Ingé4 - Travaux Pratiques de C++
TP3 : Lempel–Ziv–Welch (binôme)

Johan Thapper
Polytech Paris-Sud, 2012-2013

January 17, 2013

1 Aim of the TP

The aim of this TP is to implement the LZW compression algorithm in two classes (one for encoding and one for decoding) with stream-like interfaces. To your help you are given the code for writing and reading variable width bit strings to/from a stream. You are also given the pseudo-code for the encoding and decoding algorithms. In this TP you will learn how to make a simple implementation of the LZW compression algorithm using the standard STL containers and how to simulate a stream interface to this algorithm. You will also learn a little bit about the efficiency of those containers.

2 Turn-in

Your files should be sent in an email to `thapper@lri.fr`. The deadline is set to Sunday February 3, 23:59. You have the choice of either using `tar` + `gzip` to compress your files, or to use `tar` + the LZW-algorithm that you have written. If you want to use your LZW algorithm, then it is very important that your code follows precisely the pseudo-code given here. Otherwise, I will be unable to open it!

2.1 Grading

- For a passing grade, you only need to work on Part I.
- For higher grades, you should also implement some of the improvements suggested in Part II.

3 The LZW Algorithm

The LZW compression algorithm is an improvement by Welch of the LZ78 compression algorithm invented by Lempel and Ziv. The idea behind the algorithm is to maintain a dictionary of *code words* for substrings of the input and to write these code words instead of the original strings. The lengths of the code words increase as the dictionary fills up. Therefore the dictionary is periodically cleared.

The dictionary is never saved to the compressed file, but the encoding algorithm is designed so that the decoder can still maintain its own copy of the dictionary and know precisely what each code word means. Usually, the decoder has already stored the incoming code word in its dictionary, but once in a while, it receives a code word that it does not recognise. If this is the next code word in the sequence, then the decoder will be able to guess the right character string. If the unknown code word is further along in the sequence, then the input is incorrect.

To your help, this document contains the full pseudo-code for the main parts of the encoder and decoder. For further explanation of the LZW algorithm, see:

- http://fr.wikipedia.org/wiki/Lempel-Ziv-Welch
- http://en.wikipedia.org/wiki/Lzw

3.1 Files

The folder http://www.lri.fr/~thapper/courses/cpp-2012/TP3/ contains some files necessary to complete this TP:

- `TP3.pdf`
- `obitstream.h` and `obitstream.cpp`
- `ibitstream.h` and `ibitstream.cpp`
- `extra.h` and `extra.cpp`

4 Part I – Compress and Decompress

Follow steps 1. through 5. below to complete the first part of the TP. Details on how to implement the classes `olzwstream` and `ilzwstream` are given in Sections 4.1 through 4.3.

1. Write the class `olzwstream`. Use the screen for output while debugging. Try to encode the string `TOBEORNOTTOBEORTOBEORNOT` and compare your result with the following table:

output #	code word	meaning
1	256	`clear_code`
2	84	`T`
3	79	`O`
4	66	`B`
5	69	`E`
6	79	`O`
7	82	`R`
8	78	`N`
9	79	`O`
10	84	`T`
11	258	`TO`
12	260	`BE`
13	262	`OR`
14	267	`TOB`
15	261	`EO`
16	263	`RN`
17	265	`OT`
18	257	`end_code`

2. Once the class is working, change to `obitstream` for output.

3. Now it is time for a more difficult test. Write a main program that creates an image and writes it to a file using the `write_gif`-function in `extra.h`. You can for example use the Mandelbrot fractal image from lecture 5[1].

4. Write the class `ilzwstream` using the class `ibitstream` for input.

5. Write a small interface to your two classes that allows the user to compress or decompress a file. Test your program on the file http://norvig.com/big.txt.

4.1 Details

You will implement the LZW compression algorithm using two classes: `olzwstream` and `ilzwstream`. The following illustrates how the `olzwstream`-class is intended to be used:

```
ofstream f("fichier");
olzwstream lzw(&f);
lzw.put('a');
// ...
lzw.close();
```

The idea is that when you write to an `olzwstream`-object (using `put`), the data gets encoded and written to another stream (the `ofstream f` in the example). Similarly, when you read from an `ilzwstream`-object (using `get`), the object decodes some compressed data from an ordinary stream (e.g., a file), and hands you a byte. Please respect the basic interface suggested below, but feel free to improve it.

Some notation: $\emptyset$ means an empty sequence of characters. $\langle \texttt{c} \rangle$ means the sequence containing only the character `c`. If `last` is a sequence, then $\langle \texttt{last}, \texttt{c} \rangle$ means the sequence consisting of the characters in `last` followed by the character `c` added to the end. `last`.*first* means the first character of the character sequence `last`.

The following variables and constants will be needed:

- `last` is a sequence of characters that the encoder is currently treating.
- `min_code_size` = 8 is the starting code size.
- `max_code_size` = 12 is the maximum code size.
- `cur_code_size` is the number of bits that each code word currently uses. Whenever the dictionary contains more code words than can be represented using `max_code_size` bits, the algorithm will reinitialize the dictionary and reset the variable `cur_code_size` to `min_code_size`.
- `nb_symbols = (1ul << min_code_size)` is the number of sequences and code words initially inserted into the dictionary. This will by 256, one for each byte.
- `clear_code` = `nb_symbols` is a code word that indicates that the dictionary should be cleared.
- `end_code` = `nb_symbols`+1 is a code word that indicates the end of the compressed data.
- `next_code` is the next available code word.

For the bitstream classes to work properly, it is necessary that the type `uint_32` defined in `extra.h` is at least 32 bits. If the type `unsigned int` on your system is less than this, please change the `typedef` in `extra.h` to `unsigned long`.

[1] http://www.lri.fr/~thapper/courses/cpp-2012/lecture5/mandelbrot.cpp

4.2 The Class `olzwstream`

The class `olzwstream` represents the encoder. Its constructor takes an `ostream`-pointer `strm` as input and will write the encoded stream to an `obitstream`-object. For the debugging phase, you may ignore `strm` and write directly to the terminal using `cout`.

The class should contain:

- A constructor that takes as argument an `ostream`-pointer.
- A method `void put(char c)` that takes an input byte (the character `c`) and (possibly) writes some encoded data to the stream. The pseudo-code for `put` is given in Section A.
- A method `void close()` that terminates the encoding. This method should write the code word for `last`, then write the code word `end_code`, and finally call `flush()` on the `obitstream`.

The class will need to contain a dictionary that can translate a sequence of characters into a code word. To start with, you can use `map<vector<char>,uint_32> dict`. Before you start encoding, you need to initialize the dictionary and write out a `clear_code`. It will sometimes also be necessary to initialize the dictionary during the encoding, so it is best to write a method `void initialize()` that does this job. This method starts by clearing the dictionary and the sequence `last`. It then adds the sequence ⟨`c`⟩ with code word `c` to the dictionary, for each `c` between 0 and `nb_symbols`−1. Finally, it sets the variables `next_code` = `nb_symbols`+2 and `cur_code_size` = `min_code_size`+1.

4.3 The Class `ilzwstream`

The decoder will sometimes translate a single code word into several characters. Therefore, it will be necessary to keep an internal buffer in the class `ilzwstream`. You can use `list<char> buffer` for this purpose.

The class should contain:

- A constructor that takes as arguments an `istream`-pointer.
- A method `void read_to_buffer()` that reads a code word and adds the decoded characters to the internal buffer.
- A method `int get(char& c)` that reads a byte from the internal buffer into `c`. The return value is 1 if a character was read and 0 otherwise.
- A method `bool eof()` that returns `true` if the decoder has read the `end_code` *and* the buffer is empty. Otherwise it returns `false`.

The dictionary and the `initialize`-method are almost the same as for `olzwstream`, but the decoder needs to translate in the opposite direction; from code words to sequences of characters. You can define the dictionary to be `map<uint_32,vector<char> > dict`[2].

To write the method `read_to_buffer()`, use the pseudo-code in Section A. The method `get(char& c)` can then be implemented by calling `read_to_buffer()` (if necessary) and then (a) if the buffer is still empty, set an internal flag indicating that there is nothing more to read and return 0; (b) if the buffer is not empty, then take one character from the buffer, store it in `c`, and return 1.

[2]There are some easy ways to improve this definition.

5 Part II – Improvements

This section suggests some possible improvements to the TP. If you have other ideas, please feel free to implement them.

5.1 Error Handling

What happens if you run your decoder on a corrupt file? Add some basic error handling both to the `ibitstream`/`obitstream`-classes and to your own classes.

5.2 GIF Handling

Create a class `GIF_image` that can read and write a gif image and that gives access to the image (size, data, and colour table). Write a small application that exemplifies the use of your class. For a description of the GIF file format, see the following links:

- http://en.wikipedia.org/wiki/Graphics_Interchange_Format
- http://www.fileformat.info/format/gif/spec/index.htm

5.3 Faster Compression/Decompression

If you have used standard STL containers such as `map` and `vector`, the implementation is quite slow. Improve the running time. Document what you have changed and describe why. Note that you are not supposed to improve the compression ratio–the basic algorithm must be the same.

A Pseudo-code for `olzwstream::put(char c)`

- Remember to change the bit length of the `obitstream` to `cur_code_size` whenever the latter changes.

Algorithm 1: Pseudo-code for `olzwstream::put`

```
Input: A character c
if ⟨last, c⟩ is in the dictionary then
    last = ⟨last, c⟩
else
    write the code word for last
    insert ⟨last, c⟩ in the dictionary with code word next_code
    if next_code == (1ul << cur_code_size) then
        if cur_code_size == max_code_size then
            write clear_code
            initialize() the dictionary
        else
            increase next_code by 1
            increase cur_code_size by 1
        end
    end
    last = ⟨c⟩
end
```

B Pseudo-code for `ilzwstream::read_to_buffer()`

- Remember to change the bit length of the `ibitstream` to `cur_code_size` whenever the latter changes.

Algorithm 2: Pseudo-code for `ilzwstream::read_to_buffer`

```
read a code word into cw
if cw == clear_code then
    initialize() the dictionary
    last = ∅
    read a code word into cw
end
if cw == end_code then
    set an internal flag indicating that there is nothing more to read
    return
end
if cw is in the dictionary then
    let str be the character sequence for cw
else
    let str be ⟨last, last.first⟩
end
add str to the buffer
if last is not empty then
    insert the code word next_code in the dictionary with character sequence
    ⟨last, str.first⟩
    increase next_code by 1
    if next_code == (1ul << cur_code_size) then
        if cur_code_size < max_code_size then
            increase cur_code_size by 1
        end
    end
end
last = str
```