THE MAGIC OF

CHAPTER 2

Layout

In the previous chapter, we learned that each element in the page is a rectangular box. In this chapter, we will see how multiple boxes get laid out on a page.

Block, inline, and inline-block

With respect to layout, the `display` property has three values you should be most concerned with. Here are the main differences between how these three display types get laid out:

`block`

: My width is sized by my parent and I can have widths and heights set on me. My height is determined by my content.

`inline`

My width and height are determined by *my contents* and widths and heights don't do anything to me. Think of me like a word flowing in a paragraph.

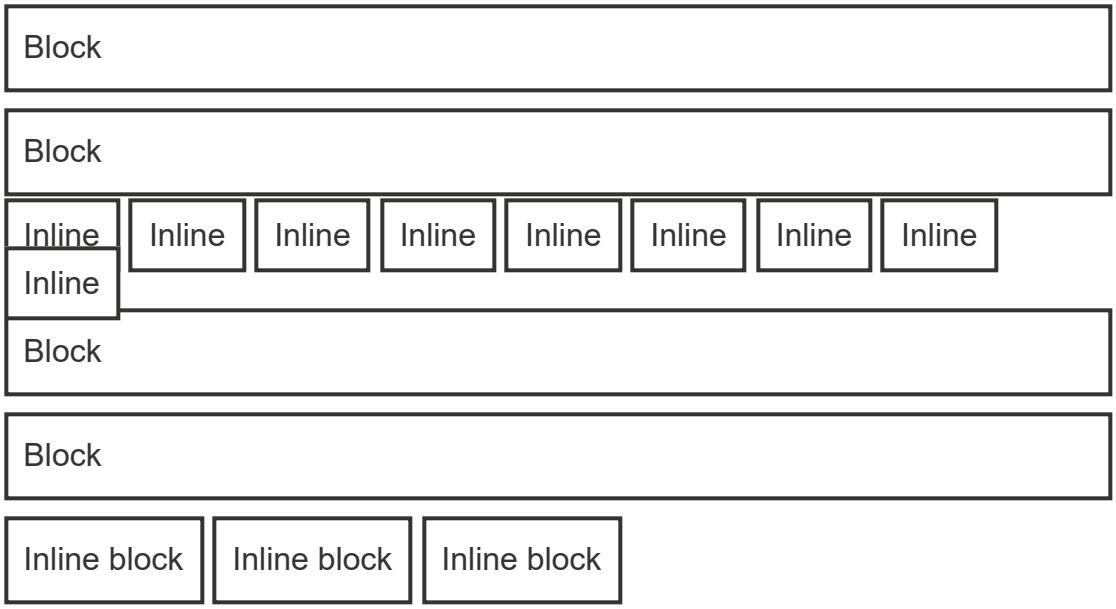
`inline-block`

I am the same as `block` except my width is determined by my contents.

Example

Below we have elements of each of these three display types with the following additional CSS applied to all of them:

```
* {
    background: #eee;
    border: .125em solid;
    margin-bottom: .5em;
    padding: .5em
}
```



☐ Set all widths to 20%

One thing to note here is the difference between `inline` and `inline-block`. The `inline` elements display their `.5em` padding and `.125em` border, but only the lefts and rights (and not tops and bottoms) of these actually affect their layout. Whereas the `inline-block` elements reposition themselves in layout due to their padding and border, just as do the `block` elements.

Also note that when setting the width to `20%` on all elements, the `block` elements *still don't wrap*. Assuming no floats are in the mix, `block` elements

do not allow horizontal neighbors.

MAGIC

Horizontal scrolling with inline-block

Horizontal scrolling sections can be tricky. Fortunately, this is a place where `inline-block` can help out.

Let's say I want to display some code with a background color applied to each row of text on hover:

```
body {
    background: red /* I know it's weird to make the whole page red, bu
}
```

If you scroll to the right, you'll see that the row hover color doesn't extend all of the way to the right. This is because each row wrapper is a block element, sizing itself to the width of its parent, not to the scroll width of its parent.

By adding an `inline-block` element which wraps all of the rows, we get the scroll we want, and the row elements (still `display: block`) can fill the width of *that* element, which is the same as the scroll width of the whole code block, because it is sized by *its contents*—in this case, the longest row.

```
body {
    background: red /* I know it's weird to make the whole page red, bu
}
```

HTML and body

The `html` and `body` elements are rectangles, just like any other elements on the page. We'll cover them more in depth in a later chapter—but for now, just know that they're both `block` elements.

Tables

Tables are crazy, and Chapter 3 covers them in more detail. But with respect to layout, think of a `table` like an `inline-block` element with one special property: its `table-cell` children can center their contents vertically.

Aside from the relatively new and experimental `display` property `flex` (which may be covered in a later chapter), no other element can do this.

So with respect to layout, think of tables as a tool which can be used to center *arbitrary content* vertically.

M A G I C

Vertical centering content with unknown height

Vertical center centering with a table couldn't be simpler.

But if you're using a table for this purpose (and not to display tabular-data), you should instead use another type of element (`div` , for example), and set its display property to `table` to mimick the table behavior.

```
<style>
    .vertical-outer {
        display: table;
        height: 10em
    }

    .vertical-inner {
        display: table-cell;
        vertical-align: middle
    }
</style>

<div class="vertical-outer">
    <div class="vertical-inner">
        <p>I'm so centered it's not even funny.</p>
    </div>
</div>
```

I'm so centered it's not even funny.

And that's it!

As an aside, centering something vertically when you know its height is trivial. First position the element, then set a `top` and `bottom` to the same value (`0` works), set your desired `height` , and then set `margin-top` and `margin-bottom` to `auto` .

Text align

Basically, text-align lets you align text, child inline elements, and child inline block elements to the left, right, center, or justified. (You know what these mean if you've ever used a WYSIWYG editor.)

Now for some magic:

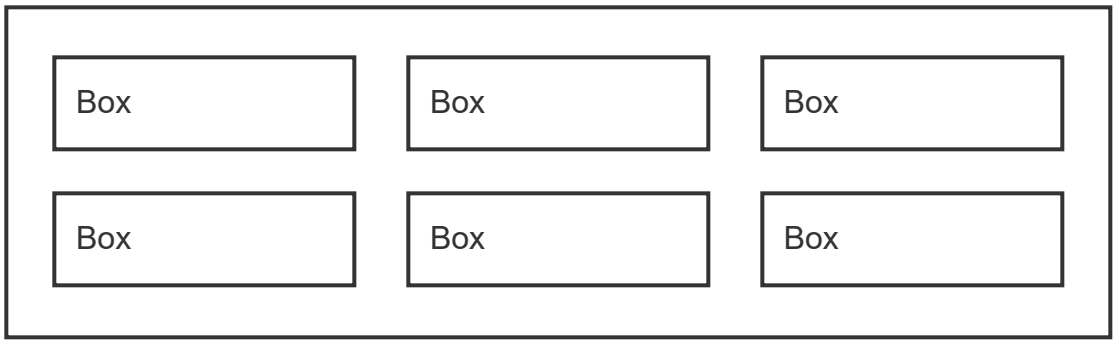
MAGIC

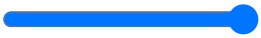
Grid with text-align justify

Since `inline-block` elements are treated more or less as text, you can use `text-align: justify` on a list of `inline-block` elements to create a grid structure.

```
<style>
    .grid {
        border: .125rem solid;
        text-align: justify;
        font-size: 0;
        padding: 4% 4% 0 4%
    }

    .box {
        font-size: 1rem;
        display: inline-block;
        background: #eee;
        border: .125em solid;
        width: 30%;
        padding: 2%
    }

    /* All but the last 3 boxes */
    .box:nth-last-child(n+5) {
        margin-bottom: 4%
    }

    .break {
```

```
        display: inline-block;
        width: 30%;
        height: 0
    }
</style>

<div class="grid">
    <div class="box">Column</div>
    <div class="box">Column</div>
    <div class="box">Column</div>
    <div class="box">Column</div>
    <div class="box">Column</div>
    <div class="box">Column</div>
    <div class="break"></div>
</div>
```





Adjust the width of the boxes and note that `text-align: justify` keeps the grid intact.

Floats

Floats are crazy, so crazy that they'll also get their own chapter.

But when it comes to positioning, basically what you need to know is that floated elements behave kind of like `inline-block` elements, regardless of what their `display` property value actually is.

The truth is, these days, since `inline-block` is supported pretty widely, there's not as much use for `float` anymore. We'll still cover it since it's a card up your sleeve, and you should know how to whip it out. But don't worry about it too much just yet.

In the meantime, you can read up on the relationship between display, position, and float in the w3 CSS2 spec.[1]

Positioning

Ahh, this is where the real fun begins.

An element is said to be “positioned” if its `position` property is any value except `static`.

When an element is positioned, it is laid out according to whichever positioning properties `top`, `bottom`, `left`, and `right` it has set.

This means not only do these properties reposition (or move) elements, they also can resize elements. For example, with `position absolute` or `fixed`, you can set both a `top` and `bottom` to essentially impose a fixed height on the element. The precedence here can get pretty complicated, but as a general rule, if you set `top`, `bottom`, *and* `height` for a positioned element, the `height` value will be ignored.

The `position` property can take on the following values:

`static`
: *The default*. Any `top`, `right`, `bottom`, or `left` properties are ignored.

`absolute`
: The element will be removed from its original layout position and positioned relative to its nearest positioned parent by the positioning properties.

`fixed`
: The element will be removed from its original layout position and positioned relative to the window. (Mobile devices with zoom may have indeterminate behavior.)

`relative`
: Unlike absolute or fixed, the element stays in its original layout position and the `top`, `right`, `bottom`, or `left` properties only *nudge* it from that original position.

This stuff can be confusing, so we’ll highlight some important takeaways from these descriptions:

- `absolute` and `fixed` elements are not part of normal document layout. When their dimensions change, only their child elements are affected. (There is a subtle exception to this which is that `absolute` positioned elements can cause a scroll bar [in the positive content flow direction: by default, to the right or down] and this can affect the layout of other elements in the page.)
- `static` and `relative` elements are part of the layout. When their layout changes, so do their document neighbors.
- When *nudged* via `top`, `right`, `bottom`, or `left`, `relative` elements do not affect their document neighbors. Instead, those neighbors act like the element was never *nudged* from its original position. (The scroll exception applies here as well.)
- Confusingly, `relative` is not so-named because its child elements will be positioned “relative” to it. (That is simply a consequence of it being positioned at all, and so the same could be said about `absolute` and `fixed` elements as well.) Rather, it is so-named to describe how you can “relatively” *nudge* it based on its original position.

Now again, for a little magic:

MAGIC

100% `top`, `bottom`, `left`, or `right`

Positioning a child element abbutted to the outside of its parent is a bit tricky.

The naive approach is to use a negative positioning property which matches its dimension.

For example:

```
.parent {
    position: relative;
    text-align: center;
    padding: 1.25em;
    background: #eee
}

.child {
    position: absolute;
    height: 2.5em;
    top: -2.5em;
    right: 0;
```

```
    line-height: 2.5em;
    background: #444;
    color: #fff;
    padding: 0 .625em
}
```

Child

Parent

Note the following two lines of CSS:

```
height: 2.5em;
top: -2.5em;
```

This part is unideal because it's not DRY and because we had to specify a height. *When possible, it's better not to specify fixed values in CSS.* The more you can let things get sized by content the better, because it means your design is more flexible, supports more use-cases, and is less likely to create future bugs.

So what can we do instead? Use `100%` values.

Instead of thinking moving the child up by a negative `top`, think of it as moving `100%` from the bottom. Now our same example can be written like this instead:

```
.parent {
    position: relative;
    text-align: center;
    padding: 1.25em;
    background: #eee
}

.child {
    position: absolute;
    bottom: 100%;
    right: 0;
    background: #444;
    color: #fff;
    padding: .625em
}
```

Parent

Notice how in this version we were able to simplify the padding and line-height because the child box is now sizing itself to its contents, rather than the other way around.

Transforms

These will definitely get their own chapter. Transforms are where a lot of the real magic lives. But for now, note that unfortunately transformed elements are treated as positioned even if they are statically-positioned[2]. Commit this to memory or at some point it will probably burn you.

Further reading

- Quirksmode: display — has a great toy for playing with the different property values.
- MDN: Layout
- MDN: position

Citations

1. Relationships between display, position, and float
2. Un-fixing Fixed Elements with CSS Transforms

Chapter 1 Chapter 3