xmlformat Tutorial

Paul DuBois `<paul@kitebird.com>`

Table of Contents

1. Introduction

This document is a user guide that provides a tutorial introduction to the **xmlformat** program. Another document, *The **xmlformat** Document Formatter*, describes the capabilities of **xmlformat** in more detail.

2. Formatting a Document

Suppose you have an XML document named `doc1.xml` that looks like this:

```
<event>
<description>I bought a new coffee cup!</description>
<date><year>2004</year><month>2</month><day>1</day></date>
</event>
```

Suppose further that you want it to look like this:

```
<event>
 <description>I bought a new coffee cup!</description>
 <date>
  <year>2004</year>
  <month>2</month>
  <day>1</day>
 </date>
</event>
```

By happy coincidence, that happens to be exactly the default output style produced by **xmlformat**. To reformat your document, all you have to do is run **xmlformat** with the document filename as the argument, saving the output in another file:

```
% xmlformat doc1.xml > output
```

Note: `%` represents your shell prompt; do not type it as part of the command.

If you are confident that the output style produced by **xmlformat** will be as you desire, you can be reckless and perform an in-place conversion:

```
% xmlformat -i doc1.xml
```

In this case, **xmlformat** reads the document from the input file, reformats it, and writes it back out to the same file, replacing the file's original contents. If you are not quite so reckless, use `-i` in conjunction with a `-b` option to make a backup file that contains the original document. `-b` takes an argument that specifies the suffix to add to the original filename to create the backup filename. For example, to back up the original `doc1.xml` file in a file named `doc1.xml.bak`, use this command:

```
% xmlformat -i -b .bak doc1.xml
```

3. Using a Configuration File

In the preceding example, the desired output style for `doc1.xml` was the same as what **xmlformat** produces by default. But what if the default style is *not* what you want? In that case, you must tell **xmlformat** how to handle your document. This is at once both the weakness and strength of **xmlformat**. The weakness is that it is extra work to instruct **xmlformat** how you want it to format a document. The strength is that it's possible to do so. Other XML formatters do not require any extra work, but that's because they are not configurable.

Suppose `doc2.xml` looks like this:

```
<example><title>Compiling and Running a Program</title>
<para>To compile and run the program,
use the following commands, where
<replaceable>source-file</replaceable>
is the name of the source file:</para><screen>
<userinput>cc</userinput> <replaceable>source-file</replaceable>
<userinput>./a.out</userinput>
</screen>
</example>
```

That's ugly, and you want it to rewrite it like this:

```
<example>

<title>Compiling and Running a Program</title>

<para>
 To compile and run the program, use the following commands,
 where <replaceable>source-file</replaceable> is the name of
 the source file:
</para>

<screen>
<userinput>cc</userinput> <replaceable>source-file</replaceable>
<userinput>./a.out</userinput>
</screen>

</example>
```

The key characteristics of this rewrite are as follows:

- Child elements of the `<example>` element are separated by blank lines, but not indented within it.
- The text inside the `<para>` element is reformatted, adjusted to 60 characters per line and indented.
- The contents of the `<screen>` element are left alone.

Unfortunately, if you run `doc2.xml` through **xmlformat**, it comes out like this:

```
<example>
 <title>Compiling and Running a Program</title>
 <para>To compile and run the program,
use the following commands, where
<replaceable>source-file</replaceable>
is the name of the source file:</para>
 <screen>
  <userinput>cc</userinput>
  <replaceable>source-file</replaceable>
  <userinput>./a.out</userinput>
 </screen>
</example>
```

This output is unsuitable. Among the offenses committed by **xmlformat**, two are most notable:

- The text of the `<para>` element has been left alone, not reformatted.
- The `<screen>` element content has been reformatted, not left intact.

In these respects, it appears that **xmlformat** has done exactly the *opposite* of what was wanted! Furthermore, had you used the `-i` option to reformat the file in place without using `-b` to make a backup, at this point you would have a file containing a `<screen>` element that you'd have to fix up by hand to restore it to its original condition.

What a worthless, worthless program!

The rewriting of the `<screen>` element points to an important lesson: Before trusting **xmlformat** with your documents, it's best to run some tests and tune your configuration as necessary to make sure it will produce the results you want. Otherwise, you may produce changes that affect the integrity of your documents. This is particularly true when they contain elements such as `<screen>` or `<programlisting>` that should be copied verbatim, without change.

Configuring **xmlformat** amounts to writing a configuration file that instructs it what to do. For `doc2.xml`, that means telling **xmlformat** to leave the `<screen>` element alone, to normalize the text of the paragraph to fill lines and wrap them to a given length, and to put blank lines around sub-elements of the `<example>` element.

Let's begin by creating a very basic configuration file. What should we call it? **xmlformat** can read configuration settings from a file named on the command line with a `-f` or `--config-file` option. This means you can name the file whatever you want. However, if you put the settings in a file named `xmlformat.conf` in the current directory, **xmlformat** will read the file automatically. That's an easier approach, because you won't need to use a command-line option to specify the configuration file. So create a file named `xmlformat.conf` that contains the following two lines:

```
screen
  format = verbatim
```

These lines specify that `<screen>` elements should be formatted as verbatim elements. That is, **xmlformat** should reproduce their content in the output exactly as it appears in the input, without modification. The first line must begin in column 1 (no preceding spaces or tabs). The second line must begin with at least one space or tab. Presence or absence of whitespace is how **xmlformat** distinguish the names of elements to be formatted from the instructions that indicate *how* to format them.

After creating `xmlformat.conf`, run **xmlformat** again to process `doc2.xml`. It reads the newly created configuration file and produces this result:

```
<example>
 <title>Compiling and Running a Program</title>
 <para>To compile and run the program,
use the following commands, where
<replaceable>source-file</replaceable>
is the name of the source file:</para>
<screen>
<userinput>cc</userinput> <replaceable>source-file</replaceable>
<userinput>./a.out</userinput>
</screen>
</example>
```

That's a little better: **xmlformat** has not destroyed the `<screen>` element by reformatting it. But problems remain: The paragraph content has not been reformatted, and there are no blank lines between sub-elements.

Let's take care of the paragraph next. To set up its formatting, add a section to `xmlformat.conf` for `<para>` elements:

```
para
  format = block
  normalize = yes
  wrap-length = 60
  subindent = 1

screen
  format = verbatim
```

The order of sections in the configuration file doesn't matter. Put them in the order that makes most sense to you. The order of option lines under the initial section line doesn't matter, either.

The first two options in the `para` section specify that the `<para>` element is a block element, and that text within it should be normalized. Turning on the `normalize` option tells **xmlformat** that it's okay to reformat the text within the element. This means that runs of whitespace within the text are collapsed to single spaces, and that whitespace at the beginning and end of the text can be adjusted (typically to put the text on different lines than the element's opening and closing tags). Enabling normalization also allows you to perform text line-wrapping and indenting. The `wrap-length` option specifies the maximum number of characters per line, and `subindent` specifies the indenting of text and sub-elements, relative to the element's own tags. Note that when **xmlformat** performs line-wrapping, it includes the currently prevailing indent as part of the line length. (For example, if the prevailing indent is 20 spaces and `wrap-length` value is `60`, lines will contain at most 40 characters following the indentation.)

After adding the `para` section to `xmlformat.conf`, **xmlformat** produces this result:

```
<example>
 <title>Compiling and Running a Program</title>
 <para>
  To compile and run the program, use the following
  commands, where
  <replaceable>source-file</replaceable>
  is the name of the source file:
 </para>
<screen>
<userinput>cc</userinput> <replaceable>source-file</replaceable>
<userinput>./a.out</userinput>
</screen>
</example>
```

The paragraph now is wrapped and indented. However, it doesn't seem to be wrapped *quite* correctly, because the `<replaceable>` element actually would fit on the previous line. This happens because no formatting options were specified for `<replaceable>` in the configuration file. As a result, it is treated as having the default element type of `block`, using the default behavior that block elements are written out beginning on a new line.

To fix this problem, we should configure `<replaceable>` as an inline element. That will cause it to be formatted inline with the other text (and thus line-wrapped along with it). Modify the configuration file to include a `replaceable` section: this:

```
para
  format = block
  normalize = yes
  wrap-length = 60
  subindent = 1

replaceable
  format = inline

screen
  format = verbatim
```

The resulting output after making this change is as follows:

```
<example>
 <title>Compiling and Running a Program</title>
 <para>
  To compile and run the program, use the following
  commands, where <replaceable>source-file</replaceable> is
  the name of the source file:
 </para>
<screen>
<userinput>cc</userinput> <replaceable>source-file</replaceable>
<userinput>./a.out</userinput>
</screen>
</example>
```

We're getting close now. All we need to do is space out the `<example>` child elements with a blank line in between. Sub-element spacing is controlled by three formatting properties:

- `entry-break` controls spacing after the opening tag of an element (that is, the spacing upon entry into the element's content).
- `element-break` controls the spacing between sub-elements.
- `exit-break` controls spacing before the closing tag of an element (that is, the spacing upon exit from the element's content).

The value for each of these formatting options should be an integer indicating the number of newlines to write. A value of `1` causes one newline, which acts simply to break to the next line. To get a blank line, the break value needs to be `2`. Modify the configuration file by adding a section for `<example>` elements:

```
example
  format = block
  entry-break = 2
  element-break = 2
  exit-break = 2
  subindent = 0

para
  format = block
  normalize = yes
  wrap-length = 60
  subindent = 1

replaceable
  format = inline

screen
  format = verbatim
```

The resulting output is:

```
<example>

<title>Compiling and Running a Program</title>

<para>
 To compile and run the program, use the following commands,
 where <replaceable>source-file</replaceable> is the name of
 the source file:
</para>

<screen>
<userinput>cc</userinput> <replaceable>source-file</replaceable>
<userinput>./a.out</userinput>
</screen>

</example>
```

We're done!

You may be thinking, "Wow, that's a lot of messing around just to format that tiny little document." That's true. However, the effort of setting up configuration files tends to be "reusable," in the sense that you can use the same file to format multiple documents that all should be written using the same style. Also, if you have different projects requiring different styles, it tends to be easiest to begin setting up the configuration file for one project by beginning with a copy of the file from another project.

4. Discovering "Inherited" Formatting Options

In the final formatting of `doc2.xml`, note that the paragraph tags appear on separate lines preceding and following the paragraph content. This occurs despite the fact that the configuration file specifies no break values in the `para` section, because if you omit formatting options for an element, it "inherits" the default properties. In the case of the `<para>` element, the relevant unspecified properties are the `entry-break` and `exit-break` values. For block elements, both have a value of `1` by default (that is, one newline), which causes a line break after the opening tag and before the closing tag.

If you want to see all the formatting options **xmlformat** will use, run it with the `--show-config` option. For example:

```
% xmlformat --show-config
*DEFAULT
  format = block
  entry-break = 1
  element-break = 1
  exit-break = 1
  subindent = 1
  normalize = no
  wrap-length = 0

*DOCUMENT
  format = block
  entry-break = 0
  element-break = 1
  exit-break = 1
  subindent = 0
  normalize = no
  wrap-length = 0

example
  format = block
  entry-break = 2
  element-break = 2
  exit-break = 2
  subindent = 0
  normalize = no
  wrap-length = 0

para
  format = block
  entry-break = 1
  element-break = 1
  exit-break = 1
  subindent = 1
  normalize = yes
  wrap-length = 60
```

```
replaceable
  format = inline

screen
  format = verbatim
```

No configuration file is specified on the command line, so **xmlformat** reads the default configuration file, `xmlformat.conf`. Then it displays the resulting configuration options. You can see that the `para` section has inherited break values from the `*DEFAULT` section.

5. Checking for Unconfigured Elements

Any elements appearing in the input document that are not named in the configuration file are formatted using the values of the `*DEFAULT` section. If the file contains no `*DEFAULT` section, **xmlformat** uses built-in default values.

If you want to see whether there are any elements in the document for which you haven't specified any formatting options, run **xmlformat** with the `--show-unconfigured-elements` option. For example:

```
% xmlformat --show-unconfigured-elements doc2.xml
The following document elements were assigned no formatting options:
title
```

As it happens, the title already formats in the desired fashion, so there's no necessity of adding anything more to the configuration file.