A LIST APART

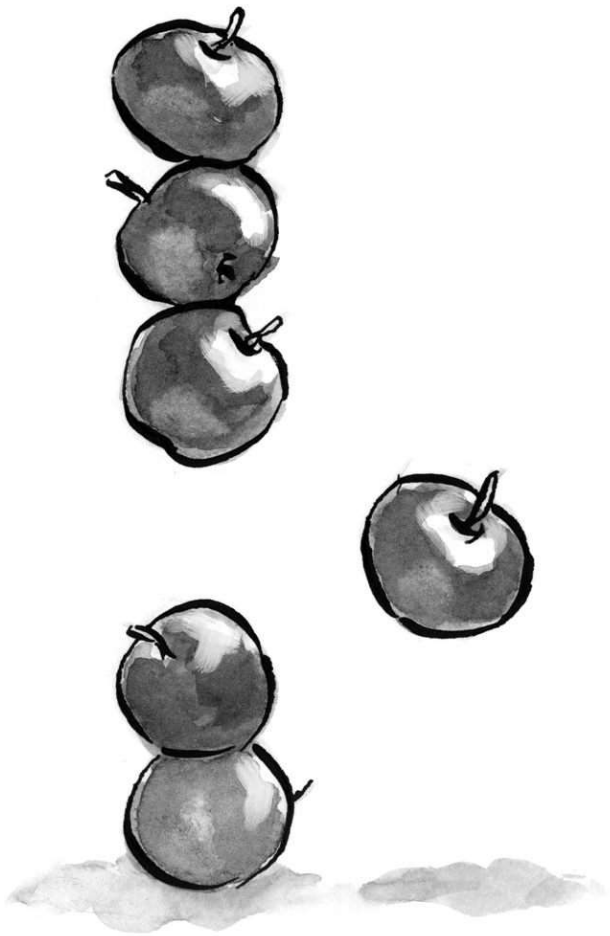

Illustration by Kevin Cornell

CSS Positioning 101

by **Noah Stokes** · November 16, 2010

Published in CSS, HTML, Layout & Grids

If you're a front end developer or a designer who likes to code, CSS-based layouts are at the very core of your work. In what might be a refresher for some, or even an "a-ha!" for others, let's look at the CSS `position` property to see how we can use it to create standards-compliant, table-free CSS layouts.

CSS positioning is often misunderstood. Sometimes, in a bug-fixing fury, we apply different `position` values to a given selector until we get one that works. This is a tedious process that can work for a time, but it behooves us to know *why* specifying something like `position: relative`

can fix your layout bug. My hope is that we can learn the `position` property's values and behaviors, and most importantly, how a value can affect your markup.

The CSS specification offers us five `position` properties: `static`, `relative`, `absolute`, `fixed`, and `inherit`. Each property serves a specific purpose. Understanding that purpose is the key to mastering CSS-based layouts.

GET WITH THE FLOW

First, let's take a step back to recognize the world we're working in. Much like our real world, in CSS, we work within boundaries. In CSS, this boundary is called the normal flow *(http://www.w3.org/TR/CSS21/visuren.html#normal-flow)*. According to the CSS 2.1 spec:

> *Boxes in the normal flow belong to a formatting context, which may be block or inline, but not both simultaneously. Block boxes participate in a block formatting context. Inline boxes participate in an inline formatting context.*

Think of a "box," as described by the spec as a wooden block—not unlike the ones you played with as a young whippersnapper. Now, think of the *normal flow* as a law similar to the law of gravity. The normal flow of the document is how your elements stack one on top of each other, from the top down, in the order in which they appear in your markup. You may remember stacking alphabet blocks into giant towers: The normal flow is no different than those wooden blocks bound by the law of gravity. As a child, you had one block on top of another; in your markup, you have one element after another. What you couldn't do as a child, however, was give those blocks properties that could defy the law of gravity. All of the sudden, CSS seems a lot cooler than those alphabet blocks.

Static and relative—nothing new here

The `static` and `relative` `position` properties behave like your childhood blocks—they stack as you would expect. Note that `static` is the default `position` value of an element, should you fail to apply any other value. If you have three statically positioned elements in your code, they will stack one on top of the next, as you might expect. Let's take a look at an example with three elements, all with a `position` value of `static`:

```
#box_1 {
  position: static;
  width: 200px;
  height: 200px;
  background: #ee3e64;
}

#box_2 {
```

```
  position: static;
  width: 200px;
  height: 200px;
  background: #44accf;
}

#box_3 {
  position: static;
  width: 200px;
  height: 200px;
  background: #b7d84b;
}
```

In example A *(/d/css-positioning-101/example_a.html)*, you can see three elements stacked like a simple tower. Fascinating, isn't it? This is block building 101. Congratulations!

You can use the `static` value for simple, single-column layouts where each element must sit on top of the next one. If you want to start shifting those elements around using offset properties such as `top`, `right`, `bottom`, and `left`, you're out of luck. These properties are unavailable to a `static` element. In fact, a `static` element can't even create a new coordinate system for child elements. Wait. What? You lost me at *coordinate system*. Roger that, Roger. Let's explain using the `relative` value.

Relatively positioned elements behave just like statically positioned elements; they play well with others, stack nicely, and don't cause a ruckus. Hard to believe, right? Take a look at our previous example. This time, we've applied the `relative` value:

```
#box_1 {
  position: relative;
  width: 200px;
  height: 200px;
  background: #ee3e64;
}

#box_2 {
  position: relative;
  width: 200px;
  height: 200px;
  background: #44accf;
}

#box_3 {
  position: relative;
```

```
  width: 200px;
  height: 200px;
  background: #b7d84b;
}
```

Example B *(/d/css-positioning-101/example_b.html)* proves that relatively positioned elements behave exactly the same way as statically positioned elements. What you may not know is that elements with a `relative` position value are like Clark Kent *(http://en.wikipedia.org/wiki/Clark_Kent)*—they hide far greater powers than their static siblings.

For starters, we can adjust a relatively positioned element with offset properties: `top`, `right`, `bottom`, and `left`. Using the markup from our previous example, let's add an offset position to `#box_2`:

```
#box_2 {
  position: relative;
  left: 200px;
  width: 200px;
  height: 200px;
  background: #44accf;
}
```

Example C *(/d/css-positioning-101/example_c.html)* shows this `relative` positioning in action. Our three blocks are stacked up nicely, but this time the blue block (`#box_2`) is pushed out 200 pixels from the left. This is where we start to bend the law of gravity to our will. The blue block is still in the flow of the document—elements are stacking one on top of the other—but notice the green block (`#box_3`) on the bottom. It's sitting underneath the blue block, even though the blue block isn't directly above it. When you use the offset property to shift a relatively positioned element, it doesn't affect the element(s) that follow. The green box is still positioned as if the blue box were in its non-offset position. Try that with your alphabet block tower.

Creating a coordinate system for child elements is another one of the relative positioning property's super powers. A coordinate system is a reference point for offset properties. Recall in example C *(/d/css-positioning-101/example_c.html)*, our blue block (`#box_2`) is not sitting inside of any other elements, so the coordinate system it's using to offset itself 200 pixels from the left is the document itself. If we place the `#box_2` element inside of `#box_1`, we'll get a different result, as `#box_2` will position itself *relative* to the coordinate system from `#box_1`. For the next example, we won't change any CSS, we'll adjust our HTML to move `#box_2` inside of `#box_1`:

```
<div id="box_1">
  <div id="box_2"></div>
</div>
```

Example D *(/d/css-positioning-101/example_d.html)* shows our new markup. Because of the new coordinate system, the blue block measures its offset 200 pixels from the left of the red block (`#box_1`) instead of the document. This practice is more common with elements set to `position: absolute` —the way you wish you could have built towers when you were younger.

Absolute—anywhere, anytime

If the `relative` value acts like Superman *(http://en.wikipedia.org/wiki/Superman)*, then the `absolute` value mirrors *Inception (http://en.wikipedia.org/wiki/Inception_(film))*—a place where you design your own world. Unlike the `static` and `relative` values, an absolutely positioned element is removed from the normal flow. This means you can put it anywhere, and it won't *affect or be affected* by any other element in the flow. Think of it as an element with a giant strip of velcro on its back. Just tell it where to stick and it sticks. Exactly like the `relative` value, absolutely positioned elements respond to offset properties for positioning. You can set an element to `top: 100px` and `left: 200px;` and that element will sit exactly 100px from the top and 200px from the left of the document. Let's look at an example using four elements:

```
#box_1 {
  position: absolute;
  top: 0;
  left: 0;
  width: 200px;
  height: 200px;
  background: #ee3e64;
}
#box_2 {
  position: absolute;
  top: 0;
  right: 0;
  width: 200px;
  height: 200px;
  background: #44accf;
}
#box_3 {
  position: absolute;
  bottom: 0;
  left: 0;
  width: 200px;
  height: 200px;
  background: #b7d84b;
}
#box_4 {
```

```
  position: absolute;
  bottom: 0;
  right: 0;
  width: 200px;
  height: 200px;
  background: #ebde52;
}
```

Example E *(/d/css-positioning-101/example_e.html)* shows four boxes, each in a corner of the browser window. Since we set each box's `position` value to `absolute`, we've essentially velcroed a box to each corner of our browser window. As you resize the browser, those boxes will stay in their respective corners. If you shrink the browser window so that the boxes overlap, you'll notice that there is no interaction at all—that's because they're out of the document's normal flow.

Just like `relative` elements, `absolute` elements create a new coordinate system for child elements. Example F *(/d/css-positioning-101/example_f.html)* extends Example E *(/d/css-positioning-101/example_e.html)*, with an orange element set inside each box. Notice the offset coordinates are in respect to each parent element.

Not to be outdone by other `position` property values, the `absolute` value offers some really cool functionality using the offset property. Use two or all four `offset` properties, and you can stretch an element without defining any width or height—it's bound only by its parent element or the document itself. Let's see it in action. Consider the following code:

```
#box_a {
  position: absolute;
  top: 10px;
  right: 10px;
  bottom: 10px;
  left: 10px;
  background: red;
}
#box_b {
  position: absolute;
  top: 20px;
  right: 20px;
  bottom: 20px;
  left: 20px;
  background: white;
}
```

In example G *(/d/css-positioning-101/example_g.html)* we've created a border offset 10 pixels by the document, and it's entirely fluid as the document resize—all with `absolute` positioning and offsets. In another example, we can create a two-column layout that fills the entire height of the document. Here is the CSS:

```
#box_1 {
  position: absolute;
  top: 0;
  right: 20%;
  bottom: 0;
  left: 0;
  background: #ee3e64;
}

#box_2 {
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 80%;
  background: #b7d84b;
}
```

Example H *(/d/css-positioning-101/example_h.html)* shows a full-screen, two-column layout. While this likely isn't the best approach to a two-column layout, it still shows the power the `absolute` value holds. Using some creative thinking you can find several useful applications for `position: absolute`. Similar tricks use only a single offset value. For example:

```
#box_1 {
  width: 200px;
  height: 200px;
  background: #ee3e64;
}
#box_2 {
  position: absolute;
  left: 100px;
  width: 200px;
  height: 200px;
  background: #44accf;
}
```

In example H2 *(/d/css-positioning-101/example_h2.html)*, focus on the blue block (`#box_2`). Notice how I use only one offset, `left: 100px;` . This allows the `#box_2` element to maintain its top edge and still shift 100 pixels to the left. If we applied a second offset to the top, we would see that our blue block (`#box_2`) is pulled to the top of the document. See that here, in example H3 *(/d/css-positioning-101/example_h3.html)*:

```
#box_2 {
  position: absolute;
  top: 0;
  left: 100px;
  width: 200px;
  height: 200px;
  background: #44accf;
}
```

Fixed—always there

An element with `position: fixed` shares all the rules of an absolutely positioned element, except that the viewport (browser/device window) positions the `fixed` element, as opposed to any parent element. Additionally, a `fixed` element does not scroll with the document. It stays, well…fixed. Let's look at an example:

```
#box_2 {
  position: fixed;
  bottom: 0;
  left: 0;
  right: 0;
}
```

Example I *(/d/css-positioning-101/example_i.html)* shows a footer with some copyright text in it as a `fixed` element. As you scroll, notice that it doesn't move. Notice that the `left` and `right` offset properties are set to zero. Since the `fixed` value behaves similar to the `absolute` value, we can stretch the width of the element to fit the viewport while fixing the element to the bottom using `bottom: 0;` . Use caution with the `fixed` value: Support in older browsers is spotty at best. For example, older versions of Internet Explorer render `fixed` elements as `static` elements. And, you now know that `static` elements don't behave like `fixed` elements, right? If you do plan to use `fixed` elements in a layout, there are several solutions that can help make your element behave properly in browsers that don't support the `fixed` value.

Inherit—Something for nothing

As I mentioned at the beginning of this article, there are five values available to the `position` property. The fifth value is `inherit`. It works as the name implies: The element inherits the value of its parent element. Typically, `position` property elements do not naturally inherit their parent's values—the `static` value is assigned if no `position` value is given. Ultimately, you can type `inherit` or the parent element's value and get the same result.

IN ACTION

All this talk and no action. Let's take a look at a real-world example website that uses all the position values. Example J *(/d/css-positioning-101/example_j.html)* shows a typical website layout with a header, navigation, content, and footer. Let's walk through each element, discuss its `position` property, and why we chose that property.

Let's start with our `#container` element. This is simply the containing element that I'm using to center our content within the viewport. The `#nav` element is the first element within our `#container` element. No `position` property is assigned to the `#nav` element, so by default, it's set to `static`. This is fine: We don't need to do anything to offset this element, or create any new coordinate systems with it. We will need to do that with `#content` on our next element, so for that element, we've applied a `position` property of `relative`.

Since we're not using any offsets here, the `position` value has no real influence on the `#content` element, but we placed it there to create a new coordinate system for the `#callout` element. Our `#callout` element is set to `position: absolute`, and since its parent element, `#content` is set to `relative`, the offset properties we're using on `#callout` are based off the coordinates created by `#content`. The `#callout` element uses a negative 80px pixel offset on the right to pull the element outside of its containing parent element. Additionally, I've used one of the cooler features of the `absolute` value on our `#callout` element: by setting the top and bottom offsets to 100px, I've stretched the `#callout` element to the full height of the document minus the 100px offset on top and bottom.

Since the `#callout` element has an `absolute` value, it does not affect other elements. Therefore, we need to add some padding to the `#content` element to keep our paragraphs from disappearing beneath it. Setting the padding on the right of `#content` to 250px keeps our content in full view for our users. To bring up the rear, we've added a footer with a `fixed` position to keep it fixed to the bottom of the page. As we scroll, our footer stays in place. Just as we added padding to the `#content` to keep our paragraphs from disappearing under it, we need to do the same for the `#footer` element as it is a sibling of the `absolute` value. Adding 60px to the `#content` element's bottom padding ensures that we can scroll the entire document and not miss any text that would normally be hidden under the `#footer` element.

Now we have a nice, simple layout with navigation, some content, a callout area, and a footer using `static`, `relative`, `absolute`, and `fixed` elements. Since we're using the `fixed` value in this layout, we should apply some techniques to make sure that older browsers still respect our design. [Note: There used to be a link to sample techniques, but that site has since been taken over by malware. Apologies. —*Ed.*]

Conclusion

With the `position` property's power, you can accomplish many a layout with confidence and success. Thankfully, 80% of the `position` property values have excellent support in both modern and older browsers. The `fixed` value is the one that you should watch out for. Understanding the core of these `property` values gives you a solid CSS-based layout foundation, limited only by your imagination. Hopefully, your days of guessing position values in a last-minute bug fix frenzy are now over.

About the Author

Noah Stokes

Noah Stokes is a web designer, developer and partner/co-founder of Bold, a web studio. He is a husband and father of two young boys, guitar player and music lover. You can find him on the Twitter (@motherfuton) or catch him on his blog Es Bueno.

MORE FROM THIS AUTHOR

CSS Floats 101 *(/article/css-floats-101)*