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 with 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 probably 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 also 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 background and/or foreground colours. 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`' (where `\e` denotes 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[0;39mworld";
```

prints "hello" in red and "world" in "normal" color (whatever that means). Note that some compilers actually accept the *non-standard* escape sequence '`\e`' for the escape character. Although we'll use '`\e`' to denote the escape character in this document, its use in the C++ program should be avoided as it is non-portable. (Use '`\033`' instead.)

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

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

Actually, it is possible to change both the background & the foreground colours with one escape code. For example, the escape code "`\e[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 black as the starting (foreground) colour & "map" italic text to blue, bold text to red, & underlined text to green. (This means that "`carefully`" is printed in blue, "`all`" is in red,"`80 minutes`" is in green & the rest is in black.) To achieve this, our program needs to

1. print '`\e[0;30m`' (for black, which is the starting colour) when it sees the tag `<text>`
2. print '`\e[0;34m`' (for blue) when it sees the tag `<italic>`
3. switch back to the previous colour (black) when it sees `</italic>` (by printing an appropriate escape code)
4. print '`\e[0;31m`' (for red) when it sees the tag `<bold>`
5. switch back to the previous colour (black) when it sees `</bold>`
6. print '`\e[0;32m`' (for green) when it sees the tag `<underline>`
7. switch back to the previous colours (black) 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.) The purpose of the `<text>` tag is to specify the starting colour that the program should have returned to by the time it 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.

4 Configuration

To make the program flexible, the mapping of tags to escape codes can be configured. This configuration is stored as a preamble with the text file. We'll call such a file an ML files.

An ML file has the following format:

```
<ml>
<config>
   # section for configuration data
</config>
<text>
   # section for the text to "highlight"
</text>
</ml>
```

i.e., it must start & end with the 2 tags `<ml>` & `</ml>` respectively. After the `<ml>` tag, there must be a configuration section which is started by the `<config>` tag & terminated by the `</config>` tag. The text section, which contains the text with markups, then follows. It is started & terminated by the `<text>` & `</text>` tags respectively. The 6 tags — `<ml>`, `<config>`, `<text>` & their corresponding end tags — are reserved &, except for `<text>` & `</text>`, can only occcur in the locations shown above. The exception is that the `<text>` & `</text>` pair must also occur once in the configuration section — this is so that we can map them to the escape code used for the starting colour. (See below.)

Note that whitespace

- before `<ml>`,
- between `<ml>` & `<config>`,
- between `</config>` & `<text>`,
- between `</text>` & `</ml>`, &
- after `</ml>`

is insignificant & is allowed. However, whitespace within the text section (between the `<text>` & `</text>` tags shown above) is significant & must be preserved in the output.

All tags are delimited by '`<`' and '`>`'. We call the text between '`<`' & '`>`' (for a start tag) or between '`</`' & '`>`' (for an end tag) the tag name. Hence there are 2 tags for each tag name — a start tag & an end tag. (Note: valid tag names must be non-empty.)

Furthermore, the program regards any occurrence of the character '`<`' or '`>`' in the the file as part of a tag. ('`<`' starts a tag; '`>`' terminates 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. (This means that we cannot use the two 4-character sequences "`&lt;`" & "`&gt;`" to represent themselves. This can be fixed if we use "`&amp;`" to represent `&`. However, for simplify, we'll not be doing this in this assignment.) The program needs to translate these two references into the corresponding characters when processing certain parts of the ML file. More details are provided below.

The configuration section contains items, each mapping a tag name to an escape code. An item has the form:

```
<tagname>escape_code</tagname>comments
```

In the above, the 2 tag names must match. The escape code is the text that is between the 2 tags. Comments can follow the end tag & last until the start tag of the next item or until `</config>` if there are no more configuration items.

Tag names are case-sensitive & valid tag names must be non-empty & must consist only of alphanumeric characters (alphabets or digits). The escape code cannot contain '<', '>', or any whitespace character. (Note however that it can be empty — in this case, the escape code has no effect as nothing is printed.) The program needs to validate each configuration item by checking that the start & end tags match, & that the tag name & escape code are valid. Furthermore, it needs to translate the 2-character sequence "`\e`" into the escape character (octal 33 in ASCII) & the 2 character entity references "`<`" & ' '`>`" into '<' & '>' whenever it encounters them in the escape code. (However, it does not need to handle other escape sequences.)

When the program encounters a configuration item that is invalid (e.g. an invalid tag name or escape code), it exits after printing an error message. It needs to do the same if more than one configuration item refer to the same tag name.

As an example, the mapping used in the above sample text can be specified by the following config section: (Amended — original version had extra '>' in the following config section)

```
<config>
  <text>\e[0;30m</text>           # the starting colour is black
  <italic>\e[0;34m</italic>       # italic is blue; this also
                                     shows a multi-line comment
  <bold>\e[0;31m</bold>            # bold is red
  <underline>\e[0;32m</underline>  # underline is green
</config>
```

(Note: if you are using a console with a black background, you may want to use white as the starting foreground colour.)

Note that whitespace is allowed between `<config>` & the first tag in the config section.

The following shows some valid & invalid configuration items:

```
<config>
  <text></text>                      # valid; empty escape code
  <data> </data>                     # invalid; whitespace in escape code
  <bold>\e[0;31m </bold>             # invalid; again whitespace in escape code
  <italic>\e[0;35m</italic>          # valid
  <underline>\e[0;32m</Underline>    # invalid; start & end tags don't match
  <emphasize!>\e[0;33m</emphasize!>  # invalid tag name
  <italic>\e[0;31m</italic>          # invalid; italic already mapped
</config>
```

Note that since the escape codes in the configuration file can be any string, it is possible to test your program without actually changing colours.

5 The Program

The program must be invoked with the name of an ML file to process. It reads in the mappings specified in the configuration section and then proceeds to use them to "highlight" the text in the text section that follows.

The program needs to translate the 2 character entity references "`<`" & "`>`" into '`<`' & '`>`' respectively in the output. For example, the output for the following

```
<text>if x &gt; 3 and x &lt; 4, then eveything's fine!</text>
```

would be

```
if x > 3 and x < 4, then eveything's fine!
```

printed in the starting format.

When the program encounters an error in the input file, either in the config or the text section, it exits after printing an error message to standard error. The error message should give a brief reason & indicate whether the error is in the config or the text section. We've already seen the validation required in the configuration section. Two examples of errors in the text section are the occurrence of an invalid tag (perhaps because it is not one in the config section), or the existence of overlapping tagged regions.

Note that it is possible for the program to have displayed part of the highlighted text before it encounters an error, prints an error message, & exits.

6 Additional Requirements

You must use C++ I/O streams. Avoid the use of 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 reads & validates the configuration section. If the section is valid, the program prints (to standard output) each tagname that is in effect in the format specified by its escape code. The program then exits & does not process the text section. For the first sample configuration section in section 4, the debug mode output would be something like

```
text
italic
bold
underline
```

except that the lines may be in a different order. Furthermore, after printing the above, the program should print the escape code for `text` to switch back to the starting colour.

7 Submission and Grading

This assignment is due at 10pm, Saturday, October 25, 2008. Instructions on how to submit your assignment will be provided.

If your program does not compile, you may receive zero for the assignment. Otherwise, the grade breakdown for this assignment is approximately as follows:

Coding style & code clarity	10%
Handling configuration section (requires debug mode)	30%
Handling invalid input in text section	10%

Handling character entity references (`<` and `>`)	10%
Text output (highlighting markups)	40%