizes the probability of being affected by RAM errors during compression.

'`Exit status`'
: Returning a warning status of 2 is a design flaw of compress that leaked into the design of gzip. Both bzip2 and lzip are free from this flaw.

5 File format

Perfection is reached, not when there is no longer anything to add, but when there is no longer anything to take away.
— Antoine de Saint-Exupery

In the diagram below, a box like this:

```
+---+
|   | <-- the vertical bars might be missing
+---+
```

represents one byte; a box like this:

```
+==============+
|              |
+==============+
```

represents a variable number of bytes.

A lzip file consists of a series of "members" (compressed data sets). The members simply appear one after another in the file, with no additional information before, between, or after them.

Each member has the following structure:

```
+--+--+--+--+----+----+=============+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| ID string | VN | DS | LZMA stream | CRC32 |   Data size   |  Member size  |
+--+--+--+--+----+----+=============+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

All multibyte values are stored in little endian order.

'ID string (the "magic" bytes)'
: A four byte string, identifying the lzip format, with the value "LZIP" (0x4C, 0x5A, 0x49, 0x50).

'VN (version number, 1 byte)'
: Just in case something needs to be modified in the future. 1 for now.

'DS (coded dictionary size, 1 byte)'
: The dictionary size is calculated by taking a power of 2 (the base size) and subtracting from it a fraction between 0/16 and 7/16 of the base size.
Bits 4-0 contain the base 2 logarithm of the base size (12 to 29).
Bits 7-5 contain the numerator of the fraction (0 to 7) to subtract from the base size to obtain the dictionary size.
Example: 0xD3 = 2^19 - 6 * 2^15 = 512 KiB - 6 * 32 KiB = 320 KiB
Valid values for dictionary size range from 4 KiB to 512 MiB.

'LZMA stream'
: The LZMA stream, finished by an end of stream marker. Uses default values for encoder properties. See Chapter 7 [Stream format], page 16, for a complete description.

'CRC32 (4 bytes)'
: Cyclic Redundancy Check (CRC) of the uncompressed original data.

‘`Data size (8 bytes)`’
: Size of the uncompressed original data.

‘`Member size (8 bytes)`’
: Total size of the member, including header and trailer. This field acts as a distributed index, allows the verification of stream integrity, and facilitates safe recovery of undamaged members from multimember files.

6 Algorithm

In spite of its name (Lempel-Ziv-Markov chain-Algorithm), LZMA is not a concrete algorithm; it is more like "any algorithm using the LZMA coding scheme". LZMA compression consists in describing the uncompressed data as a succession of coding sequences from the set shown in Section '`What is coded`' (see [what-is-coded], page 16), and then encoding them using a range encoder. For example, the option '`-0`' of clzip uses the scheme in almost the simplest way possible; issuing the longest match it can find, or a literal byte if it can't find a match. Inversely, a much more elaborated way of finding coding sequences of minimum size than the one currently used by clzip could be developed, and the resulting sequence could also be coded using the LZMA coding scheme.

Clzip currently implements two variants of the LZMA algorithm; fast (used by option '`-0`') and normal (used by all other compression levels).

The high compression of LZMA comes from combining two basic, well-proven compression ideas: sliding dictionaries (LZ77/78) and markov models (the thing used by every compression algorithm that uses a range encoder or similar order-0 entropy coder as its last stage) with segregation of contexts according to what the bits are used for.

Clzip is a two stage compressor. The first stage is a Lempel-Ziv coder, which reduces redundancy by translating chunks of data to their corresponding distance-length pairs. The second stage is a range encoder that uses a different probability model for each type of data; distances, lengths, literal bytes, etc.

Here is how it works, step by step:

1) The member header is written to the output stream.

2) The first byte is coded literally, because there are no previous bytes to which the match finder can refer to.

3) The main encoder advances to the next byte in the input data and calls the match finder.

4) The match finder fills an array with the minimum distances before the current byte where a match of a given length can be found.

5) Go back to step 3 until a sequence (formed of pairs, repeated distances, and literal bytes) of minimum price has been formed. Where the price represents the number of output bits produced.

6) The range encoder encodes the sequence produced by the main encoder and sends the bytes produced to the output stream.

7) Go back to step 3 until the input data are finished or until the member or volume size limits are reached.

8) The range encoder is flushed.

9) The member trailer is written to the output stream.

10) If there are more data to compress, go back to step 1.

During compression, clzip reads data in large blocks (one dictionary size at a time). Therefore it may block for up to tens of seconds any process feeding data to it through a pipe. This is normal. The blocking intervals get longer with higher compression levels because dictionary size increases (and compression speed decreases) with compression level.

The ideas embodied in clzip are due to (at least) the following people: Abraham Lempel and Jacob Ziv (for the LZ algorithm), Andrey Markov (for the definition of Markov chains), G.N.N. Martin (for the definition of range encoding), Igor Pavlov (for putting all the above together in LZMA), and Julian Seward (for bzip2's CLI).

7 Format of the LZMA stream in lzip files

Lzip uses a simplified form of the LZMA stream format chosen to maximize safety and interoperability.

The LZMA algorithm has three parameters, called "special LZMA properties", to adjust it for some kinds of binary data. These parameters are; '`literal_context_bits`' (with a default value of 3), '`literal_pos_state_bits`' (with a default value of 0), and '`pos_state_bits`' (with a default value of 2). As a general purpose compressor, lzip only uses the default values for these parameters. In particular '`literal_pos_state_bits`' has been optimized away and does not even appear in the code.

Lzip finishes the LZMA stream with an "End Of Stream" (EOS) marker (the distance-length pair 0xFFFFFFFFU, 2), which in conjunction with the '`member size`' field in the member trailer allows the verification of stream integrity. The LZMA stream in lzip files always has these two features (default properties and EOS marker) and is referred to in this document as LZMA-302eos. The EOS marker is the only marker allowed in lzip files.

The second stage of LZMA is a range encoder that uses a different probability model for each type of symbol; distances, lengths, literal bytes, etc. Range encoding conceptually encodes all the symbols of the message into one number. Unlike Huffman coding, which assigns to each symbol a bit-pattern and concatenates all the bit-patterns together, range encoding can compress one symbol to less than one bit. Therefore the compressed data produced by a range encoder can't be split in pieces that could be described individually.

It seems that the only way of describing the LZMA-302eos stream is describing the algorithm that decodes it. And given the many details about the range decoder that need to be described accurately, the source code of a real decoder seems the only appropriate reference to use.

What follows is a description of the decoding algorithm for LZMA-302eos streams using as reference the source code of "lzd", an educational decompressor for lzip files which can be downloaded from the lzip download directory. The source code of lzd is included in appendix A. See Appendix A [Reference source code], page 24.

7.1 What is coded

The LZMA stream includes literals, matches, and repeated matches (matches reusing a recently used distance). There are 7 different coding sequences:

Bit sequence	**Name**	**Description**
0 + byte	literal	literal byte
1 + 0 + len + dis	match	distance-length pair
1 + 1 + 0 + 0	shortrep	1 byte match at latest used distance
1 + 1 + 0 + 1 + len	rep0	len bytes match at latest used distance
1 + 1 + 1 + 0 + len	rep1	len bytes match at second latest used distance
1 + 1 + 1 + 1 + 0 + len	rep2	len bytes match at third latest used distance
1 + 1 + 1 + 1 + 1 + len	rep3	len bytes match at fourth latest used distance

In the following tables, multibit sequences are coded in normal order, from most significant bit (MSB) to least significant bit (LSB), except where noted otherwise.

Lengths (the '`len`' in the table above) are coded as follows:

Bit sequence	Description
0 + 3 bits	lengths from 2 to 9
1 + 0 + 3 bits	lengths from 10 to 17
1 + 1 + 8 bits	lengths from 18 to 273

The coding of distances is a little more complicated, so I'll begin explaining a simpler version of the encoding.

Imagine you need to encode a number from 0 to 2^32 - 1, and you want to do it in a way that produces shorter codes for the smaller numbers. You may first encode the position of the most significant bit that is set to 1, which you may find by making a bit scan from the left (from the MSB). A position of 0 means that the number is 0 (no bit is set), 1 means the LSB is the first bit set (the number is 1), and 32 means the MSB is set (i.e., the number is >= 0x80000000). Then, if the position is >= 2, you encode the remaining position - 1 bits. Let's call these bits "direct_bits" because they are coded directly by value instead of indirectly by position.

The inconvenient of this simple method is that it needs 6 bits to encode the position, but it just uses 33 of the 64 possible values, wasting almost half of the codes.

The intelligent trick of LZMA is that it encodes in what it calls a "slot" the position of the most significant bit set, along with the value of the next bit, using the same 6 bits that would take to encode the position alone. This seems to need 66 slots (twice the number of positions), but for positions 0 and 1 there is no next bit, so the number of slots needed is 64 (0 to 63).

The 6 bits representing this "slot number" are then context-coded. If the distance is >= 4, the remaining bits are encoded as follows. '`direct_bits`' is the amount of remaining bits (from 1 to 30) needed to form a complete distance, and is calculated as (slot >> 1) - 1. If a distance needs 6 or more direct_bits, the last 4 bits are encoded separately. The last piece (all the direct_bits for distances 4 to 127, or the last 4 bits for distances >= 128) is context-coded in reverse order (from LSB to MSB). For distances >= 128, the '`direct_bits - 4`' part is encoded with fixed 0.5 probability.

Bit sequence	Description
slot	distances from 0 to 3
slot + direct_bits	distances from 4 to 127
slot + (direct_bits - 4) + 4 bits	distances from 128 to 2^32 - 1

7.2 The coding contexts

These contexts ('`Bit_model`' in the source), are integers or arrays of integers representing the probability of the corresponding bit being 0.

The indices used in these arrays are:

'`state`' A state machine ('`State`' in the source) with 12 states (0 to 11), coding the latest 2 to 4 types of sequences processed. The initial state is 0.

'`pos_state`'
Value of the 2 least significant bits of the current position in the decoded data.

'`literal_state`'
Value of the 3 most significant bits of the latest byte decoded.

'`len_state`'
Coded value of the current match length (length - 2), with a maximum of 3. The resulting value is in the range 0 to 3.

In the following table, '`!literal`' is any sequence except a literal byte. '`rep`' is any one of '`rep0`', '`rep1`', '`rep2`', or '`rep3`'. The types of previous sequences corresponding to each state are:

State	Types of previous sequences
0	literal, literal, literal
1	match, literal, literal
2	rep or (!literal, shortrep), literal, literal
3	literal, shortrep, literal, literal
4	match, literal
5	rep or (!literal, shortrep), literal
6	literal, shortrep, literal
7	literal, match
8	literal, rep
9	literal, shortrep
10	!literal, match
11	!literal, (rep or shortrep)

The contexts for decoding the type of coding sequence are:

Name	Indices	Used when
bm_match	state, pos_state	sequence start
bm_rep	state	after sequence 1
bm_rep0	state	after sequence 11
bm_rep1	state	after sequence 111
bm_rep2	state	after sequence 1111
bm_len	state, pos_state	after sequence 110

The contexts for decoding distances are:

Name	Indices	Used when
bm_dis_slot	len_state, bit tree	distance start
bm_dis	reverse bit tree	after slots 4 to 13
bm_align	reverse bit tree	for distances >= 128, after fixed probability bits

There are two separate sets of contexts for lengths (‘Len_model’ in the source). One for normal matches, the other for repeated matches. The contexts in each Len_model are (see ‘decode_len’ in the source):

Name	**Indices**	**Used when**
choice1	none	length start
choice2	none	after sequence 1
bm_low	pos_state, bit tree	after sequence 0
bm_mid	pos_state, bit tree	after sequence 10
bm_high	bit tree	after sequence 11

The context array ‘bm_literal’ is special. In principle it acts as a normal bit tree context, the one selected by ‘literal_state’. But if the previous decoded byte was not a literal, two other bit tree contexts are used depending on the value of each bit in ‘match_byte’ (the byte at the latest used distance), until a bit is decoded that is different from its corresponding bit in ‘match_byte’. After the first difference is found, the rest of the byte is decoded using the normal bit tree context. (See ‘decode_matched’ in the source).

7.3 The range decoder

The LZMA stream is consumed one byte at a time by the range decoder. (See ‘normalize’ in the source). Every byte consumed produces a variable number of decoded bits, depending on how well these bits agree with their context. (See ‘decode_bit’ in the source).

The range decoder state consists of two unsigned 32-bit variables; ‘range’ (representing the most significant part of the range size not yet decoded), and ‘code’ (representing the current point within ‘range’). ‘range’ is initialized to 2^32 - 1, and ‘code’ is initialized to 0.

The range encoder produces a first 0 byte that must be ignored by the range decoder. This is done by shifting 5 bytes in the initialization of ‘code’ instead of 4. (See the ‘Range_decoder’ constructor in the source).

7.4 Decoding and verifying the LZMA stream

After decoding the member header and obtaining the dictionary size, the range decoder is initialized and then the LZMA decoder enters a loop (See ‘decode_member’ in the source) where it invokes the range decoder with the appropriate contexts to decode the different coding sequences (matches, repeated matches, and literal bytes), until the "End Of Stream" marker is decoded.

Once the "End Of Stream" marker has been decoded, the decompressor reads and decodes the member trailer, and verifies that the three integrity factors (CRC, data size, and member size) match those calculated by the LZMA decoder.

8 Extra data appended to the file

Sometimes extra data are found appended to a lzip file after the last member. Such trailing data may be:

- Padding added to make the file size a multiple of some block size, for example when writing to a tape. It is safe to append any amount of padding zero bytes to a lzip file.
- Useful data added by the user; a cryptographically secure hash, a description of file contents, etc. It is safe to append any amount of text to a lzip file as long as none of the first four bytes of the text match the corresponding byte in the string `"LZIP"`, and the text does not contain any zero bytes (null characters). Nonzero bytes and zero bytes can't be safely mixed in trailing data.
- Garbage added by some not totally successful copy operation.
- Malicious data added to the file in order to make its total size and hash value (for a chosen hash) coincide with those of another file.
- In rare cases, trailing data could be the corrupt header of another member. In multimember or concatenated files the probability of corruption happening in the magic bytes is 5 times smaller than the probability of getting a false positive caused by the corruption of the integrity information itself. Therefore it can be considered to be below the noise level. Additionally, the test used by clzip to discriminate trailing data from a corrupt header has a Hamming distance (HD) of 3, and the 3 bit flips must happen in different magic bytes for the test to fail. In any case, the option '`--trailing-error`' guarantees that any corrupt header will be detected.

Trailing data are in no way part of the lzip file format, but tools reading lzip files are expected to behave as correctly and usefully as possible in the presence of trailing data.

Trailing data can be safely ignored in most cases. In some cases, like that of user-added data, they are expected to be ignored. In those cases where a file containing trailing data must be rejected, the option '`--trailing-error`' can be used. See [–trailing-error], page 5.

9 A small tutorial with examples

WARNING! Even if clzip is bug-free, other causes may result in a corrupt compressed file (bugs in the system libraries, memory errors, etc). Therefore, if the data you are going to compress are important, give the option '`--keep`' to clzip and don't remove the original file until you verify the compressed file with a command like '`clzip -cd file.lz | cmp file -`'. Most RAM errors happening during compression can only be detected by comparing the compressed file with the original because the corruption happens before clzip compresses the RAM contents, resulting in a valid compressed file containing wrong data.

Example 1: Extract all the files from archive '`foo.tar.lz`'.

```
  tar -xf foo.tar.lz
or
  clzip -cd foo.tar.lz | tar -xf -
```

Example 2: Replace a regular file with its compressed version '`file.lz`' and show the compression ratio.

```
clzip -v file
```

Example 3: Like example 1 but the created '`file.lz`' is multimember with a member size of 1 MiB. The compression ratio is not shown.

```
clzip -b 1MiB file
```

Example 4: Restore a regular file from its compressed version '`file.lz`'. If the operation is successful, '`file.lz`' is removed.

```
clzip -d file.lz
```

Example 5: Verify the integrity of the compressed file '`file.lz`' and show status.

```
clzip -tv file.lz
```

Example 6: Compress a whole device in /dev/sdc and send the output to '`file.lz`'.

```
  clzip -c /dev/sdc > file.lz
or
  clzip /dev/sdc -o file.lz
```

Example 7: The right way of concatenating the decompressed output of two or more compressed files. See Chapter 8 [Trailing data], page 20.

```
Don't do this
  cat file1.lz file2.lz file3.lz | clzip -d -
Do this instead
  clzip -cd file1.lz file2.lz file3.lz
```

Example 8: Decompress '`file.lz`' partially until 10 KiB of decompressed data are produced.

```
clzip -cd file.lz | dd bs=1024 count=10
```

Example 9: Decompress 'file.lz' partially from decompressed byte at offset 10000 to decompressed byte at offset 14999 (5000 bytes are produced).

```
clzip -cd file.lz | dd bs=1000 skip=10 count=5
```

Example 10: Create a multivolume compressed tar archive with a volume size of 1440 KiB.

```
tar -c some_directory | clzip -S 1440KiB -o volume_name -
```

Example 11: Extract a multivolume compressed tar archive.

```
clzip -cd volume_name*.lz | tar -xf -
```

Example 12: Create a multivolume compressed backup of a large database file with a volume size of 650 MB, where each volume is a multimember file with a member size of 32 MiB.

```
clzip -b 32MiB -S 650MB big_db
```

10 Reporting bugs

There are probably bugs in clzip. There are certainly errors and omissions in this manual. If you report them, they will get fixed. If you don't, no one will ever know about them and they will remain unfixed for all eternity, if not longer.

If you find a bug in clzip, please send electronic mail to `lzip-bug@nongnu.org`. Include the version number, which you can find by running '`clzip --version`'.

Appendix A Reference source code

```
/* Lzd - Educational decompressor for the lzip format
   Copyright (C) 2013-2021 Antonio Diaz Diaz.

   This program is free software. Redistribution and use in source and
   binary forms, with or without modification, are permitted provided
   that the following conditions are met:

   1. Redistributions of source code must retain the above copyright
   notice, this list of conditions, and the following disclaimer.

   2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions, and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
*/
/*
   Exit status: 0 for a normal exit, 1 for environmental problems
   (file not found, invalid flags, I/O errors, etc), 2 to indicate a
   corrupt or invalid input file.
*/

#include <algorithm>
#include <cerrno>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <stdint.h>
#include <unistd.h>
#if defined(__MSVCRT__) || defined(__OS2__) || defined(__DJGPP__)
#include <fcntl.h>
#include <io.h>
#endif


class State
  {
  int st;

public:
  enum { states = 12 };
  State() : st( 0 ) {}
  int operator()() const { return st; }
```

```
  bool is_char() const { return st < 7; }

  void set_char()
    {
    const int next[states] = { 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 4, 5 };
    st = next[st];
    }
  void set_match()     { st = ( st < 7 ) ? 7 : 10; }
  void set_rep()       { st = ( st < 7 ) ? 8 : 11; }
  void set_short_rep() { st = ( st < 7 ) ? 9 : 11; }
  };


enum {
  min_dictionary_size = 1 << 12,
  max_dictionary_size = 1 << 29,
  literal_context_bits = 3,
  literal_pos_state_bits = 0,                          // not used
  pos_state_bits = 2,
  pos_states = 1 << pos_state_bits,
  pos_state_mask = pos_states - 1,

  len_states = 4,
  dis_slot_bits = 6,
  start_dis_model = 4,
  end_dis_model = 14,
  modeled_distances = 1 << ( end_dis_model / 2 ),      // 128
  dis_align_bits = 4,
  dis_align_size = 1 << dis_align_bits,

  len_low_bits = 3,
  len_mid_bits = 3,
  len_high_bits = 8,
  len_low_symbols = 1 << len_low_bits,
  len_mid_symbols = 1 << len_mid_bits,
  len_high_symbols = 1 << len_high_bits,
  max_len_symbols = len_low_symbols + len_mid_symbols + len_high_symbols,

  min_match_len = 2,                                   // must be 2

  bit_model_move_bits = 5,
  bit_model_total_bits = 11,
  bit_model_total = 1 << bit_model_total_bits };

struct Bit_model
  {
  int probability;
```

```
  Bit_model() : probability( bit_model_total / 2 ) {}
  };

struct Len_model
  {
  Bit_model choice1;
  Bit_model choice2;
  Bit_model bm_low[pos_states][len_low_symbols];
  Bit_model bm_mid[pos_states][len_mid_symbols];
  Bit_model bm_high[len_high_symbols];
  };


class CRC32
  {
  uint32_t data[256];           // Table of CRCs of all 8-bit messages.

public:
  CRC32()
    {
    for( unsigned n = 0; n < 256; ++n )
      {
      unsigned c = n;
      for( int k = 0; k < 8; ++k )
        { if( c & 1 ) c = 0xEDB88320U ^ ( c >> 1 ); else c >>= 1; }
      data[n] = c;
      }
    }

  void update_buf( uint32_t & crc, const uint8_t * const buffer,
                   const int size ) const
    {
    for( int i = 0; i < size; ++i )
      crc = data[(crc^buffer[i])&0xFF] ^ ( crc >> 8 );
    }
  };

const CRC32 crc32;


typedef uint8_t Lzip_header[6];         // 0-3 magic bytes
                                        //   4 version
                                        //   5 coded dictionary size
typedef uint8_t Lzip_trailer[20];
                        //  0-3  CRC32 of the uncompressed data
                        //  4-11 size of the uncompressed data
                        // 12-19 member size including header and trailer
```

```
class Range_decoder
  {
  unsigned long long member_pos;
  uint32_t code;
  uint32_t range;

public:
  Range_decoder() : member_pos( 6 ), code( 0 ), range( 0xFFFFFFFFU )
    {
    for( int i = 0; i < 5; ++i ) code = ( code << 8 ) | get_byte();
    }

  uint8_t get_byte() { ++member_pos; return std::getc( stdin ); }
  unsigned long long member_position() const { return member_pos; }

  unsigned decode( const int num_bits )
    {
    unsigned symbol = 0;
    for( int i = num_bits; i > 0; --i )
      {
      range >>= 1;
      symbol <<= 1;
      if( code >= range ) { code -= range; symbol |= 1; }
      if( range <= 0x00FFFFFFU )                        // normalize
        { range <<= 8; code = ( code << 8 ) | get_byte(); }
      }
    return symbol;
    }

  unsigned decode_bit( Bit_model & bm )
    {
    unsigned symbol;
    const uint32_t bound = ( range >> bit_model_total_bits ) * bm.probability;
    if( code < bound )
      {
      range = bound;
      bm.probability +=
        ( bit_model_total - bm.probability ) >> bit_model_move_bits;
      symbol = 0;
      }
    else
      {
      range -= bound;
      code -= bound;
      bm.probability -= bm.probability >> bit_model_move_bits;
      symbol = 1;
```

```
    }
  if( range <= 0x00FFFFFFU )                              // normalize
    { range <<= 8; code = ( code << 8 ) | get_byte(); }
  return symbol;
  }

unsigned decode_tree( Bit_model bm[], const int num_bits )
  {
  unsigned symbol = 1;
  for( int i = 0; i < num_bits; ++i )
    symbol = ( symbol << 1 ) | decode_bit( bm[symbol] );
  return symbol - ( 1 << num_bits );
  }

unsigned decode_tree_reversed( Bit_model bm[], const int num_bits )
  {
  unsigned symbol = decode_tree( bm, num_bits );
  unsigned reversed_symbol = 0;
  for( int i = 0; i < num_bits; ++i )
    {
    reversed_symbol = ( reversed_symbol << 1 ) | ( symbol & 1 );
    symbol >>= 1;
    }
  return reversed_symbol;
  }

unsigned decode_matched( Bit_model bm[], const unsigned match_byte )
  {
  unsigned symbol = 1;
  for( int i = 7; i >= 0; --i )
    {
    const unsigned match_bit = ( match_byte >> i ) & 1;
    const unsigned bit = decode_bit( bm[symbol+(match_bit<<8)+0x100] );
    symbol = ( symbol << 1 ) | bit;
    if( match_bit != bit )
      {
      while( symbol < 0x100 )
        symbol = ( symbol << 1 ) | decode_bit( bm[symbol] );
      break;
      }
    }
  return symbol & 0xFF;
  }

unsigned decode_len( Len_model & lm, const int pos_state )
  {
  if( decode_bit( lm.choice1 ) == 0 )
```

```
      return decode_tree( lm.bm_low[pos_state], len_low_bits );
    if( decode_bit( lm.choice2 ) == 0 )
      return len_low_symbols +
             decode_tree( lm.bm_mid[pos_state], len_mid_bits );
    return len_low_symbols + len_mid_symbols +
           decode_tree( lm.bm_high, len_high_bits );
    }
  };


class LZ_decoder
  {
  unsigned long long partial_data_pos;
  Range_decoder rdec;
  const unsigned dictionary_size;
  uint8_t * const buffer;       // output buffer
  unsigned pos;                 // current pos in buffer
  unsigned stream_pos;          // first byte not yet written to stdout
  uint32_t crc_;
  bool pos_wrapped;

  void flush_data();

  uint8_t peek( const unsigned distance ) const
    {
    if( pos > distance ) return buffer[pos - distance - 1];
    if( pos_wrapped ) return buffer[dictionary_size + pos - distance - 1];
    return 0;                   // prev_byte of first byte
    }

  void put_byte( const uint8_t b )
    {
    buffer[pos] = b;
    if( ++pos >= dictionary_size ) flush_data();
    }

public:
  explicit LZ_decoder( const unsigned dict_size )
    :
    partial_data_pos( 0 ),
    dictionary_size( dict_size ),
    buffer( new uint8_t[dictionary_size] ),
    pos( 0 ),
    stream_pos( 0 ),
    crc_( 0xFFFFFFFFU ),
    pos_wrapped( false )
    {}
```

```
  ~LZ_decoder() { delete[] buffer; }

  unsigned crc() const { return crc_ ^ 0xFFFFFFFFU; }
  unsigned long long data_position() const
    { return partial_data_pos + pos; }
  uint8_t get_byte() { return rdec.get_byte(); }
  unsigned long long member_position() const
    { return rdec.member_position(); }

  bool decode_member();
  };


void LZ_decoder::flush_data()
  {
  if( pos > stream_pos )
    {
    const unsigned size = pos - stream_pos;
    crc32.update_buf( crc_, buffer + stream_pos, size );
    if( std::fwrite( buffer + stream_pos, 1, size, stdout ) != size )
      { std::fprintf( stderr, "Write error: %s\n", std::strerror( errno ) );
        std::exit( 1 ); }
    if( pos >= dictionary_size )
      { partial_data_pos += pos; pos = 0; pos_wrapped = true; }
    stream_pos = pos;
    }
  }


bool LZ_decoder::decode_member()        // Returns false if error
  {
  Bit_model bm_literal[1<<literal_context_bits][0x300];
  Bit_model bm_match[State::states][pos_states];
  Bit_model bm_rep[State::states];
  Bit_model bm_rep0[State::states];
  Bit_model bm_rep1[State::states];
  Bit_model bm_rep2[State::states];
  Bit_model bm_len[State::states][pos_states];
  Bit_model bm_dis_slot[len_states][1<<dis_slot_bits];
  Bit_model bm_dis[modeled_distances-end_dis_model+1];
  Bit_model bm_align[dis_align_size];
  Len_model match_len_model;
  Len_model rep_len_model;
  unsigned rep0 = 0;            // rep[0-3] latest four distances
  unsigned rep1 = 0;            // used for efficient coding of
  unsigned rep2 = 0;            // repeated distances
```

```
  unsigned rep3 = 0;
  State state;

  while( !std::feof( stdin ) && !std::ferror( stdin ) )
    {
    const int pos_state = data_position() & pos_state_mask;
    if( rdec.decode_bit( bm_match[state()][pos_state] ) == 0 )  // 1st bit
      {
      // literal byte
      const uint8_t prev_byte = peek( 0 );
      const int literal_state = prev_byte >> ( 8 - literal_context_bits );
      Bit_model * const bm = bm_literal[literal_state];
      if( state.is_char() )
        put_byte( rdec.decode_tree( bm, 8 ) );
      else
        put_byte( rdec.decode_matched( bm, peek( rep0 ) ) );
      state.set_char();
      continue;
      }
    // match or repeated match
    int len;
    if( rdec.decode_bit( bm_rep[state()] ) != 0 )               // 2nd bit
      {
      if( rdec.decode_bit( bm_rep0[state()] ) == 0 )            // 3rd bit
        {
        if( rdec.decode_bit( bm_len[state()][pos_state] ) == 0 ) // 4th bit
          { state.set_short_rep(); put_byte( peek( rep0 ) ); continue; }
        }
      else
        {
        unsigned distance;
        if( rdec.decode_bit( bm_rep1[state()] ) == 0 )          // 4th bit
          distance = rep1;
        else
          {
          if( rdec.decode_bit( bm_rep2[state()] ) == 0 )        // 5th bit
            distance = rep2;
          else
            { distance = rep3; rep3 = rep2; }
          rep2 = rep1;
          }
        rep1 = rep0;
        rep0 = distance;
        }
      state.set_rep();
      len = min_match_len + rdec.decode_len( rep_len_model, pos_state );
      }
```

```
    else                                        // match
      {
      rep3 = rep2; rep2 = rep1; rep1 = rep0;
      len = min_match_len + rdec.decode_len( match_len_model, pos_state );
      const int len_state = std::min( len - min_match_len, len_states - 1 );
      rep0 = rdec.decode_tree( bm_dis_slot[len_state], dis_slot_bits );
      if( rep0 >= start_dis_model )
        {
        const unsigned dis_slot = rep0;
        const int direct_bits = ( dis_slot >> 1 ) - 1;
        rep0 = ( 2 | ( dis_slot & 1 ) ) << direct_bits;
        if( dis_slot < end_dis_model )
          rep0 += rdec.decode_tree_reversed( bm_dis + ( rep0 - dis_slot ),
                                             direct_bits );
        else
          {
          rep0 +=
            rdec.decode( direct_bits - dis_align_bits ) << dis_align_bits;
          rep0 += rdec.decode_tree_reversed( bm_align, dis_align_bits );
          if( rep0 == 0xFFFFFFFFU )             // marker found
            {
            flush_data();
            return ( len == min_match_len );    // End Of Stream marker
            }
          }
        }
      state.set_match();
      if( rep0 >= dictionary_size || ( rep0 >= pos && !pos_wrapped ) )
        { flush_data(); return false; }
      }
    for( int i = 0; i < len; ++i ) put_byte( peek( rep0 ) );
    }
  flush_data();
  return false;
  }


int main( const int argc, const char * const argv[] )
  {
  if( argc > 2 || ( argc == 2 && std::strcmp( argv[1], "-d" ) != 0 ) )
    {
    std::printf(
      "Lzd %s - Educational decompressor for the lzip format.\n"
      "Study the source to learn how a lzip decompressor works.\n"
      "See the lzip manual for an explanation of the code.\n"
      "\nUsage: %s [-d] < file.lz > file\n"
      "Lzd decompresses from standard input to standard output.\n"
```

```
      "\nCopyright (C) 2021 Antonio Diaz Diaz.\n"
      "License 2-clause BSD.\n"
      "This is free software: you are free to change and redistribute it.\n"
      "There is NO WARRANTY, to the extent permitted by law.\n"
      "Report bugs to lzip-bug@nongnu.org\n"
      "Lzd home page: http://www.nongnu.org/lzip/lzd.html\n",
      PROGVERSION, argv[0] );
    return 0;
    }

#if defined(__MSVCRT__) || defined(__OS2__) || defined(__DJGPP__)
  setmode( STDIN_FILENO, O_BINARY );
  setmode( STDOUT_FILENO, O_BINARY );
#endif

  for( bool first_member = true; ; first_member = false )
    {
    Lzip_header header;                          // verify header
    for( int i = 0; i < 6; ++i ) header[i] = std::getc( stdin );
    if( std::feof( stdin ) || std::memcmp( header, "LZIP\x01", 5 ) != 0 )
      {
      if( first_member )
        { std::fputs( "Bad magic number (file not in lzip format).\n",
                      stderr ); return 2; }
      break;                                     // ignore trailing data
      }
    unsigned dict_size = 1 << ( header[5] & 0x1F );
    dict_size -= ( dict_size / 16 ) * ( ( header[5] >> 5 ) & 7 );
    if( dict_size < min_dictionary_size || dict_size > max_dictionary_size )
      { std::fputs( "Invalid dictionary size in member header.\n", stderr );
        return 2; }

    LZ_decoder decoder( dict_size );             // decode LZMA stream
    if( !decoder.decode_member() )
      { std::fputs( "Data error\n", stderr ); return 2; }

    Lzip_trailer trailer;                        // verify trailer
    for( int i = 0; i < 20; ++i ) trailer[i] = decoder.get_byte();
    int retval = 0;
    unsigned crc = 0;
    for( int i = 3; i >= 0; --i ) crc = ( crc << 8 ) + trailer[i];
    if( crc != decoder.crc() )
      { std::fputs( "CRC mismatch\n", stderr ); retval = 2; }

    unsigned long long data_size = 0;
    for( int i = 11; i >= 4; --i )
      data_size = ( data_size << 8 ) + trailer[i];
```

```
  if( data_size != decoder.data_position() )
    { std::fputs( "Data size mismatch\n", stderr ); retval = 2; }

  unsigned long long member_size = 0;
  for( int i = 19; i >= 12; --i )
    member_size = ( member_size << 8 ) + trailer[i];
  if( member_size != decoder.member_position() )
    { std::fputs( "Member size mismatch\n", stderr ); retval = 2; }
  if( retval ) return retval;
  }

if( std::fclose( stdout ) != 0 )
  { std::fprintf( stderr, "Error closing stdout: %s\n",
                  std::strerror( errno ) ); return 1; }
return 0;
}
```

Concept index