COMP 3512
Assignment 1

1 Introduction

The purpose of this assignment is to write a program that allows us to use commands to "format" text.

The input is a text file consisting of a nested list of words. By a list of words, we mean words enclosed between brackets. A list is started by an opening bracket, the character '(', that we'll refer to as the left delimiter & terminated by a matching closing bracket, the character ')', that we'll refer to as the right delimiter. Basically, the first word after the left delimiter is the command that specifies how to format the rest of the text in that list.

For example, in the following text

```
(normal Read these instructions (italic carefully).  This is a closed-book
exam.  There are 6 questions with a total of 25 marks.  Answer (bold all)
questions.  Time allowed: (underline 80 minutes).)
```

the intention of the command `italic` is to indicate the use of italic fonts for the word `carefully`; similarly, `bold` & `underline` indicate other ways to format the delimited text. We can think of the command `normal` as indicating the use of a normal font.

Since our program runs in a text console, it isn't possible to change fonts. We'll use commands to indicate how to "highlight" text by changing the foreground colour. This can be accomplished using ANSI escape codes.

2 ANSI Escape Codes

ANSI escape codes are special sequences of characters that make it possible, among other things, to change the foreground and background colours of displayed text. For example, after printing the 7-character sequence "`\e[0;31m`" (here we use `\e` to denote the single escape character) to the console, any text that is printed to the console afterwards will be displayed in red (until this is changed by another escape code).

In the ASCII encoding, the escape character is octal 33. The following C++ statement

```
cout << "\033[0;31mhello \033[39;49mworld";
```

prints "hello" in red and "world" in "default" colors (whatever that means). Some compilers accept the escape sequence '`\e`' for the escape character, but this is *non-standard* & should be avoided — use '`\033`' (or '`\x1b`') instead.

The following are some ANSI escape codes that change the foreground colour. Note that they all start with '`\033[`' and end with an '`m`'.

Black	`\033[0;30m`
Red	`\033[0;31m`
Green	`\033[0;32m`
Brown	`\033[0;33m`
Blue	`\033[0;34m`
Purple	`\033[0;35m`
Cyan	`\033[0;36m`
Grey	`\033[0;37m`

Actually, it is possible to change both the background & the foreground colours with one escape code. For example, the escape code "`\033[0;31;43m`" specifies "red on yellow".

3 Highlighting Text

Each command specifies a mode which may correspond to colour (specified by an escape code). So for this assignment, we basically need to map commands to "escape codes". In the example text from section 1, reproduced below,

```
(normal Read these instructions (italic carefully).  This is a closed-book
exam.  There are 6 questions with a total of 25 marks.  Answer (bold all)
questions.  Time allowed: (underline 80 minutes).)
```

we may, for example, choose to use grey colour as the normal mode & "map" italic mode to blue, bold mode to red, & underlined mode to green. (These "colours", or rather their escape codes, are specified in a configuration file — see section 5.) This means that "`carefully`" is printed in blue, "`all`" in red, "`80 minutes`" in green & the rest in grey. To achieve this, our program needs to

1. print '`\033[0;37m`' (for grey, which is the normal colour) when it sees the command `normal`
2. print '`\033[0;34m`' (for blue) when it sees the command `italic`
3. switch back to the previous colour (grey) by printing an appropriate escape code) when it sees the matching right delimiter that terminates the `italic` command (the right delimiter that follows the word `carefully`
4. print '`\033[0;31m`' (for red) when it sees the command `bold`
5. switch back to the previous colour (grey) when it sees the right delimiter that terminates the `bold` command (the right delimiter that follows the word `all`)
6. print '`\033[0;32m`' (for green) when it sees the command `underline`
7. switch back to the previous colours (grey) when it sees the right delimiter that terminates the `underline` command (the right delimiter that follows the word `minutes`)
8. switch to the "default mode" (by printing a string specified by the macro `DEFAULT_MODE`) when it sees the final right delimiter

The default value of `DEFAULT_MODE` is the empty string. But it can be configured at compile time. For example, for `g++` under the bash shell, we can use the switch

```
-DDEFAULT_MODE='"\033[39;49m"'
```

to specify a value of `"\033[39;49m"` for `DEFAULT_MODE`.

With one exception, all whitespace in the input between the very first left delimiter & its matching right delimiter is significant & must be preserved in the output. The exception is whitespace characters that follow a command name — they are dropped from the output. (This means, for example, that a line break following a command will be dropped.) Note that names of commands cannot contain whitespace.

Note also that leading whitespace before the first left delimiter & trailing whitespace after the matching right delimiter are dropped from the output as specified in section 4.

4 Lists, Delimiters & Brackets

A list is started by a left delimiter & terminated by a matching right delimiter. Lists can be nested. For example:

```
(normal this (bold is a(italic short
)simple) test)
```

In this case, when the `italic` command is terminated, the program should switch back to the mode associated with `bold`.

Unmatched left or right delimiters are an error. Also, a list must contain a command which means that it must contain at least one word. Hence the following 3 sets of input are all invalid:

```
(normal this (bold is a short) simple
) )test)

(normal this (bold is a (italic short)
simple test)

(normal this ( ) (bold is a short) simple
test)
```

The first has too many right delimiters whereas the second has too few of them. The last has an empty list.

Another requirement for a valid input is that its very first "token" (which may be preceded by whitespace characters) must be a left delimiter & input must end with the corresponding right delimiter (perhaps with trailing whitespace). In other words, there must be a top-level list where all other lists are directly or indirectly nested. This means that the following input is not valid:

```
  this (bold is a (italic short)
  simple) test
```

However, the leading & trailing whitespaces of the top-level list are dropped from the output.

When the program encounters an error in the input file (e.g., an invalid command — see section 5 — or an unmatched left or right delimiter), it *reverts back to the "default mode"* before printing an error message (to standard error) & exiting. The error message should indicate the location (specified by the line number) of the error. (The first line of the input file is line 1.) Note that it is possible for the program to have displayed part of the "formatted" text before it encounters an error, switches back to the "default mode", prints an error message & exits.

Command names are delimited by whitespace characters, left delimiters & right delimiters. Consider the following:

```
  (normal this (bold(italic) is a short
  )simple test)
```

The 3 command names are `normal`, `bold` & `italic`. Note that `bold` is delimited by 2 left delimiters whereas `italic` is delimited by a left & a right delimiter. Note however that the `italic` mode in this example is not useful as it does not apply to any text. (For simplicity, the program should still switch to "italic" mode when it sees the command.)

In order to be able to use opening & closing brackets in our text without their being regarded as left & right delimiters, we stipulate that 2 consecutive opening brackets constitute 1 left bracket (rather than 2 left delimiters) & similarly, 2 consecutive closing brackets constitute 1 right bracket (rather than 2 right delimiters). So, for the input

```
  (start this (()) is a (((((bold very)) complicated) example)
```

the output text would be

```
  this () is a ((very) complicated example
```

Note that `"very) complicated"` would be in `bold` mode & the rest in `start` mode. (The 5 opening brackets before `bold` constitute 2 left brackets followed by 1 left delimiter; the 2 closing brackets after `very` constitute 1 right bracket.)

Note that left & right brackets do not need to match, but left & right delimiters do.

5 Configuration

To make the program flexible, the mapping of commands to "escape codes" can be configured. This configuration is stored in a configuration file. The default name of this configuration file is `config` (without extension & in the current directory) but it can be specified as a command-line argument. The program takes at most one argument on the command-line — the name of the configuration file. It reads the input text via standard input & displays the "formatted" text to standard output.

The configuration file is processed line by line. Each valid line must contain at least 2 words: the first word is regarded as the command name & the second is regarded as the "escape code". Additional words after the first two words are ignored — they can serve as comments.

Command names are case-sensitive & obviously cannot contain whitespace characters.

The "escape code" is simply the word after the command name & is not required to be an actual ANSI escape code. However, in order to make it easy to specify the escape character in this "escape code", we allow the use of the 2-character sequence `\e` (i.e., backslash followed by `e`) to denote it. If the user actually wants to have a backslash that is followed by an 'e' (rather than the escape character), they will have to use the 3-character sequence `\\e`. Note that we do not regard a backslash as special (it does not "escape" the following character) — only the 2-character `\e` sequence is specially handled if it is not preceded by another backslash. The program needs to translate appropriate `\e` sequence (in the second word of each line of the configuration file) into the actual escape character (octal 33 in ASCII).

By not insisting that the "escape code" be a valid ANSI escape code, we make it possible to test the program without actually changing colours.

As an example, the mapping used in the example in section 3 can be specified by the following configuration file:

```
  bold          \e[0;31m  # red
  italic        \e[0;34m  # blue
  underline     \e[0;32m  # green
  normal        \e[0;37m  # grey
  color(brown)  \e[0;33m
```

Note that the extra words in a line can serve as comments.

When the program encounters a line that is invalid (e.g., there are fewer than 2 words), it simply silently skips that line. Furthermore, if two or more valid lines contain the same command name, the first occurrence is the one in effect. As a result, the following configuration file has the same effect as the one above:

```
normal        \e[0;37m  # grey
italic        \e[0;34m  # this line in effect for italic
bold          \e[0;31m  # red
italic        \e[0;35m  # doesn't override previous value
underline     \e[0;32m  # green
color(brown)  \e[0;33m
BOLD
```

Note that the last line is invalid because there's no escape code and hence is skipped.

The above configuration shows a command named `color(brown)`. The following shows how you can use this command in the input:

```
(normal this is a(color((brown)) simple) example.)
```

Note that we have to use `color((brown))` in the input file.

The program reads in the mappings contained in the configuration file and proceeds to use it to format the input text (read from standard input); it displays the output to standard output.

6 Additional Requirements

In this course, we limit ourselves to C++11. Except when explicitly stated to the contrary, you must implement any class or function that you use & that is not in that standard C++ library. Failure to do so may result in a score of 0 for the assignment.

You must use C++ I/O streams & must not use global variables (except for constants).

An additional requirement is that the program must be able to operate in debug mode if it is compiled with the macro `DEBUG` defined. In this mode, the program only processes the configuration file which defaults to `config` but can be specified as a command-line argument (as in non-debug mode). After processing the configuration file, the program displays the valid command names enclosed in brackets "formatted" by the "escape codes" that are in effect & then exits. For the second configuration file in section 5, the debug mode output should be something like:

```
(bold)
(color(brown))
(normal)
(italic)
(underline)
```

except that the lines may be in a different order. (Note that in the above `(normal)` is in grey — it'll look white on a black background.)

To see more clearly what's going on, consider the following config file in which the "escape codes" don't actually contain the escape character & where `DEFAULT_MODE` has the value `"D"`:

```
bold       B
italic     I
underline  U
normal     N
```

The debug output, except for the order of the lines, is essentially as follows:

```
B(bold)D
I(italic)D
N(normal)D
U(underline)D
```

7 Submission & Grading

This assignment is due at 11pm, Saturday, November 5, 2016. Submit your source code in a zip file to `Share In` in the directory:

```
\COMP\3512\a1\set<X>\
```

where `<X>` is your set. Your zip file should be named `<lastname><firstname>_<id>.zip`, where `<lastname>` is your last name, `<firstname>` your first name & `<id>` your student ID. For example, `SimpsonHomer_a12345678.zip`. Do not use spaces in this file name.

Your zip file must unzip directly to your source file(s) without creating any directories. We'll basically compile your file(s) using

```
g++ -std=c++11 -W -Wall -pedantic *.cpp
```

after unzipping. Your program must compile *without warnings or errors* with the above command.

Do not submit rar files. They will not be accepted.

If you need to submit more than one version, name the zip file of each later version with a version number after your name, e.g., `SimpsonHomer_a12345678_v2.zip`. If more than one version is submitted, we'll only mark the version with the highest version number. (If it is unclear to us which version to mark, you may fail to get credit for the assignment.)

Assuming the above requirements are met, the grade breakdown for this assignment is *approximately* as follows:

Coding style & code clarity	10%
Handling configuration file (may require debug mode)	20%
Handling invalid input	15%
Handling left & right brackets (not delimiters)	15%
Text "formatting"	40%