Learn to Code
HTML & CSS

Lesson 5

Positioning Content

In this Lesson

5

CSS

SHARE

One of the best things about CSS is that it gives us the ability to position content and elements on a page in nearly any imaginable way, bringing structure to our designs and helping make content more digestible.

There are a few different types of positioning within CSS, and each has its own application. In this chapter we're going to take a look at a few different use cases—creating reusable layouts and uniquely positioning one-off elements—and describe a few ways to go about each.

Positioning with Floats

One way to position elements on a page is with the `float` property. The `float` property is pretty versatile and can be used in a number of different ways.

Essentially, the `float` property allows us to take an element, remove it from the normal flow of a page, and position it to the left or right of its parent element. All other elements on the page will then flow around the floated element. An `<img>` element floated to the side of a few paragraphs of text, for example, will allow the paragraphs to wrap around the image as necessary.

When the `float` property is used on multiple elements at the same time, it provides the ability to create a layout by floating elements directly next to or opposite each other, as seen in multiple-column layouts.

The `float` property accepts a few values; the two most popular values are `left` and `right`, which allow elements to be floated to the left or right of their parent element.

```
1    img {
2      float: left;
3    }
```

Floats in Practice

Let's create a common page layout with a header at the top, two columns in the center, and a footer at the bottom. Ideally this page would be marked up using the `<header>`, `<section>`, `<aside>`, and `<footer>` elements as discussed in Lesson 2, "Getting to Know HTML." Inside the `<body>` element, the HTML may look like this:

```
1    <header>...</header>
2    <section>...</section>
3    <aside>...</aside>
4    <footer>...</footer>
```

Layout without Floats Demo

| HTML | CSS | **RESULT** | ⬨ |

Here the `<section>` and `<aside>` elements, as block-level elements, will be stacked on top of one another by default. However, we want these elements to sit side by side. By floating the `<section>` to the `left` and the `<aside>` to the `right`, we can position them as two columns sitting opposite one another. Our CSS should look like this:

```
1       section {
2         float: left;
3       }
4       aside {
5         float: right;
6       }
```

For reference, when an element is floated, it will float all the way to the edge of its parent element. If there isn't a parent element, the floated element will then float all the way to the edge of the page.

When we float an element, we take it out of the normal flow of the HTML document. This causes the width of that element to default to the width of the content within it. Sometimes, such as when we're creating columns for a reusable layout, this behavior is not desired. It can be corrected by adding a fixed `width` property value to each column. Additionally, to prevent floated elements from touching one another, causing the content of one to sit directly next to the content of the other, we can use the `margin` property to create space between elements.

Here, we are extending the previous code block, adding a `margin` and `width` to each column to better shape our desired outcome.

```
1       section {
2         float: left;
3         margin: 0 1.5%;
4         width: 63%;
5       }
6       aside {
7         float: right;
8         margin: 0 1.5%;
9         width: 30%;
10      }
```

Layout with Floats Demo

HTML	**CSS**	**RESULT**	

Floats May Change an Element's Display Value

When floating an element, it is also important to recognize that an element is removed from the normal flow of a page, and that may change an element's default `display` value. The `float` property relies on an element having a `display` value of `block`, and may alter an element's default `display` value if it is not already displayed as a block-level element.

For example, an element with a `display` value of `inline`, such as the `<span>` inline-level element, ignores any `height` or `width` property values. However, should that inline-level element be floated, its `display` value will be changed to block, and it may then accept `height` or `width` property values.

As we float elements we must keep an eye on how their `display` property values are affected.

With two columns we can float one column to the left and another to the right, but with more columns we must change our approach. Say, for example, we'd like to have a row of three columns between our `<header>` and `<footer>` elements. If we drop our `<aside>` element and use three `<section>` elements, our HTML might look like this:

```
1    <header>...</header>
2    <section>...</section>
3    <section>...</section>
4    <section>...</section>
5    <footer>...</footer>
```

To position these three `<section>` elements in a three-column row, instead of floating one column to the left and one column to the right, we'll float all three `<section>` elements to the left. We'll also need to adjust the width of the `<section>` elements to account for the additional columns and to get them to sit one next to the other.

```
1    section {
2        float: left;
3        margin: 0 1.5%;
4        width: 30%;
5    }
```

Here we have three columns, all with equal width and margin values and all floated to the `left`.

Three-column Layout with Floats Demo

HTML   **CSS**   **RESULT**

Clearing & Containing Floats

The `float` property was originally designed to allow content to wrap around images. An image could be floated, and all of the content surrounding that image could then naturally flow around it. Although this works great for images, the `float` property was never actually intended to be used for layout and positioning purposes, and thus it comes with a few pitfalls.

One of those pitfalls is that occasionally the proper styles will not render on an element that it is sitting next to or is a parent element of a floated element. When an element is floated, it is taken out of the normal flow of the page, and, as a result, the styles of elements around that floated element can be negatively impacted.

Often `margin` and `padding` property values aren't interpreted correctly, causing them to blend into the floated element; other properties can be affected, too.

Another pitfall is that sometimes unwanted content begins to wrap around a floated element. Removing an element from the flow of the document allows all the elements around the floated element to wrap and consume any available space around the floated element, which is often undesired.

With our previous two-column example, after we floated the `<section>` and `<aside>` elements, and before we set a width property value on either of them, the content within the `<footer>` element would have wrapped in between the two floated elements above it, filling in any available space. Consequently, the `<footer>` element would have sat in the gutter between the `<section>` and `<aside>` elements, consuming the available space.

Layout without Cleared or Contained Floats Demo

To prevent content from wrapping around floated elements, we need to clear, or contain, those floats and return the page to its normal flow. We'll proceed by looking at how to clear floats, and then we'll take a look at how to contain floats.

Clearing Floats

Clearing floats is accomplished using the `clear` property, which accepts a few different values: the most commonly used values being `left`, `right`, and `both`.

```
1    div {
2        clear: left;
3    }
```

The left value will clear left floats, while the right value will clear right floats. The both value, however, will clear both left and right floats and is often the most ideal value.

Going back to our previous example, if we use the clear property with the value of both on the <footer> element, we are able to clear the floats. It is important that this clear be applied to an element appearing after the floated elements, not before, to return the page to its normal flow.

```
1    footer {
2      clear: both;
3    }
```

Layout with Cleared Floats Demo

Containing Floats

Rather than clearing floats, another option is to contain the floats. The outcomes of containing floats versus those of clearing them are nearly the same; however, containing floats does help to ensure that all of our styles will be rendered properly.

To contain floats, the floated elements must reside within a parent element. The parent element will act as a container, leaving the flow of the document completely normal outside of it. The CSS for that parent element, represented by the group class below, is shown here:

```
1    .group:before,
2    .group:after {
```

```
 3        content: "";
 4        display: table;
 5      }
 6    .group:after {
 7        clear: both;
 8      }
 9    .group {
10        clear: both;
11        *zoom: 1;
12      }
```

There's quite a bit going on here, but essentially what the CSS is doing is clearing any floated elements within the element with the class of `group` and returning the flow of the document back to normal.

More specifically, the `:before` and `:after` pseudo-elements, as mentioned in the Lesson 4 exercise, are dynamically generated elements above and below the element with the class of `group`. Those elements do not include any content and are displayed as `table`-level elements, much like block-level elements. The dynamically generated element after the element with the class of `group` is clearing the floats within the element with the class of `group`, much like the `clear` from before. And lastly, the element with the class of `group` itself also clears any floats that may appear above it, in case a left or right float may exist. It also includes a little trickery to get older browsers to play nicely.

It is more code than the `clear: both;` declaration alone, but it can prove to be quite useful.

Looking at our two-column page layout from before, we could wrap the `<section>` and `<aside>` elements with a parent element. That parent element then needs to contain the floats within itself. The code would look like this:

HTML

```
1      <header>...</header>
2      <div class="group">
3        <section>...</section>
4        <aside>...</aside>
5      </div>
6      <footer>...</footer>
```

CSS

```css
1       .group:before,
2       .group:after {
3         content: "";
4         display: table;
5       }
6       .group:after {
7         clear: both;
8       }
9       .group {
10        clear: both;
11        *zoom: 1;
12      }
13      section {
14        float: left;
15        margin: 0 1.5%;
16        width: 63%;
17      }
18      aside {
19        float: right;
20        margin: 0 1.5%;
21        width: 30%;
22      }
```

Layout with Contained Floats Demo

The technique shown here for containing elements is know as a "clearfix" and can often be found in other websites with the class name of `clearfix` or `cf`. We've chosen to use the class name of `group`, though, as it is representing a group of elements, and better expresses the content.

As elements are floated, it is important to keep note of how they affect the flow of a page and to make sure the flow of a page is reset by either clearing or containing the floats as necessary. Failing to keep track of floats can cause quite a few headaches, especially as pages begin to have multiple rows of multiple columns.

In Practice

Let's return to the Styles Conference website to try floating some content.

1. First things first, before we begin floating any elements, let's provide a way to contain those floats by adding the clearfix to our CSS. Within the `main.css` file, just below our grid styles, let's add the clearfix under the class name `group`, just like before.

```
1    /*
2       ======================================
3       Clearfix
4       ======================================
5    */
6    .group:before,
7    .group:after {
8      content: "";
9      display: table;
10   }
11   .group:after {
12     clear: both;
13   }
14   .group {
15     clear: both;
16     *zoom: 1;
17   }
```

2. Now that we can contain floats, let's float the primary `<h1>` within the `<header>` element to the left and allow all of the other content in the header to wrap to the right of it.

   To do this, let's add a class of `logo` to the `<h1>` element. Then within our CSS, let's add a new section of styles for the primary header. In this section we'll select the

`<h1>` element with the `logo` class and then `float` it to the left.

HTML

```
1    <h1 class="logo">
2      <a href="index.html">Styles Conference</a>
3    </h1>
```

CSS

```
1    /*
2      ======================================
3      Primary header
4      ======================================
5    */
6
7    .logo {
8      float: left;
9    }
```

3   While we're at it, let's add a little more detail to our logo. We'll begin by placing a
    `<br>` element, or line break, between the word "Styles" and the word
    "Conference" to force the text of our logo to sit on two lines.

    Within the CSS, let's add a border to the top of our logo and some vertical
    `padding` to give the logo breathing room.

HTML

```
1    <h1 class="logo">
2      <a href="index.html">Styles <br> Conference</a>
3    </h1>
```

CSS

```
1    .logo {
2      border-top: 4px solid #648880;
3      padding: 40px 0 22px 0;
4      float: left;
5    }
```

4   Because we floated the `<h1>` element, we'll want to contain that `float`. The closest parent element of the `<h1>` element is the `<header>` element, so we'll want to add the class of `group` to the `<header>` element. Doing this applies the clearfix styles we set up earlier to the `<header>` element.

```
1    <header class="container group">
2        ...
3    </header>
```

5   The `<header>` element is taking shape, so let's take a look at the `<footer>` element. Much like we did with the `<header>` element, we'll float our copyright to the left within the `<small>` element and let all other elements wrap around it to the right.

Unlike the `<header>` element, though, we're not going to use a class directly on the floated element. This time we're going to apply a class to the parent of the floated element and use a unique CSS selector to select the element and then float it.

Let's start by adding the class of `primary-footer` to the `<footer>` element. Because we know we'll be floating an element within the `<footer>` element, we should also add the class of `group` while we're at it.

```
1    <footer class="primary-footer container group">
2        ...
3    </footer>
```

6   Now that the class of `primary-footer` is on the `<footer>` element, we can use that class to prequalify the `<small>` element with CSS. We'll want to select and `float` the `<small>` element to the `left`. Let's not forget to create a new section within our `main.css` file for these primary footer styles.

```
1    /*
2    ======================================
3    Primary footer
4    ======================================
5    */
6
7    .primary-footer small {
8
```

```
9          float: left;
        }
```

To review, here we are selecting the `<small>` element, which must reside within an element with the class attribute value of `primary-footer`, such as our `<footer>` element, for example.

7 Lastly, let's put some `padding` on the top and bottom of the `<footer>` element to help separate it a little more from the rest of the page. We can do this directly by using the `primary-footer` class with a class selector.

```
1    .primary-footer {
2        padding-bottom: 44px;
3        padding-top: 44px;
4    }
```

With all of these changes to the `<header>` and `<footer>` elements, we have to be sure to make them on every page, not just the `index.html` page.
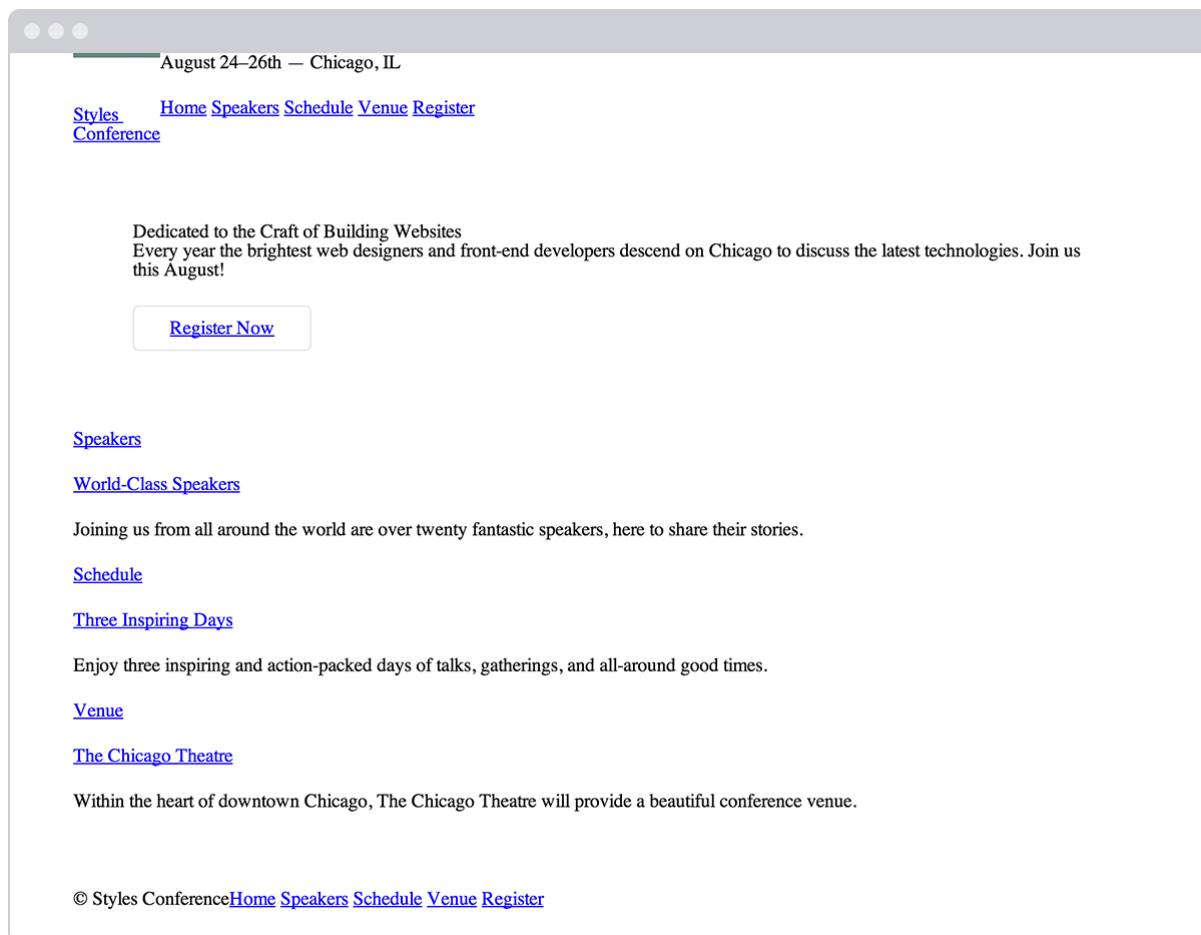

August 24–26th — Chicago, IL

Styles Conference   Home Speakers Schedule Venue Register

Dedicated to the Craft of Building Websites
Every year the brightest web designers and front-end developers descend on Chicago to discuss the latest technologies. Join us this August!

Register Now

Speakers

World-Class Speakers

Joining us from all around the world are over twenty fantastic speakers, here to share their stories.

Schedule

Three Inspiring Days

Enjoy three inspiring and action-packed days of talks, gatherings, and all-around good times.

Venue

The Chicago Theatre

Within the heart of downtown Chicago, The Chicago Theatre will provide a beautiful conference venue.

© Styles Conference Home Speakers Schedule Venue Register

Fig 5.01

With a few floats, the `<header>` and `<footer>` elements on our Styles Conference home page are coming together

Positioning with Inline-Block

In addition to using floats, another way we can position content is by using the `display` property in conjunction with the `inline-block` value. The inline-block method, as we'll discuss, is primarily helpful for laying out pages or for placing elements next to one another within a line.

Recall that the `inline-block` value for the `display` property will display elements within a line while allowing them to accept all box model properties, including `height`, `width`, `padding`, `border`, and `margin`. Using inline-block elements allows us to take full advantage of the box model without having to worry about clearing any floats.

Inline-Block in Practice

Let's take a look at our three-column example from before. We'll start by keeping our HTML just as it is:

```
1    <header>...</header>
2    <section>...</section>
3    <section>...</section>
4    <section>...</section>
5    <footer>...</footer>
```

Now instead of floating our three `<section>` elements, we'll change their `display` values to `inline-block`, leaving the `margin` and `width` properties from before alone. Our resulting CSS will look like this:

```
1    section {
2      display: inline-block;
3      margin: 0 1.5%;
4      width: 30%;
5    }
```

Unfortunately, this code alone doesn't quite do the trick, and the last `<section>` element is pushed to a new row. Remember, because inline-block elements are displayed on the same line as one another, they include a single space between them. When the size of each single space is added to the `width` and horizontal `margin` values of all the elements in the row, the total width becomes too great, pushing the last

`<section>` element to a new row. In order to display all of the `<section>` elements on the same row, the white space between each `<section>` element must be removed.

Inline-Block Elements with White Space Demo

Removing Spaces Between Inline-Block Elements

There are a number of ways to remove the space between inline-block elements, and some are more complex than others. We are going to focus on two of the easiest ways, both of which happen inside HTML.

The first solution is to put each new `<section>` element's opening tag on the same line as the previous `<section>` element's closing tag. Rather than using a new line for each element, we'll end and begin elements on the same line. Our HTML could look like this:

```
1    <header>...</header>
2    <section>
3       ...
4    </section><section>
5       ...
6    </section><section>
7       ...
8    </section>
9    <footer>...</footer>
```

Writing inline-block elements this way ensures that the space between inline-block elements within HTML doesn't exist; consequently, the space will not appear when the page is rendered.

Inline-Block Elements without White Space Demo

Another way to remove the white space between inline-block elements is to open an HTML comment directly after an inline-block element's closing tag. Then, close the HTML com- ment immediately before the next inline-block element's opening tag. Doing this allows inline-block elements to begin and end on separate lines of HTML and "comments out" any potential spaces between the elements. The resulting code would look like this:

```
1    <header>...</header>
2    <section>
3       ...
4    </section><!--
5    --><section>
6       ...
7    </section><!--
8    --><section>
9       ...
10    </section>
11    <footer>...</footer>
```

Neither of these options is perfect, but they are helpful. I tend to favor using comments for better organization, but which option you choose is entirely up to you.

Creating Reusable Layouts

When building a website, it is always best to write modular styles that may be reused elsewhere, and reusable layouts are high on the list of reusable code. Layouts can be

created using either floats or inline-block elements, but which works best and why?

Whether it's better to use floats or inline-block elements to lay out the structure of a page is open to debate. My approach is to use inline-block elements to create the grid— or layout—of a page and to then use floats when I want content to wrap around a given element (as floats were intended to do with images). Generally, I also find inline-block elements easier to work with.

That said, use whatever works best for you. If you are comfortable with one approach over the other, then go for it.

Currently there are new CSS specifications in the works—specifically `flex-` and `grid-` based properties—that will help address how to best lay out pages. Keep an eye out for these methods as they begin to surface.

In Practice

With a solid understanding of reusable layouts, the time has come to implement one in our Styles Conference website.

1. For the Styles Conference website, we'll create a three-column reusable layout using inline-block elements. We'll do so in a way that allows us to have three columns of equal width or two columns with the total width split between them, two-thirds in one and one-third in the other.

    To begin, we'll create classes that define the `width` of these columns. The two classes we'll create are `col-1-3`, for one-third, and `col-2-3`, for two-thirds. Within the grid section of our `main.css` file, let's go ahead and define these classes and their corresponding widths.

```
1    .col-1-3 {
2      width: 33.33%;
3    }
4    .col-2-3 {
5      width: 66.66%;
6    }
```

2. We'll want both of the columns to be displayed as inline-block elements. We'll need to make sure that their vertical alignment is set to the `top` of each column, too.

    Let's create two new selectors that will share the display and vertical-alignment property styles.

```
1    .col-1-3,
2    .col-2-3 {
3      display: inline-block;
4      vertical-align: top;
5    }
```

Looking at the CSS again, we've created two class selectors, `col-1-3` and `col-2-3`, that are separated with a comma. The comma at the end of the first selector signifies that another selector is to follow. The second selector is followed by the opening curly bracket, {, which signifies that style declarations are to follow. By comma-separating the selectors, we can bind the same styles to multiple selectors at one time.

3   We'll want to put some space in between each of the columns to help break up the content. We can accomplish this by putting horizontal `padding` on each of the columns.

This works well; however, when two columns are sitting next to one another, the width of the space between them will be double that of the space from the outside columns to the edge of the row. To balance this we'll place all of our columns within a grid and add the same padding from our columns to that grid.

Let's use a class name of `grid` to identify our grid, and then let's identify the same horizontal `padding` for our grid, `col-1-3`, and `col-2-3` classes. With commas separating our selectors again, our CSS looks like this:

```
1    .grid,
2    .col-1-3,
3    .col-2-3 {
4      padding-left: 15px;
5      padding-right: 15px;
6    }
```

4   When we're setting up the horizontal `padding`, we'll need to be careful. Remember, in the last lesson we created a container element, known by the class of `container`, to center all of our content on a page within a 960-pixel-wide element. Currently if we were to put an element with the class of `grid` inside an element with the class of `container`, their horizontal paddings would add to one another, and our columns would not appear proportionate to the width of the rest of the page.

We don't want this to happen, so instead, we'll have to share some of the styles from the `container` rule set with the `grid` rule set. Specifically, we'll need to

share the `width` property and values (to make sure our page stays fixed at `960` pixels wide) and the `margin` property and values (to center any element with the class of grid on the page).

We'll accomplish this by breaking up the old `container` rule set into the following:

```
1    .container,
2    .grid {
3      margin: 0 auto;
4      width: 960px;
5    }
6    .container {
7      padding-left: 30px;
8      padding-right: 30px;
9    }
```

Now any element with the class of `container` or grid will be `960` pixels wide and centered on the page. Additionally, we've preserved the existing horizontal padding for any element with the class of `container` by moving it into a new, separate rule set.

5　All right—all of the heavy lifting needed to get our reusable grid styles into place is finished. Now it's time to work in our HTML and to see how these classes perform.

We'll begin with the teasers on the home page, within our `index.html` file, aligning them into three columns. Currently, the teasers are wrapped in a `<section>` element with the class of `container`. We'll want to change that class from `container` to `grid` so that we can begin placing columns within it.

```
1    <section class="grid">
2      ...
3    </section>
```

6　Next, we'll want to add a class of `col-1-3` to each of the `<section>` elements within the `<section>` element with the class of `grid`.

```
1    <section class="grid">
2
3      <section class="col-1-3">
4        ...
5      </section>
```

```
 6
 7        <section class="col-1-3">
 8           ...
 9        </section>
10
11        <section class="col-1-3">
12           ...
13        </section>
14
15     </section>
```

7   And lastly, because each of our columns is an inline-block element, we'll want to
     make sure we remove the empty white space between them. We'll use comments
     to do this, and we'll add a little bit of documentation noting each upcoming
     section while we're at it to better organize our code.

```
 1     <section class="grid">
 2
 3        <!-- Speakers -->
 4
 5        <section class="col-1-3">
 6           ...
 7        </section><!--
 8
 9        Schedule
10
11        --><section class="col-1-3">
12           ...
13        </section><!--
14
15        Venue
16
17        --><section class="col-1-3">
18           ...
19        </section>
20
21     </section>
```

To review, on line 3 we leave a comment identifying the "Speakers" section to
follow. At the end of line 7, we open a comment immediately after the closing

`</section>` tag. Within that comment, on line 9 we identify the "Schedule" section to come. We then close the comment at the beginning of line 11, just before the opening `<section>` tag. This same comment structure reappears on lines 13 through 17 between the two `<section>` elements, right before the "Venue" section. In all, we've commented out any potential white space between the columns while also using those comments to identify our sections.

We now have a reusable three-column grid that supports multiple arrangements, using both one-third- and two-thirds-width columns. Our home page now has three columns, breaking up all the different teasers.
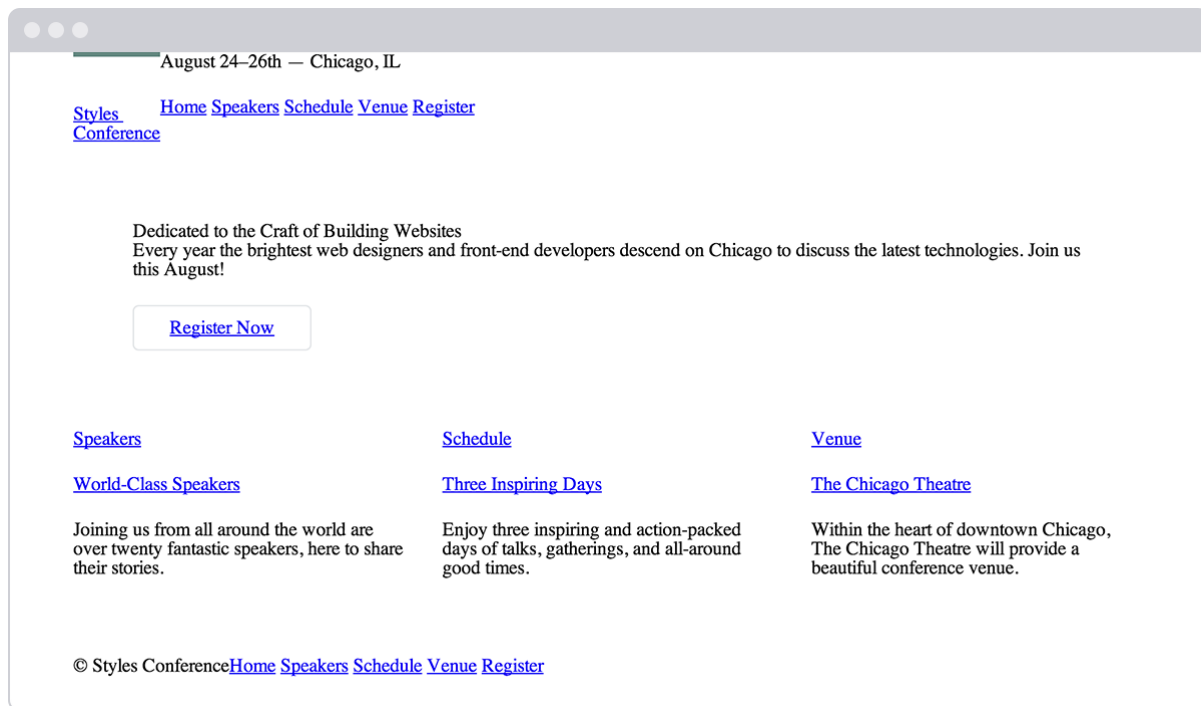

Fig 5.02

Our Styles Conference home page now includes a three-column layout

Demo & Source Code

Below you may view the Styles Conference website in its current state, as well as download the source code for the website in its current state.

View the Styles Conference Website   or [Download the Source Code](#) (Zip file)

Uniquely Positioning Elements

Every now and then we'll want to [precisely position](#) an element, but floats or inline-block elements won't do the trick. Floats, which remove an element from the flow of a page, often produce unwanted results as surrounding elements flow around the floated element. Inline-block elements, unless we're creating columns, can be fairly awkward to get into the proper position. For these situations we can use the `position` property in connection with box offset properties.

The `position` property identifies *how* an element is positioned on a page and whether or not it will appear within the normal flow of a document. This is used in conjunction with the box offset properties—`top`, `right`, `bottom`, and `left`—which identify exactly *where* an element will be positioned by moving elements in a number of different directions.

By default every element has a `position` value of `static`, which means that it exists in the normal flow of a document and it doesn't accept any box offset properties. The `static` value is most commonly overwritten with a `relative` or `absolute` value, which we'll examine next.

Relative Positioning

The `relative` value for the `position` property allows elements to appear within the normal flow a page, leaving space for an element as intended while not allowing other elements to flow around it; however, it also allows an element's display position to be modified with the box offset properties. For example, consider the following HTML and CSS:

HTML

```
1    <div>...</div>
2    <div class="offset">...</div>
3    <div>...</div>
```

CSS

```
1    div {
2      height: 100px;
3      width: 100px;
4    }
5    .offset {
6      left: 20px;
7      position: relative;
8      top: 20px;
9    }
```

Relative Positioning Demo

Here the second `<div>` element, the element with the class of `offset`, has a `position` value of `relative` and two box offset properties, `left` and `top`. This preserves the original position of the element, and other elements are not allowed to move into this space. Additionally, the box offset properties reposition the element, pushing it `20` pixels from the `left` and `20` pixels from the `top` of its original location.

With relatively positioned elements, it's important to know that the box offset properties identify where an element will be moved from given its original position. Thus, the `left` property with a value of `20` pixels will actually push the element towards the right, from the left, `20` pixels. The `top` property with a value of `20` pixels, then, will push an element towards the bottom, from the top, `20` pixels.

When we position the element using the box offset properties, the element overlaps the element below it rather than moving that element down as the `margin` or `padding` properties would.

Absolute Positioning

The `absolute` value for the `position` property is different from the `relative` value in that an element with a `position` value of `absolute` will not appear within the normal flow of a document, and the original space and position of the absolutely positioned element will not be preserved.

Additionally, absolutely positioned elements are moved in relation to their closest relatively positioned parent element. Should a relatively positioned parent element not exist, the absolutely positioned element will be positioned in relation to the `<body>` element. That's quite a bit of information; let's take a look at how this works inside some code:

HTML

```
1      <section>
2        <div class="offset">...</div>
3      </section>
```

CSS

```
1      section {
2        position: relative;
3      }
4      div {
5        position: absolute;
6        right: 20px;
7        top: 20px;
8      }
```

Absolute Positioning Demo

In this example the `<section>` element is relatively positioned but doesn't include any box offset properties. Consequently its position doesn't change. The `<div>` element with a class of `offset` includes a `position` value of `absolute`. Because the `<section>` element is the closest relatively positioned parent element to the `<div>` element, the `<div>` element will be positioned in relation to the `<section>` element.

With relatively positioned elements, the box offset properties identify in which direction an element would be moved in relation to itself. With absolutely positioned elements, the box offset properties identify in which direction an element will be moved in relation to its closest relatively positioned parent element.

As a result of the `right` and `top` box offset properties, the `<div>` element will appear `20` pixels from the `right` and `20` pixels from the `top` of the `<section>`.

Because the `<div>` element is absolutely positioned, it does not sit within the normal flow of the page and will overlap any surrounding elements. Additionally, the original position of the `<div>` is not preserved, and other elements are able to occupy that space.

Typically, most positioning can be handled without the use of the `position` property and box offset properties, but in certain cases they can be extremely helpful.

Summary

Learning how to position content within HTML and CSS is a huge step toward mastering the two languages. Add to this the box model, and we're well on our way to becoming front-end developers.

To review, within this lesson we covered the following:

- What floats are and how to use them to position content
- How to clear and contain floated elements
- How to position content with inline-block elements
- How to remove the white space between inline-block elements
- How to uniquely position content with relatively and absolutely positioned elements

We're adding new skills with each lesson, so let's keep going. Next up, typography!

Resources & Links

- CSS Float Theory via Smashing Magazine
- CSS Positioning 101 via A List Apart

Lesson 4
Opening the Box Model

Lesson 6
Working with Typography

Learn More HTML & CSS or Study Other Topics

Learning how to code HTML & CSS and building successful websites can be challenging, and at times additional help and explanation can go a long way. Fortunately there are plenty of online schools, boot camps, workshops, and the alike, that can help.

Select your topic of interest below and I will recommend a course I believe will provide the best learning opportunity for you.

Select Your Topic of Interest:

Design & Product

Front-end Development

Web Development

Mobile Development

Data & Machine Learning

Info & Cyber Security

Your Course Recommendations:

Select a topic above to view your course recommendations.

Learn to Code HTML & CSS the Book

Learn to Code HTML & CSS is an interactive beginner's guide with one express goal: **teach you how to develop and style websites** with HTML and CSS.

Outlining the fundamentals, this book covers all of the common elements of front-end design and development.

Buy Learn to Code HTML & CSS

Also available at Amazon and Barnes & Noble

Looking for Advanced HTML & CSS Lessons?

Checkout these advanced HTML and CSS lessons to take a deeper look at front-end design and development, perfect for any designer or front-end developer looking to round out their skills.

View Advanced HTML & CSS Lessons

Join the Newsletter

To stay up to date and learn when new courses and lessons are posted, please sign up for the newsletter—spam free.

Email address   Get Updates

© Shay Howe   Enjoy these lessons?   Follow @shayhowe