CSS generated content in depth: a draft for Smashing Magazine

In this post I'll discuss some possible uses of generated content. Generated content is a powerful feature of CSS. This is a draft for a future article on Smashing Magazine.

Inserting generated content

Generated content can be inserted before and after the real content of an element through the `:before` and `:after` pseudo-elements, respectively. To represent them, we can use the following fictional markup.

```
<p>
<before>Start</before>
Real content
<after>End</after>
</p>
```

And our CSS will be:

```
p:before {
content: "Start";
}
p:after {
content: "End";
}
```

As we can see, the property that actually inserts the two strings is `content`. This property accepts the following values:

none, **normal**
: The pseudo-content is not generated.

<string>
: A textual string enclosed in matching quotes.

url()
: This function allow us to insert an external resource (usually an image), as in the `background-image` property.

counter(), **counters()**
: These functions insert counters. See below for more details.

attr(attribute)
: This function allow us to insert the value of the attribute `attribute` of the given element.

Keep in mind that generated content takes up its own space on the page and its presence affects the space's computation of the element that hosts it.

Inserting strings

In the previous example, we've inserted two simple strings before and after the real content of an element. Generated content allows us to insert also more complex symbols through escaping.

```
p:before {
content: "\00A7";
padding-right: 0.2em;
}
```

The escaped sequence inside the double quotes is a hexadecimal Unicode value that refers to a paragraph symbol. We can also combine simple strings with Unicode symbols, as shown below.

```
p:before {
content: "( " "\00A7" " )";
padding-right: 0.2em;
}
```

Keep in mind that all the textual content inside the `content` property is treated literally, that is, spaces and tabulations inserted via the keyboard will be inserted in the page as well.

Inserting images

We can insert images through the `url()` function.

```
a:before {
content: url("../img/link.gif");
padding-right: 0.2em;
}
```

As we can see, this function works exactly as in the `background-image` property.

Inserting attribute values

An attribute value of an element can be inserted through the `attr()` function.

```
a[href]:after {
content: "( " attr(href) " )";
padding-left: 0.2em;
color: #000;
font: small "Courier New", Courier, monospace;
}
```

We've just inserted the value of the `href` attribute that, as you can see, is a simple textual string.

Inserting counters

The automatic numbering of CSS is controlled by two properties, `counter-reset` and `counter-increment`. Counters defined by these properties are then used with the `counter()` and `counters()` functions of the `content` property.

The `counter-reset` property can contain one or more names of counters (identifiers), optionally followed by an integer. The integer sets the value that will be incremented by the `counter-increment` property for any occurence of the given element. The default value is 0. Negative values are allowed.

The `counter-increment` property is similar to the previous property. The basic difference here is that this property actually increments a counter. Its default increment is 1. Negative values are allowed.

Now we are ready to create a practical example. Given the following markup:

```
<dl>
```

```
        <dt>term</dt>
        <dd>definition</dd>
        <dt>term</dt>
        <dd>definition</dd>
        <dt>term</dt>
        <dd>definition</dd>
</dl>
```

we want to add a progressive numbering (1, 2, 3) to each definition term (`dt`) in the list. The relevant CSS is the following:

```
dl {
counter-reset: term;
}
dt:before {
counter-increment: term;
content: counter(term);
}
```

The first rule in the previous listing sets a counter for the definition list. This is called a scope. The name (an identifier) of the counter is `term`. Keep in mind that once we've chosen a name for our counter this must be the same also in the `counter-increment` property (of course se should use a meaningful name).

In the second rule we attach the `:before` pseudo-element to the `dt` element, since we want to insert the counter exactly before the actual content of the element. Now let's take a closer look at the second declaration of the second rule. The `counter()` function accepts our identifier (`term`) as its argument and the `content` property actually generates the counter.

As you can see, there's no space between the number and the content of the element. If we want to add more space and, say, a period (.) after the number, we could insert the following string in the `content` property:

```
dt:before {
content: counter(term) ". ";
}
```

Note that the string inside the double quotes is treated literally, that is, the space after the period is inserted as we've typed it on the keyboard. In fact, the `content` property can be regarded as the CSS counterpart of the JavaScript `document.write()` method except that this property doesn't add real content to the document. Simply put, the `content` property creates a mere abstraction on the document tree but it doesn't modify it.

In case you're wondering, we can add more styles to counters by applying other properties to the attached pseudo-element. For example:

```
dt:before {
content: counter(term);
padding: 1px 2px;
margin-right: 0.2em;
background: #ffc;
color: #000;
border: 1px solid #999;
font-weight: bold;
}
```

We've just set a background color, added some padding and a right margin, made the font bold and outlined the counters with a thin solid border. Now our counters are a little more attractive.

Furthermore, counters can be negative. When dealing with negative counters, we should only keep in mind a little maths, namely the part concerning addition and subtraction of negative and positive numbers. For example, if we need a progressive numbering starting from 0, we could write the following:

```
dl {
counter-reset: term -1;
}
dt:before {
counter-increment: term;
content: counter(term) ". ";
}
```

By setting the `counter-reset` property to -1 and incrementing it by 1, the resulting value is 0 and the numbering will actually start from that value. Negative counters can combine with positive counters to create interesting results. Consider the following example:

```
dl {
counter-reset: term -1;
}
dt:before {
counter-increment: term 3;
content: counter(term) ". ";
}
```

As you can see, addition and substraction of negative and positive numbers yield a wide range of combination between counters. With just a simple set of calculations we can get a complete control over this automatic numbering.

Another interesting feature of CSS counters lies in their capability of being nested. In fact, numbering may proceed also by using progressive sublevels, such as 1.1, 1.1.1, 2.1 and so on. For example, if we want to add a sublevel to the elements of our list, we could write the following:

```
dl {
counter-reset: term definition;
}
dt:before {
counter-increment: term;
content: counter(term) ". ";
}
dd:before {
counter-increment: definition;
content: counter(term) "." counter(definition) " ";
}
```

This example is similar to the first one, but in this case we have two counters, `term` and `definition`. The scope of both counters is set by the first rule and "lives" in the `dl` element. The second rule inserts the first counter before each definition term of the list. This rule is not particularly interesting, since its effect is already known. Instead, the last rule is the core of our code because it:

1. increments the second counter (`definition`) on `dd` elements
2. inserts the first counter (`term`), followed by a period

3. inserts the second counter (`definition`), followed by a space.

Note that steps #2 and #3 are both performed by the `content` property used on the `:before` pseudo-element attached to the definition term.

Another interesting thing to remember is that counters are "self-nesting" in the sense that resetting a counter on a descendant element (or pseudo-element) automatically creates a new instance of the counter. This is useful in the case of (X)HTML lists, where elements may be nested with arbitrary depth. However, it's not always possible to specify a different counter for each list, since this approach might produce a really redundant code. In that vein, it's useful to mention the `counters()` function. This function creates a string containing all the counters having the same name of the given counter in the scope. Counters are then separated by a string. For example, given the following markup:

```
<ol>
<li>item</li>
<li>item
<ol>
<li>item</li>
<li>item</li>
<li>item
<ol>
<li>item</li>
<li>item</li>
</ol>
</li>
</ol>
</li>
</ol>
```

The following CSS numbers the nested list items as 1, 1.1, 1.1.1, etc.

```
ol {
counter-reset: item;
}
li {
display: block;
}
li:before {
counter-increment: item;
content: counters(item, ".") " ";
}
```

In this example we have only the `item` counter for each nesting level. Instead of writing three different counters (e.g. `item1`, `item2`, `item3`) and thus creating three different scopes for each nested `ol` element, we can rely on the `counters()` function to achieve this goal. The second rule is really important and deserves a further explanation. Since ordered lists have a default marker (a number), we get rid of these marker by turning the list items into block-level elements. Keep in mind that only elements with `display: list-items` have markers. Now we can look carefully at the third rule that actually does the work. The first declaration increments the counter previously set on the outermost list. Then, in the second declaration, the `counters()` function creates all the counter's instances for the innermost lists. The structure of this function is as follows:

1. its first argument is the name of the given counter, immediately followed by a comma
2. its second argument is a period inserted in double quotes.

Note that we've inserted a space after the `counters()` function in order to keep the numbers separate from the actual content of the list items.

Counters are formatted with decimal numbers by default. However, the styles of the `list-style-type` property are also available for counters. The default notation is `counter(name)` (no style) or `counter(name, 'list-style-type')` if we want to change the default formatting. In practice, the recommended styles are:

- `decimal`
- `decimal-leading-zero`
- `lower-roman`
- `upper-roman`
- `lower-greek`
- `lower-latin`
- `upper-latin`
- `lower-alpha`
- `upper-alpha`

because we should always bear in mind that we're working with numeric systems. Furthermore, we should also be aware of the fact that the specifications don't define the rendering of alphabetical systems at the end of the alphabet. For example, after 26 list items the rendering of `lower-latin` is undefined. Real numbers are thus recommended for long lists. Here's an example:

```
dl {
counter-reset: term definition;
}
dt:before {
counter-increment: term;
content: counter(term, upper-latin) ". ";
}
dd:before {
counter-increment: definition;
```

```
content: counter(definition, lower-latin) ". ";
}
```

We can also add styles to the `counters()` function, as shown in the following example.

```
li:before {
counter-increment: item;
content: counters(item, ".", lower-roman) " ";
}
```

Note that the `counters()` function can also accept a third argument (`lower-roman`) as the last member of its arguments list, separated by a second comma from the preceding period. However, the `counters()` function doesn't allow us to specify different styles for each level of nesting.