COBB

September 1991
Vol. 2 No. 9

INSIDE MICROSOFT C

Tips & techniques for Microsoft C

Saving space with data compression

By Gary Conway

See Listing 1A on page 13

Data compression is an easy concept to understand. It's simply any technique that lets you take information in one format and convert it to a format that requires less memory or disk space. However, data compression is a relatively new topic (it was impractical before computers came along), and it can be very confusing since many compression algorithms have their bases in mathematics and information theory.

As we examine the topic of data compression, we'll cover a bit of background about the two types of data compression techniques. Then, we'll look at a method called *Run Length Encoding* (RLE). While RLE is not as sophisticated as other methods currently in use, it's a good scheme to study for newcomers to the subject of data compression. Finally, we'll present a program that implements the RLE algorithm.

Two data compression types: logical and physical

Almost all data compression techniques fall into one of two basic categories: logical or physical. The primary distinction between these two categories is that logical compression techniques know in advance and take advantage of the nature of the input data stream—physical compression techniques do not.

Logical compression

To get a feeling for how logical compression works, let's look at two logical compression techniques. First, let's suppose you want to compress a dBASE database file. It's a pretty good bet that not all the fields in each record are filled. With this in mind, you can replace unused byte positions with special compression characters. This type of compression is useful, although not universally applicable because it requires prior knowledge of the file structure.

An example of another logical compression technique is compressing ASCII files that contain no extended ASCII characters—those characters with ordinal values greater than 127. In this case, the eighth bit of each character is 0, so you can shift the entire input data stream by 1 bit for each character, thus reducing the file size by 12.5 percent. Again, you must know beforehand that the input data stream contains no extended ASCII characters. You can use numerous methods that are similar, but each presupposes a knowledge of the data structures contained in the incoming data stream. In short, logical compression techniques don't provide solutions to generic data compression.

Physical compression

Unlike logical compression, physical compression requires no prior knowledge of the data in the input stream and thus has much broader application. We'll concentrate on physical compression techniques in the remainder of this article.

Null Suppression was one of the first physical compression techniques to find widespread use. This technique scans the input stream for contiguous blocks of blank or null characters and replaces each block with one ordered pair of characters—a special compression character and a count character. The IBM 3780 BISYNC transmission protocol still employs this technique to increase transmission throughput.

You can explore several other physical compression techniques, including Bitmapping, Run Length Encoding, Half-byte Packing, Diatomic Encoding, Relative Encoding, Huffman Encoding, and Adaptive Encoding. Just as with sorting algorithms, compression techniques exhibit optimum performance when you use them correctly and deteriorated performance when you use them incorrectly. Correctly—in

Continued on page 2

IN THIS ISSUE

INSIDE MICROSOFT C

Inside Microsoft C (ISSN 1047-6075) is published monthly by The Cobb Group.

Prices: Domestic .....$69/yr. ($7.50 each)
Outside US....$89/yr. ($9.00 each)

Address: The Cobb Group
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220

Phone: Toll-free ..(800) 223-8720
Local .....(502) 491-1900
FAX .....(502) 491-4200

Staff: Editor-in-Chief.....David Reid
Consulting Editor .....Joseph Pierce
Editing.....Linda Watkins
Martha A. Clayton
Production .....Bryan Chamberlain
Publications Mgr .....Maureen Spencer
Design .....Karl Feige
Publisher.....Douglas Cobb

Address correspondence and special requests to The Editor, *Inside Microsoft C*, at the address above. Address subscriptions, fulfillment questions, and requests for bulk orders to Customer Relations, at the address above.

Postmaster: Send address changes to *Inside Microsoft C*, P.O. Box 35160, Louisville, KY 40232. Second class postage is paid in Louisville, KY.

Conventions

To avoid confusion, we'd like to explain a few of the conventions used in *Inside Microsoft C*.

When we describe programs, we'll either print them as a figure in the article (if the listing is small) or put the listings at the end of the journal and use callouts when we describe sections of the program. A callout isn't guaranteed to compile or run because it is only a fragment of code. Complete listings in figures or at the end of the journal will compile and run.

You'll also notice that the journal is peppered with words or phrases in a `monospaced font`. We use the monospaced font for function names, reserved words, variable names, class names, and so forth to show that they are references to the program or article topic.

We use the monospaced font whenever we print code in a callout or refer to a function name, variable name, array name, class name, or reserved word. When we name entities by concatenating words, we use one of two styles: `CapitalizingEachWord` or `underlining_between_words`.

Continued from page 1

Saving space with data compression

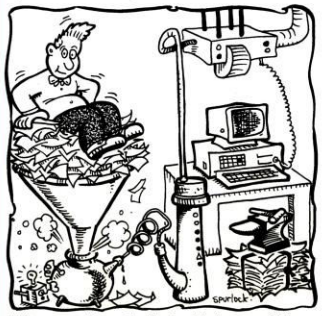
this case—means choosing the proper compression technique that yields the smallest file in the least amount of time. On occasion, the best method is actually a combination of techniques. That is, you pipe the output from one method into the input of another method. Although the end results are non-linear, they can be very attractive.

You can use several statistics to evaluate the effectiveness of data compression techniques. One typical measure applied to compression techniques is the compression ratio—the length of the input stream divided by the length of the output stream. A high compression ratio (greater than 1.25) means the data are highly compressed; a low compression ratio (1.01 to 1.25) means the data are somewhat compressed; a compression ratio less than or equal to 1 means compression failed, and the output stream is greater than or the same size as the input stream. Typically, you must balance your desired compression ratio against the execution speed of the algorithm since, as a general rule, execution time increases along with the compression ratio.

Run Length Encoding

Now that we've covered the basics of data compression, let's examine an actual compression technique called Run Length Encoding (RLE). RLE is a superset of the Null Suppression algorithm. The RLE algorithm scans the input data stream for runs of matching characters. Upon finding four or more identical characters in a row—a run—it replaces them with a special compression character and a count character.

The RLE algorithm responds only to runs of four or more characters because it can't save space by encoding runs of only two or three characters. This limit exists because the compression technique requires exactly three characters to represent a run in compressed form.



Wilbur never let practicality stand in the way of his quest for new data compression techniques.

Before looking at any code, let's manually work through an example. Suppose you have the data

```
A, BB, CCC, DDDD, EEEEE
```

As you can see, the last two groups of characters—the *D* and *E* groups—both contain four or more identical characters, which makes them candidates for RLE compression. After using RLE with the special compression character 0x90, the compressed output appears as

```
A, BB, CCC, D 0x90 3, E 0x90 4
```

Notice that RLE replaces each compressible run with a three-character sequence made up of the original character, the compression character 0x90, and the number of additional occurrences of the original character in the run. In this case, RLE reduced the number of bytes from 19 to 16 and realized a 19 percent decrease in size and a compression ratio of $^{19}/_{16}$ or 1.19.

You can apply this method to all types of data because it makes no assumptions about the type or format of the data. However, you might wonder what happens if a 0x90 character—the compression character in this example—actually appears in your file as data. You can handle this simply by outputting the 0x90 character followed by a 0 count byte. This works because the RLE decompression algorithm outputs a 0x90 character whenever it finds a 0 byte following a 0x90 character in the compressed file.

An example compression program: RLE.C

To demonstrate how RLE data compression works, we created the RLE.C program in Listing 1A on page 13. This program accepts the names of the input and output files as command line arguments and compresses the data stream from the input file into the output file. As a precaution, we coded RLE.C so it wouldn't overwrite existing files: If the output file already exists, RLE reports the error and terminates.

RLE.C consists of two functions: `main()` and `Compressor()`. After `main()` opens the input and output files, it remains in a `while` loop that calls `Compressor()` once for each character in the input stream. `Compressor()` is a State machine with five unique states: `FirstTime`, `LoneChar`, `SmallRun`, `SendRLECnt`, and `SendNewChar`. For each state, `Compressor()` takes a different set of actions and returns to the caller. Moreover, each time `Compressor()` returns, it emits a return value that the `while` loop writes to the output file (except, of course, in the case of `EOF`).

The FirstTime state

The first time `main()` calls `Compressor()`—or when the value of `State` equals `FirstTime`—`Compressor()` reads a character from the input file into the variable `PrevC` and sets `State` to `LoneChar`. Then, `Compressor()` returns `PrevC` to the `while` loop in `main()`, which writes `PrevC` to the output file. `Compressor()` enters the `FirstTime` state only at the beginning of the input file and each time after reading a 0x90 character from the input file.

The LoneChar state

`Compressor()` enters the `LoneChar` state each time it moves a lone character from the input stream directly to the output stream. The `LoneChar` state determines whether this lone character is the first in a run of four or more identical characters (a large run), the first in a run of two or three characters (a small run), or not part of a run at all. Since `LoneChar` is the most complex state in `Compressor()`, we're going to examine its instructions in detail.

The first step `Compressor()` takes in this state is testing `PrevC`'s (the previous character's) value to see if it was the `Repeat` character (0x90) or a regular character. If `PrevC` equals `Repeat` and `State` equals `LoneChar`, `Compressor()` knows that the previous iteration stored `PrevC` as a data byte, not as the RLE repeat identifier. In this case, `Compressor()` identifies the 0x90 character as data by emitting a 0.

If `PrevC` is a regular character (not equal to `Repeat`), `Compressor()` must determine if the next character from the input file matches `PrevC`. If the characters don't match, `Compressor()` simply returns the newly read character and remains in the `LoneChar` state. However, if the characters match, `Compressor()` must determine if any succeeding characters match—in other words, if there is a run of repeating characters in the file.

When `Compressor()` encounters a run of characters, it determines if it's a small run (fewer than four characters) or a large run (four or more characters). If it's a small run, `Compressor()` sets `SmallCount` to the run length, sets `State` to `SmallRun`, and emits the character in `PrevC`. However, if the run is four or more characters, `Compressor()` sets `Count` to the run length, `State` to `SendRLECnt`, and emits a `Repeat` character.

The SmallRun state

Whenever `Compressor()` finds a run of two or three matching characters, it sets `SmallCount` to the length of the run and enters the `SmallRun` state. In the next iteration, `Compressor()` executes the code following the `case` statement for the `SmallRun` state. `Compressor()` remains in the `SmallRun` state for either one or two iterations, depending on whether two or three characters were in the run. Eventually, after emitting all characters in the small run, `Compressor()` switches back to the `LoneChar` state.

The SendRLECnt state

After entering the `SendRLECnt` state and emitting a `Repeat`, `Compressor()` executes the code following the `case` statement for the `SendRLECnt` state. This code simply switches `Compressor()` to the `SendNewChar` state and emits the value `Count` minus 2.

Why `Count` minus 2? Since RLE encodes runs of only four or more characters, you'll never use the repeat count values 1 and 2 if you interpret the repeat count literally. Therefore,

to have the largest possible range of repeat counts, you should normalize all repeat counts by subtracting 2. In this way, you can represent repeat counts 3 to 257 by mapping them to the range 1 to 255. Of course, you must account for this in the decompression algorithm by adding 2 to `Count` before expanding the run back to its original length.

The SendNewChar state

`Compressor()` reaches the `SendNewChar` state after emitting a complete three-character encoded run. At this point, the variable `C` holds the last character read from the input stream—the character ending the run by not matching. For the `SendNewChar` state, `Compressor()` sets `State` to `LoneChar`, `PrevC` to `C`, and emits the character in `C`. On the subsequent iteration, `Compressor()` restarts the cycle by looking for a run of characters matching `PrevC`.

The program continues switching between states according to the above rules until it finds the end of the input file. At that time, it closes the input and output files and exits.

An example decompression program: UNRLE.C

Similar to RLE.C, the UNRLE.C program in Listing 1B on page 14 accepts the input and output filenames as command line arguments and decompresses the data stream from the input file into the output file. Also like RLE.C, UNRLE.C won't overwrite existing files. By simply glancing at UNRLE.C, you can tell that decompression is a much simpler task than compression.

UNRLE.C consists of a `main()` function and a `Decompress()` function, which performs all decompression tasks. After opening the input and output files, `main()` enters a `while` loop that calls `Decompress()` for each character in the input stream. `Decompress()` is a State machine just like the `Compressor()` function in RLE.C. However, `Decompress()` is much simpler and has only two states: `FirstTime` and `Repeating`.

`Decompress()` starts in the `FirstTime` state, writing all input characters except `Repeat` (0x90) directly to the output file. When `Decompress()` encounters a `Repeat` character in the input stream, it switches to the `Repeating` state. The `Repeating` state reads the next input character to see if it's 0. If it is, `Decompress()` writes a 0x90 to the output file; otherwise, it enters a `while` loop that expands the encoded run of characters to its original length and writes the run to the output file. In either case, `Decompress()` always switches back to the `FirstTime` state after processing for the `Repeating` state completes.

Some afterthoughts

The RLE implementation we've presented in this article—while very effective—is just one of many RLE implementations used throughout the computer industry. Some use repeat characters other than 0x90, some have special escape characters with application-specific meanings, and some compress data streams built (conceptually) of 4-bit nibbles or 16-bit words instead of 8-bit bytes.

Our algorithm's strong point is the way it differentiates between compressible runs (four or more characters) and non-compressible runs (two or three characters). Consequently, it can achieve very high compression ratios. However, there is a remote possibility you may encounter files with high concentrations of 0x90 characters. Since each 0x90 in the input stream results in two characters in the output file, such files may result in low compression ratios.

If you're a real stickler for performance, you may alter RLE.C and UNRLE.C to monitor the populations of all 256 ASCII characters in the input stream and dynamically choose the repeat character as the one occurring least frequently in the input stream. As long as the compression and decompression algorithms agree on when to switch from one repeat character to another (based on the data in the original input stream), such an approach will give you a nearly bullet-proof RLE algorithm. Of course, this enhancement will increase your algorithm's average compression ratio at the expense of execution time. In the end, you have to decide when enough is enough—do the advantages gained by further enhancements to the algorithm justify slower program execution?

Conclusion

Since understanding the concepts behind data compression algorithms is often easier than implementing them, we hope you can use RLE.C and UNRLE.C to begin your exploration of data compression techniques.

Gary Conway is the president of Infinity Design Concepts Inc. (IDC) and co-developer of the ZIP file format. For information on IDC's products—which include NARCTM, IDCSHELL™, and IDCCOM™—call (502) 636-1234.

Got a hot tip?

Do you have a tip, trick, or routine you'd like to share with other readers? If so, send it in. We may use your idea as the basis for an article. If we publish your idea, we'll give you a byline and pay you $25 or more, depending on its scope and usefulness. Send all submissions, which become property of The Cobb Group, to

Editor-in-Chief
Inside Microsoft C
The Cobb Group
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220