# COMP 3512
# Assignment 1

## 1 Introduction

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

As in HTML, we can use tags as markups in a document to indicate how to format sections of text. As an example, in the following text

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

the intention of the start tag `<italic>` & the end tag `</italic>` is to indicate the use of italic fonts for the text between them; similarly, `<bold>`, `</bold>`, `<underline>` & `</underline>` indicate other ways to format the included text. We can think of the `<text>` & `</text>` tags as indicating the use of a default or starting font.

Since our program runs in a text console, it isn't possible to change fonts. We'll use tags 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 '`\033[0;31m`' to the console, any text that is printed to the console afterwards will be displayed in red (until that is changed by another escape code). Note that the escape character has ASCII code 33 in octal & hence can be represented by `'\033'` in C.

The following C++ statement

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

prints "hello" in red and "world" in "normal" color (whatever that means). Some compilers accepts the escape sequence '`\e`' for the escape character, but that is *non-standard* & should be avoided.

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

For this assignment, we basically need to map tags to escape codes. In the example text from section 1, reproduced below,

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

we may, for example, choose to use grey as the starting (foreground) colour & "map" italic text to blue, bold text to red, & underlined text to green. (These colours are specified in a configuration file — see section 4.) This means that "`carefully`" is printed in blue, "`all`" is in red,"`80 minutes`" is in green & the rest is in grey.) To achieve this, our program needs to

1. print '`\033[0;37m`' (for grey, which is the starting colour) when it sees the tag `<text>`
2. print '`\033[0;34m`' (for blue) when it sees the tag `<italic>`
3. switch back to the previous colour (grey) when it sees `</italic>` (by printing an appropriate escape code)
4. print '`\033[0;31m`' (for red) when it sees the tag `<bold>`
5. switch back to the previous colour (grey) when it sees `</bold>`
6. print '`\033[0;32m`' (for green) when it sees the tag `<underline>`
7. switch back to the previous colours (grey) when it sees `</underline>`

Note that `</text>` does not require any handling (except to check that it is there) as we cannot revert to the colour before the starting colour. (We don't know what it is.) For a valid input, the colour should have reverted back to the starting colour before the program encounters `</text>`. (In the above, it returns to the starting colour when it encounters `</underline>`.)

Note that tags can be nested. For example:

```
this <bold>is a <italic>short</italic> simple </bold>test
```

In this case, when the `</italic>` tag is encountered, the program should switch back to the colour associated with `<bold>`.

However, tagged regions that are not nested are not allowed to "overlap". The following would be invalid:

```
<text>this <bold>is a <italic>short</bold> simple </italic>test</text>
```

In this case, the program should print an error message & exit.

All whitespace in the input file between `<text>` & `</text>` is significant & must be preserved in the output. Note that tag names cannot contain whitespace.

## 4 Configuration

To make the program flexible, the mapping of tags to escape codes can be configured. This configuration is stored in a configuration file. The default name of this configuration file is `config.txt` (in the current directory) but it can be specified as a command-line argument to the program.

Note that tags are delimited by "`<`" and "`>`" (for a start tag) or by "`</`" and "`>`" (for an end tag). The characters between the delimiters form the tag name. The configuration file specifies how tag names are mapped to "escape codes".

The configuration file is processed line by line. Each valid line must contain at least 2 words: the first must be a valid tag name, the second is regarded as an "escape code". Additional words after the two are ignored.

Tag names are case-sensitive & valid tag names must consist only of alphanumeric characters (alphabets or digits).

The "escape code" is simply the word after the tag name & is not required to be an actual ANSI escape code. Inside this word, the 2-character sequence '\' 'e' (backslash followed by `e`) is used to denote the escape character — the program needs to translate this sequence into the 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.

The tag name `text` is reserved & an "escape code" must be specified for it in the configuration file. If that is not the case, the program prints an error message & exits.

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

```
bold       \e[0;31m  # red
italic     \e[0;34m  # blue
underline  \e[0;32m  # green
text       \e[0;37m  # grey
```

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

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

```
text       \e[0;37m  # grey
italic     \e[0;34m  # this line is in effect for italic
bold       \e[0;31m  # red
italic     \e[0;35m  # doesn't override previous value
underline  \e[0;32m  # green
emphasize! \e[0;33m  # invalid
italic
```

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

## 5 The Program

The only file the program reads is the configuration file. All other input & output are performed via standard input, standard output & standard error.

As mentioned previously, the default name of the configuration file is `config.txt`. However, the program can be invoked with the name of a configuration file as its only other command-line argument.

The text to process is read via standard input & the processed text is printed to standard output. Error messages are displayed to standard error.

The first word in the input text must be `<text>` & the last word in the input text must be `</text>`. Furthermore, both `<text>` & `</text>` must occur exactly once in the input. It is an error if either of them occurs more than once. For simplicity, we require that `<text>` be the very first "word" of the very first line of the input. Note however, that there may be whitespace preceding it. Also, by "word", we are not implying that it must be delimited by whitespaces. The example input in section 1 shows `<text>` immediately followed by the word "`Read`" with no whitespace in between. Whitespace that precedes `<text>` or that follows `</text>` are dropped from the output.

The program reads in the mappings contained in the configuration file and proceeds to use it to highlight the tagged regions of the text input (i.e., it displays the input to standard output with tagged regions highlighted appropriately).

When the program encounters an error in the input (e.g., an invalid tag, perhaps because it is not one in the configuration file, or "overlapping tags" or an unmatched tag), it *reverts back to the starting colour* (the one specified by the `text` tag) before printing an error message (to standard error) & exiting. The error message should indicate the location (given by the line number) of the error. (The first line of the input is line 1.) Note that it is possible for the program to have displayed part of the highlighted text before it encounters an error, changes back to the starting colour, prints an error message & exits.

The program regards any occurrence of the character ‘`<`’ or ‘`>`’ in the input text as part of a tag. If we need a ‘`<`’ or ‘`>`’ that is not part of a tag, we need to use the character entity reference “`&lt;`” or “`&gt;`” respectively. The program needs to translate these two references into the corresponding characters in the output.

For simplicity, the program regards the characters between ‘`<`’ and the next ‘`>`’ (for a start tag), or the characters between ‘`</`’ and the next ‘`>`’ (for an end tag), as forming the tag name. The tag name must then be validated — it must be one of those specified in the configuration file. Note that since valid tag names cannot contain newline characters, a ‘`<`’ without a matching ‘`>`’ (or a ‘`>`’ without a matching ‘`<`’) *on the same line* cannot be valid. The following input, for example, is invalid:

```
if 2 < 3, then
everything's fine!
```

If we want the “less than” symbol, use

```
if 2 &lt; 3, then
everything's fine!
```

Similarly,

```
if 2 < 3, then <bold>eveything's fine!</bold>
```

is not valid either — the program would regard “`␣3,␣then␣<bold`” as the tag name which is not a valid name. (␣ denotes the space character.)

Note that `<␣bold␣>` is also invalid because its tag name “`␣bold␣`” has spaces.

## 6 Additional Requirements

You must use C++ regular expressions when processing the input (from standard input). Do not use global variables (except for constants). You may find some of the STL containers useful.

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 can still be invoked with or without the name of the configuration file on the command-line. (As before, if the configuration file is not specified on the command-line, it defaults to `config.txt`.) However, after processing the configuration file, the program displays the valid tag names enclosed in angled-brackets & “highligted” by the “escape codes” that are in effect & exits. For the configuration file in section 4, the debug mode output would be something like:

```
<bold>
<italic>
<underline>
<text>
```

except that the lines may be in a different order. (Note that in the above `<text>` 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:

```
bold       B
italic     I
underline  U
text       D
```

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

```
B<bold>D
I<italic>D
U<underline>D
D<text>D
```

Note that if the configuration file is invalid (because of the absence of the `text` tag name), the program simply prints an error message & exits.

## 7 Submission and Grading

This assignment is due at 11pm, Friday, November 6, 2015. Submit your source files in a zip file to `In` in the directory:

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

where `<X>` is your set. Your zip file must be named `<name_id>.zip`, where `<name_id>` is your name & student ID separated by an underscore (for example, `SimpsonHomer_a12345678.zip`) Do not use spaces to separate your last & first names. *Your zip file must unzip directly to your source files without creating any directories.* We'll basically compile your files using

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

after unzipping.

*Do not put executables in your zip file. Also, do not submit rar files—they will not be marked.*

If you need to submit more than one version, name the zip file of each later version with a version number after your ID, 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 use from your submissions whcih version to mark, you may fail to get credit for the assignment.)

As mentioned, you must use regular expressions when processing the input. You will lose marks if you don't.

*Your program must compile without errors or warnings under `g++` with the switches mentioned above.* If it does not meet this requirement, you may receive a score of zero for the assignment.

Assuming that your assignment meets the above requirements, the grade breakdown is *approximately* as follows:

| | |
|---|---|
| Coding style & code clarity | 10% |
| Handling configuration file (requires debug mode) | 25% |
| Handling invalid input | 20% |
| Handling character entity references (`&lt;` and `&gt;`) | 10% |
| Text output (highlighting markups) | 35% |