



Creating Page Layouts with Flexbox

“*Flexbox is the first CSS layout technique that works for the modern web.* —Paddi MacDonnell



This chapter covers

- Understanding how flexbox works
- Learning the techniques for working with flexbox containers and items
- Putting flexbox to good use with real-world ideas
- Building the holy grail layout with flexbox

In Chapter 11, you saw that floats and inline blocks can get the job done, but not without running into problems, quirks, and workarounds such as clearing floats, creating faux columns, and avoiding whitespace. Even with all that, these layout strategies can't accomplish one of the features of the holy-grail layout: displaying the footer at the bottom of the screen if the page content doesn't fill the screen height.

This chapter's layout strategy prevents all these quirks, solves the footer problem, and has the fresh-faced appeal of a modern technology. I'm talking about flexbox, and before you can start using it for layout, you need to understand what it is and how it works. The next few sections explain everything you need to know.



Understanding Flexbox

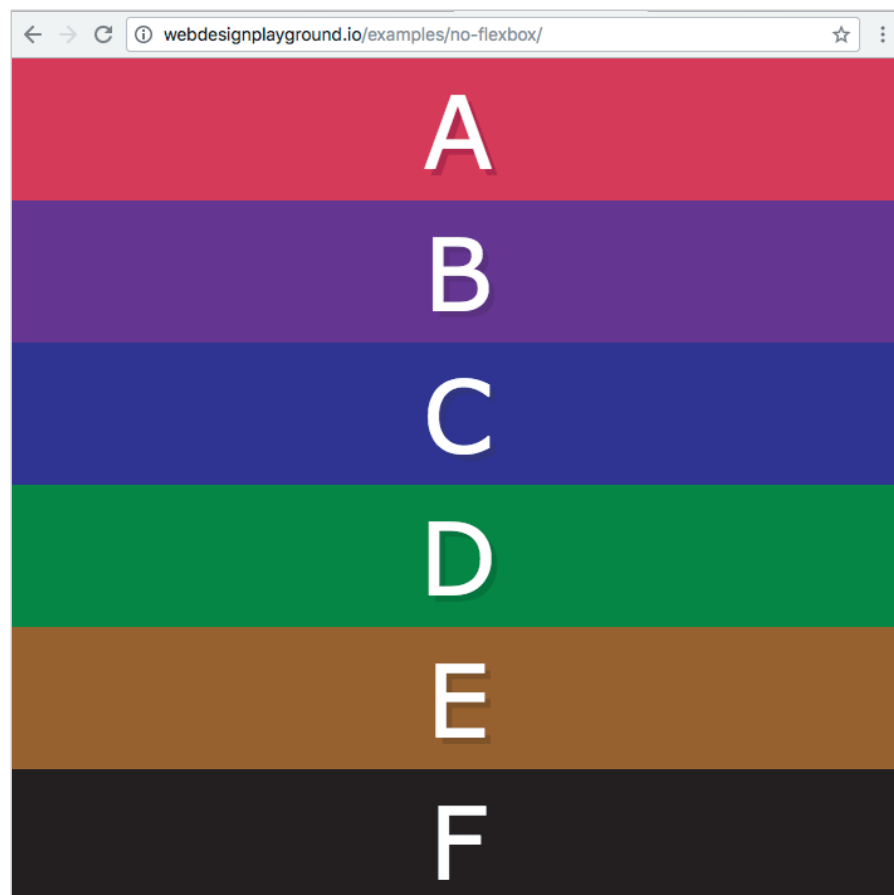
Flexbox is the welcome shorthand for this method's cumbersome official moniker: Flexible Box Layout Module. The underlying principle behind flexbox is to provide a way around the rigid, cumbersome way that the browser handles blocks of content. The default is to stack them. Consider the following collection of `div` elements:

```
<div class="container">
  <div class="item itemA">A</div>
  <div class="item itemB">B</div>
  <div class="item itemC">C</div>
  <div class="item itemD">D</div>
  <div class="item itemE">E</div>
  <div class="item itemF">F</div>
</div>
```

Not shown here are the classes I've applied to give each item element a unique background color, and Figure 12.1 shows the results. As you can see, the `div` elements are stacked and extend the width of the browser window.

► **Figure 12.1**

The default browser layout of the `div` elements





Even if you break out of this default flow with floats or inline blocks, the uncomfortable sense remains that the browser is still in charge and is fitting your blocks where *it* thinks they should go. Yes, you can tame the browser somewhat by styling your floats and inline blocks just so, but there's a brittleness to these tweaks. Try to imagine what happens to the float and inline-block holy-grail layouts if the sidebar text is longer than the article text. (Hint: It's not pretty.)

Flexbox rides to the rescue by offering simple but extremely powerful methods for laying out, distributing, aligning, sizing, and even ordering the child elements in a parent container. The *flex* part of the name comes from one of this technology's main tenets: The child items in a container should be able to change dimensions (width *and* height) by growing to fill in empty space if there's too much of it or by shrinking to allow for a reduction in space. This happens whether the amount of content changes or the size of the screen changes (such as by maximizing a window or by changing a device's screen orientation).

So flexbox is perfect, then? No, it's not. It has two main drawbacks:

- Its inherit flexibility means that it sometimes behaves in ways that appear nonsensical. It can be maddening at first, but when you've used it a few times, you begin to see why flexbox behaves the way it does.
- It's not suitable for large-scale layouts. Flexbox works wonderfully for components of a page—such as a header or sidebar—and is fine for small-scale layouts (such as the holy-grail practice layout). But big, complex projects are almost always too much for flexbox to handle. (If you have the time, wait for CSS Grid Layout to have sufficient browser support.)

When you work with flexbox, you work with two kinds of page objects: containers and items. A *flex container* is any type of parent block element—`div`, `p`, any of the HTML semantic page elements you learned in Chapter 11, even the `body` element—that surrounds one or more elements. These child elements are called *flex items*.

Okay, that's enough theory. It's time to start learning how flexbox works.

Working with Flexbox Containers

Before you can do anything with flexbox, you need to decide which block-level element will be the flex container. When you've done that, you convert that element to a container with a single CSS declaration: `display: flex`. The following rule turns the `header` element into a flex container:

```
header {  
  display: flex;  
}
```

LEARN

To learn CSS Grid basics now, see my tutorial “Getting Started with CSS Grid” on the Web Design Playground.

➡ Online: wdpg.io/grid



Creating Page Layouts with Flexbox

PLAY

You can try out all the flex-direction values interactively on the Playground.

➡ Online: wdpg.io/12-1-2

REMEMBER

The row value is the default, so declaring flex-direction: row is optional.

REMEMBER

If you applied flex-direction: column to this example, you'd get the layout shown in Figure 12.1 earlier in this chapter; the main axis would run from top to bottom, and the cross axis would run left to right. If you applied flex-direction: column-reverse, you'd get the same layout with the div elements in reverse order; the main axis would run bottom to top, and the cross axis would remain left to right.

The container's child elements automatically become flex items; no extra rules or declarations or code are required. From there, you can start customizing your flex container and its items to suit the task at hand.

I find that the best way to learn about and use flexbox is to ask yourself a series of questions—one set for containers and another for items. Here are the container questions:

- In which direction do you want the container's items to run?
- How do you want the items arranged along the main axis?
- How do you want the items arranged along the cross axis?
- Do you want the items to wrap?
- How do you want multiple lines arranged along the cross axis?

(Don't worry if you're not sure what I mean by *main axis* and *cross axis*. All will be revealed in the next section.) The next few sections ask and show you the possible answers to each of these questions.

In which direction do you want the container to run?

The first thing that's flexible about flexbox is that it doesn't dictate one and only one direction for the container's items. Although the browser's default layout rigidly enforces a vertical direction, and although floats and inline blocks work only horizontally, flexbox is happy to go either way. With flexbox, *you* decide.

Perhaps the most important flexbox concept to grasp right from the get-go is the notion that flexbox containers always have two axes:

- **Main**—The axis that runs in the same direction as the container's items
- **Cross**—The axis that runs perpendicular to the main axis (the cross axis is also called the *secondary axis*)

You determine the main-axis direction when you set the flex-direction property on a container:

```
container {  
  display: flex;  
  flex-direction: row|row-reverse|column|column-reverse;  
}
```

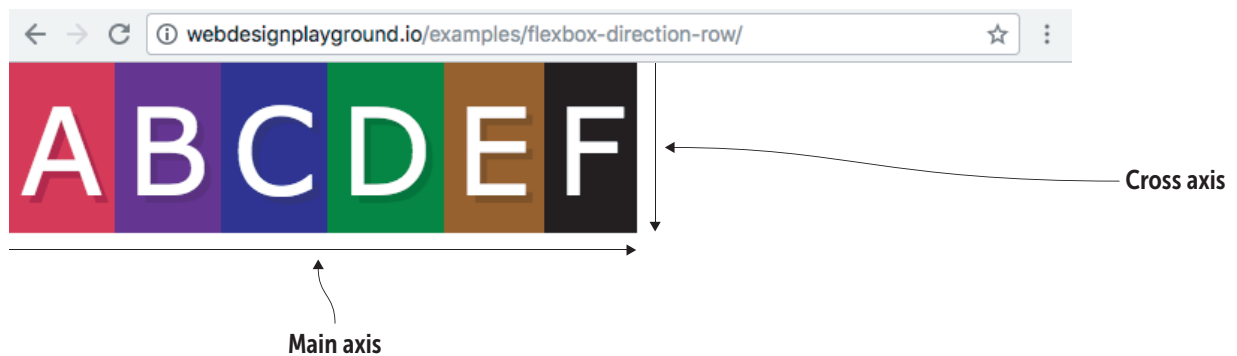
- **row**—Sets the main axis to horizontal, with items running from left to right (the default)
- **row-reverse**—Sets the main axis to horizontal, with items running from right to left
- **column**—Sets the main axis to vertical, with items running from top to bottom
- **column-reverse**—Sets the main axis to vertical, with items running from bottom to top



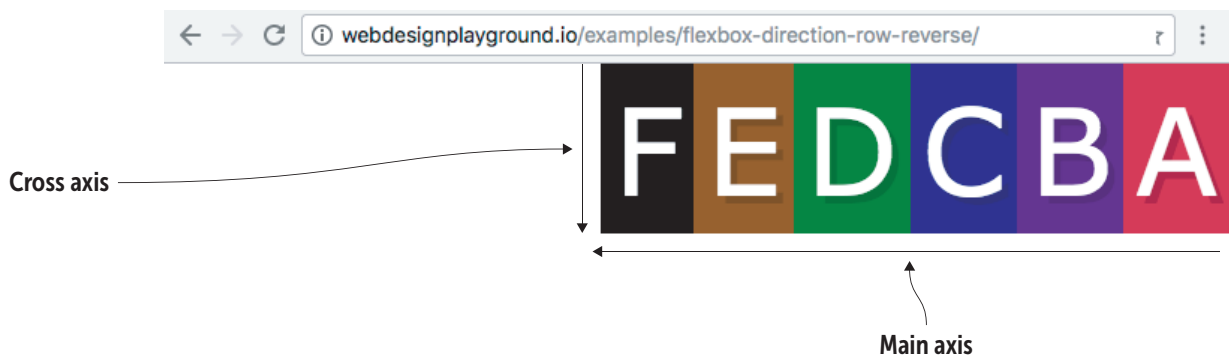
Using the `div` elements shown in Figure 12.1 earlier in this chapter, here's how you'd turn the parent `div` into a flex container by using the right-to-left (`row`) direction:

```
.container {
  display: flex;
  flex-direction: row;
}
```

Figure 12.2 shows the results, and Figure 12.3 shows what happens when you use `flex-direction: row-reverse`.



► **Figure 12.2** The `div` elements with a flex container and the `row` direction applied



► **Figure 12.3** The `div` elements with a flex container and the `row-reverse` direction applied

Figure 12.2 shows the same result as using `float: left` or `display: inline-block`, and Figure 12.3 shows the same result as using `float: right` (and isn't possible with inline blocks). With flexbox, however, you get the result by adding a couple of declarations to the container rather than styling each child element, as you do with floats and inline blocks. Right off the bat, you can see that flexbox is easier and more efficient.

PLAY

How would you use flexbox to display a numbered list in reverse order? ➡ Online: wdpg.io/12-1-4



Creating Page Layouts with Flexbox

REMEMBER

The `flex-start` value is the default, so declaring `justify-content: flex-start` is optional.

BEWARE

The `space-around` value doesn't quite work as advertised because you always get less space before the first item and after the last item (see Figure 12.5 later in this chapter). That happens because each item is given a set amount of space on either side, so inside items have two units of space between them, compared with one unit of space before the first item and after the last item.

PLAY

You can play around with the `justify-content` values interactively on the Playground. ➡ Online: wdpg.io/12-1-5

REMEMBER

The `stretch` value is the default, so declaring `align-items: stretch` is optional.

How do you want the items arranged along the main axis?

When you've used `flex-direction` to set the main axis for the container, your next decision is how you want the items to be arranged along that axis. Use the `justify-content` property on a container:

```
container {
  display: flex;
  justify-content: flex-start|flex-end|center|space-between|space-around;
}
```

- `flex-start`—Places the items at the beginning of the container (the default)
- `flex-end`—Places the items at the end of the container
- `center`—Places the items in the middle of the container
- `space-between`—Places the items with the first item at the beginning of the container, the last item at the end, and the rest of the items evenly distributed in between
- `space-around`—Distributes the items evenly within the container by supplying each item the same amount of space on either side

Figure 12.4 shows the effect that each value has on the arrangement of the items within the container when the main axis is horizontal. (Note that I've added an outline around each container so you can visualize its boundaries.)

How do you want the items arranged along the cross axis?

With the items arranged along the main axis, your next task is choosing an arrangement along the cross axis. You set this by using the container's `align-items` property:

```
container {
  display: flex;
  align-items: stretch|flex-start|flex-end|center|baseline;
}
```

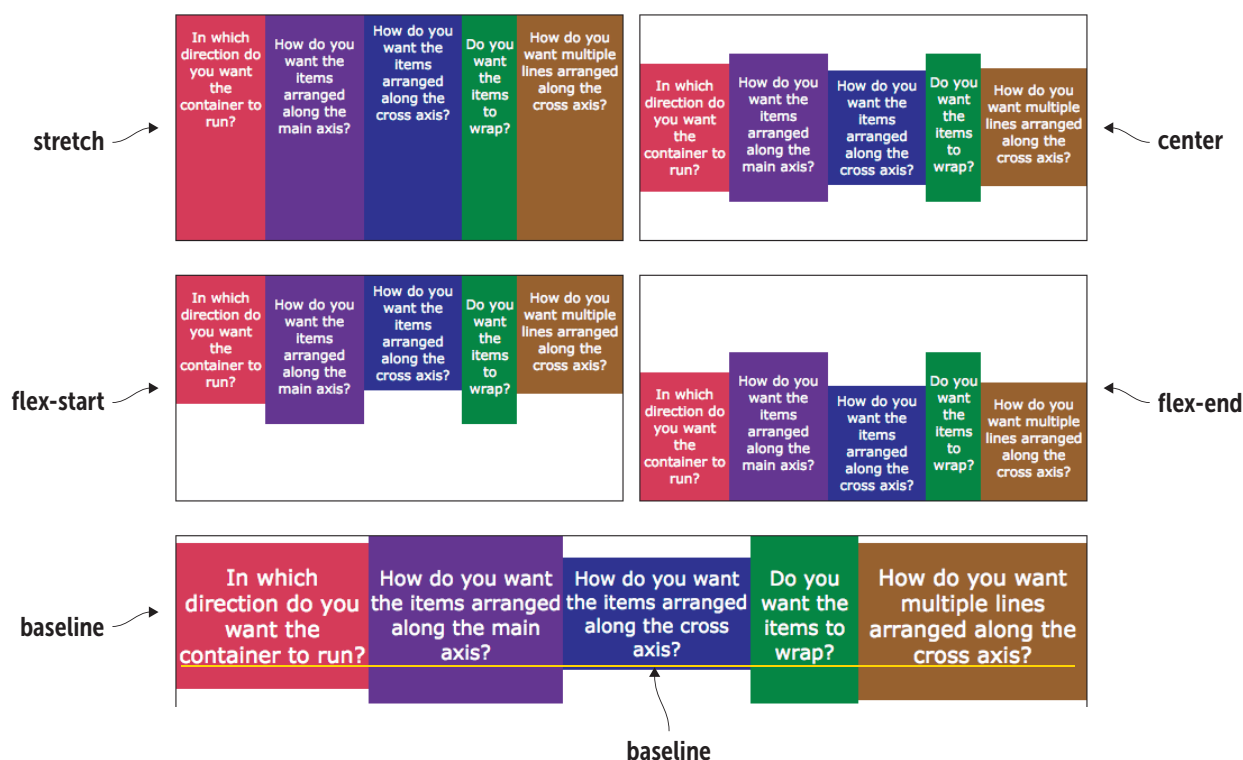
- `stretch`—Expands each item along the cross axis to fill the container (the default)
- `flex-start`—Aligns the items with the beginning of the cross axis
- `flex-end`—Aligns the items at the end of the cross axis
- `center`—Aligns the items in the middle of the cross axis
- `baseline`—Aligns the items along their baseline of the flex container



Figure 12.5 shows the effect that each value has on the arrangement of the items within the container when the cross axis is vertical. (I've added an outline around each container so you can visualize its boundaries.)



Creating Page Layouts with Flexbox



► **Figure 12.5** The `align-items` values in action

FAQ

Are these alignment options confusing, or is it just me? *Almost everyone getting started with flexbox finds alignment to be the most confusing part. It may help to think of the main axis as the justification axis, because you use the `justify-content` property to arrange items on that axis. Similarly, think of the cross axis as the alignment axis, because you arrange items on it using the `align-items` property.*

Do you want the items to wrap?

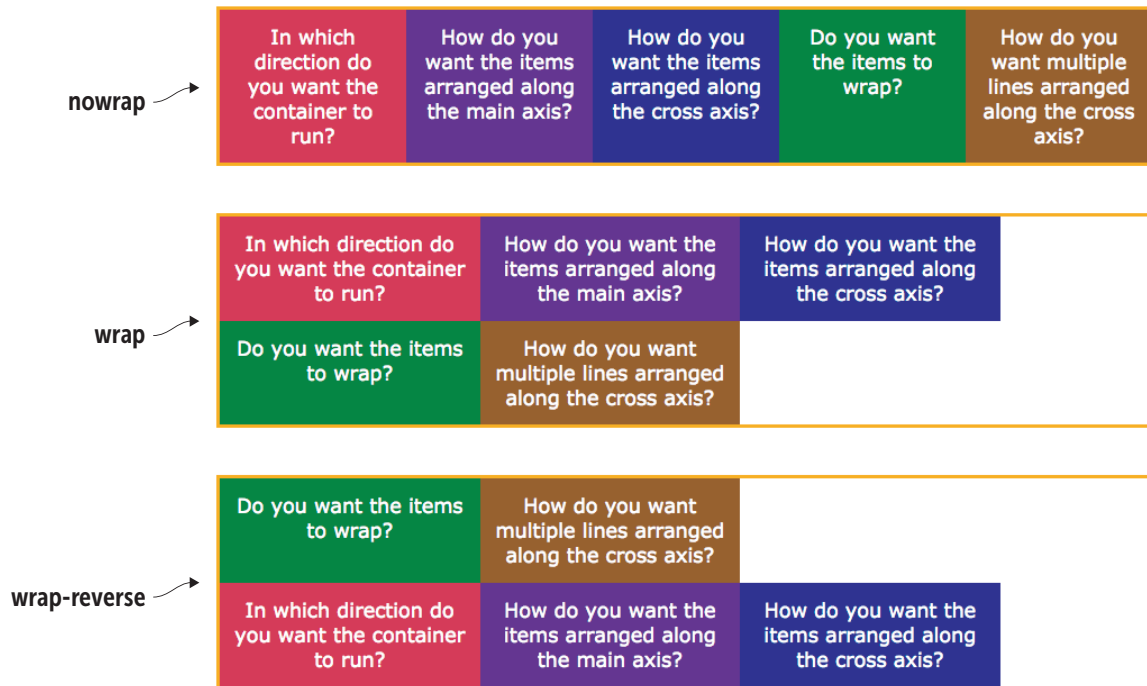
By default, flexbox treats a container as a single row (if you've declared `flex-direction` as `row` or `row-reverse`) or as a single column (if you've declared `flex-direction` as `column` or `column-reverse`). If the container's items are too big to fit into the row or column, flexbox shrinks the items along the main axis to make them fit. Alternatively, you can force the browser to wrap the container's items to multiple rows or columns rather than shrinking them. You do this by using the container's `flex-wrap` property:

```
container {  
  display: flex;  
  flex-wrap: nowrap|wrap|wrap-reverse;  
}
```

- `nowrap`—Doesn't wrap the container's items (the default)
- `wrap`—Wraps the items to as many rows or columns as needed
- `wrap-reverse`—Wraps the items at the end of the cross axis



Figure 12.6 shows the effect that each value has on the arrangement of the items within the container when the main axis is horizontal. (I've added an orange outline around each container so you can visualize its boundaries.)



► **Figure 12.6** How the flex-wrap values work

How do you want multiple lines arranged along the cross axis?

Your final container-related decision is how you want multiple lines—that is, multiple rows or columns—arranged along the cross axis. This is similar to arranging individual flex items along the main axis, except that here, you're dealing with entire lines of items. You control this arrangement by using the container's `align-content` property:

```
container {
  display: flex;
  align-content: stretch | center | flex-start | flex-end | space-between | space-around;
}
```

- `stretch`—Expands the wrapped lines along the cross axis to fill the container height (the default)
- `center`—Places the lines in the middle of the cross axis
- `flex-start`—Places the lines at the beginning of the cross axis

PLAY

You can try out the different `align-items` values interactively on the Playground.

➡ Online: wdpg.io/12-1-6

REMEMBER

The `nowrap` value is the default, so declaring `flex-wrap: nowrap` is optional.

PLAY

You can wrap your head around the three flex-wrap values by trying them out interactively on the Playground.

➡ Online: wdpg.io/12-1-8



Creating Page Layouts with Flexbox

REMEMBER

The stretch value is the default, so declaring align-content: stretch is optional.

PLAY

You can try out all the align-content values interactively on the Playground. ➡ Online: wdpg.io/12-1/10

- flex-end—Places the lines at the end of the cross axis
- space-between—Places the first line at the beginning of the cross axis, the last line at the end, and the rest of the lines evenly distributed in between
- space-around—Distributes the lines evenly within the container by supplying each line with a set amount of space on either side

Figure 12.7 shows the effect that each value has on the arrangement of the lines within the container when the main axis is horizontal. (I've added an orange outline around each container so you can visualize its boundaries.)



► Figure 12.7 Using the align-content values

Lesson 12.1: Dead-Centering an Element with Flexbox

Covers: flex and other flex container properties

➡ Online: wdpg.io/12-1-1

BEWARE

As with justify-content, the space-around value gives one unit of space before the first line and after the last line but two units of space between all the other lines.


By far the most common question related to web page layouts is a deceptively simple one: How do I center an element horizontally and vertically? That is, how can you use CSS to place an element in the dead center of the browser window? Over the years, many clever tricks have been created to achieve this goal, with most of them using advanced and complex CSS rules. Fortunately, you don't have to worry about any of that because flexbox lets you dead-center any element with four lines of CSS, as shown in the following example.



► Example

⇒ Online: wdpg.io/12-1-1

This example shows you how to center an `h1` element horizontally and vertically within the browser window.

WEB PAGE	
CSS	<pre>div { display: flex; justify-content: center; align-items: center; height: 100vh; }</pre> <p>Center the h1 horizontally.</p> <p>Center the h1 vertically.</p> <p>Set the div to the height of the window.</p>
HTML	<pre><div> <h1>Center Me!</h1> </div></pre>

This example works by turning the `div` element into a flex container, which automatically converts the `h1` element to a flex item. By setting both `justify-content` and `align-items` to `center`, and by giving the `div` the full height of the browser window (it's the width of the browser window by default), you center the `h1` in the window.

Working with Flexbox Items

Now that you know everything that's worth knowing about flexbox containers, turn your attention to the flexbox items inside those containers. As before, learning about and using flex items is best approached by asking yourself a series of questions:

- Do you want the item to grow if there's extra room?
- Do you want the item to shrink if there's not enough room?
- Do you want to suggest an initial size for an item?
- Do you want to change an item's order?
- Do you want to override an item's alignment?



Creating Page Layouts with Flexbox

The next few sections discuss these questions and provide you some answers.

Do you want the item to grow if there's extra room?

If you look back at Figure 12.4, notice that in the `flex-start` example, the flex items are bunched up at the beginning of the container, leaving a chunk of empty space to the right. This effect may be what you want, or you may prefer to have the items fill that empty space. You can do that by applying the `flex-grow` property to the item you want to expand:

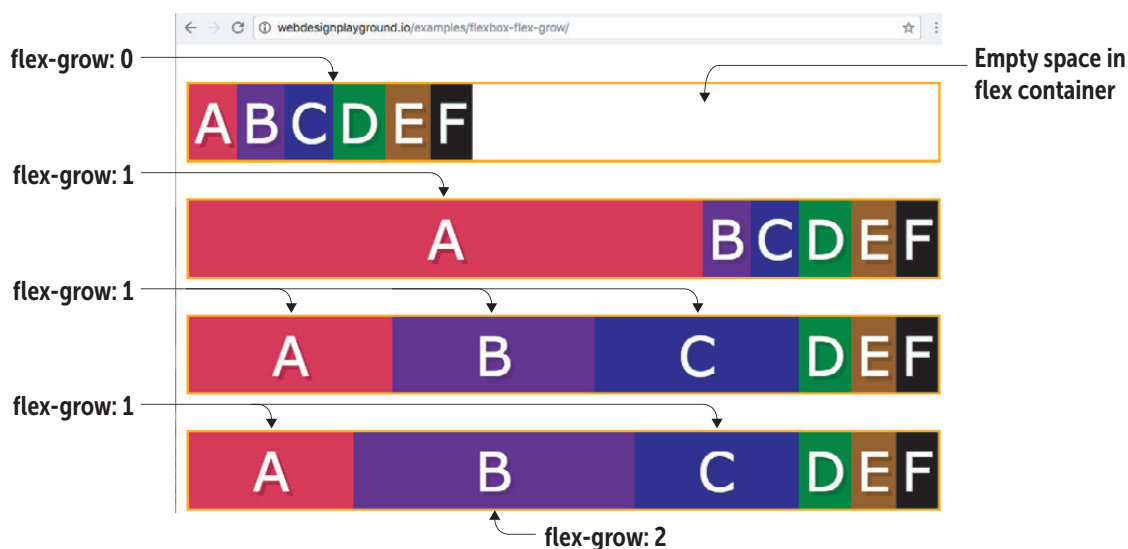
```
item {  
    flex-grow: value;  
}
```

By default, all flex items are given a `flex-grow` value of 0. To grow items to fill a container's empty space, you assign positive numbers to those items as follows (see Figure 12.8):

- If you assign any positive number to one flex item in a container, the amount of empty space in the container is added to that item.
- If you assign the same positive number to multiple flex items in a container, the amount of empty space in the container is divided evenly among those items.
- If you assign different positive numbers to multiple flex items in a container, the amount of empty space in the container is divided proportionally among those items, based on the values you provide. If you assign the values 1, 2, and 1 to three items, those items get 25 percent, 50 percent, and 25 percent of the empty space, respectively.

MASTER

To calculate what proportion of the empty space is assigned to each item, add all the `flex-grow` values for a given container and then divide the individual `flex-grow` values by that total. Values of 1, 2, and 1 add up to 4, for example, so the percentages are 25 ($\frac{1}{4}$), 50 ($\frac{2}{4}$), and 25 ($\frac{1}{4}$), respectively.



► Figure 12.8 The effect of different `flex-grow` values



Do you want the item to shrink if there's not enough room?

The opposite problem of expanding flex items to fill a container's empty space is shrinking flex items when the container doesn't have enough space. This shrinking is activated by default, so if the browser detects that the flex items are too large to fit the container, it automatically reduces the flex items to fit.

How much each item shrinks depends on its size in relation to the other items and the size of the container. Suppose that you're working with a horizontal main axis (that is, `flex-direction` is set to `row`) and that the container is 1200px wide, but each of its five items is 400px wide. That's 2000px total, so the browser must reduce the items by 800px to fit the container. In this case, because all the items are the same width, the browser reduces the width of each by 160px.

If the items have different widths, the calculations get more complicated, so I won't go into them here. Suffice it to say that the amount each item's width gets reduced depends on its initial width. The greater the initial width is, the more the item shrinks.

Rather than let the browser determine how much each item gets reduced, you can specify that a particular item be reduced more than or less than the other items. You do that by applying the `flex-shrink` property to the item:

```
item {  
  flex-shrink: value;  
}
```

By default, all flex items are given a `flex-shrink` value of 1, which means that they're all treated equally when it comes time to calculate the shrink factor. To control the shrink factor yourself, assign positive values to those items as follows (see Figure 12.9):

- If you set `flex-shrink` to a number greater than 1, the browser shrinks the item more than the other items by a factor that's somewhat proportional to the value you provide. (Again, the math is quite complicated.)
- If you set `flex-shrink` to a number greater than 0 but less than 1, the browser shrinks the item less than the other items.
- If you set `flex-shrink` to 0, the browser doesn't shrink the item.

PLAY

You can play with various flex-grow values interactively on the Playground.

➞ Online: wdpg.io/12-2-2

LEARN

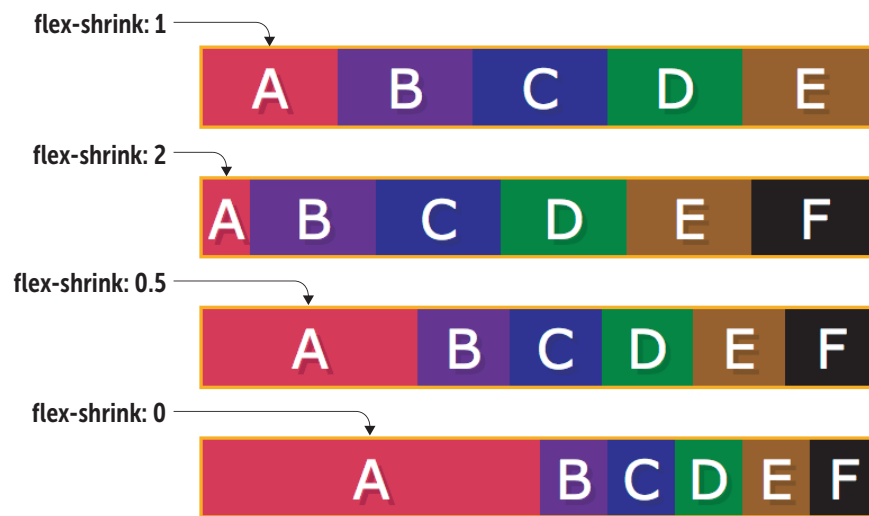
Mike Reithmuller has a lucid explanation of the math involved in calculating item shrinkage here: <https://madebymike.com.au/writing/understanding-flexbox>.

BEWARE

The browser won't shrink an item to a size less than the minimum required to display its content. If you keep increasing an item's `flex-shrink` value, and the item refuses to get smaller, the item is probably at its minimum possible size.



Creating Page Layouts with Flexbox



► **Figure 12.9** The effect of different `flex-shrink` values. Each item is 300px, and the container is 600px.

PLAY

You can try out various `flex-shrink` values interactively on the [Playground](https://wdpg.io/12-2-5). ➡ Online: wdpg.io/12-2-5

Do you want to suggest an initial size for an item?

You've seen that flex items can grow or shrink depending on how they fit in the container and that you have some control of this process via the `flex-grow` and `flex-shrink` properties. But when I say that flex items can grow or shrink, what are they growing and shrinking *from*? That depends:

- If the item has a declared width value (if `flex-direction` is set to `row`) or a declared height value (if `flex-direction` is set to `column`), the item grows or shrinks from this initial size.
- If the item doesn't have a declared width or height, the item's dimensions are set automatically by the browser to the minimum values required to fit the item's content. The item can grow from this initial value, but it can't shrink to a smaller value.

The latter case—that is, not having a declared width (for `flex-direction: row`) or height (for `flex-direction: column`)—causes two problems. First, it prevents an item from shrinking smaller than its content. Second, the initial size (that is, the minimum required to display the content) may be smaller than you require. You can solve both problems by declaring a *flex basis*, which is a suggested size for the item. You do that by applying the `flex-basis` property:

```
item {  
  flex-basis: value|auto|content;  
}
```

- *value*—Sets a specific measure for the width (with `flex-direction: row`) or height (with `flex-direction: column`). You can use any of the CSS measurement units you learned



about in Chapter 7, including `px`, `em`, `rem`, `vw`, and `vh`. You can also set *value* to a percentage.

- **auto**—Lets the browser set the initial value based on the item's `width` or `height` property (the default). In the absence of a declared `width` or `height`, `auto` is the same as `content`, discussed next.
- **content**—Sets the initial `width` or `height` based on the content of the item.

Using the flex shorthand property

You should know that flexbox offers a shorthand property for `flex-grow`, `flex-shrink`, and `flex-basis`. This property is named `flex`, and it uses any of the following syntaxes:

```
item {  
  flex: flex-grow flex-shrink flex-basis;  
  flex: flex-grow flex-shrink;  
  flex: flex-grow flex-basis;  
  flex: flex-grow;  
  flex: flex-basis;  
}
```

Here's an example declaration that uses the default values for each property:

```
flex: 0 1 auto;
```

This example sets `flex-grow` to 1 and `flex-shrink` to 0:

```
flex: 1 0;
```

This final example styles an item with a fixed size of 10em:

```
flex: 0 0 10em;
```

Do you want to change an item's order?

One of the most surprising—and surprisingly handy—tricks offered by flexbox is the ability to change the order of the items in a container. When would you use this feature? Here are two common scenarios:

- One of the important tenets of accessibility is to place a page's main content as near the top of the page as possible. If you have ads or other nonessential content in, say, a left sidebar, that content necessarily appears first in the source document. With flexbox, however, you can put the sidebar's code after the main content and then change its position so that it still appears on the left side of the page.
- A similarly important tenet of mobile web design is to place the main content on the initial screen seen by mobile users. If you don't want to restructure the content for desktop users, you can add a CSS media query that uses flexbox to change the content order, depending on the device being used.



Creating Page Layouts with Flexbox

MASTER

Negative order values are allowed, so an easy way to move an item to the front of its container is to set its order value to -1.

PLAY

You can mess around with some order values interactively on the Playground. ➡ Online: wdpg.io/12-2-6

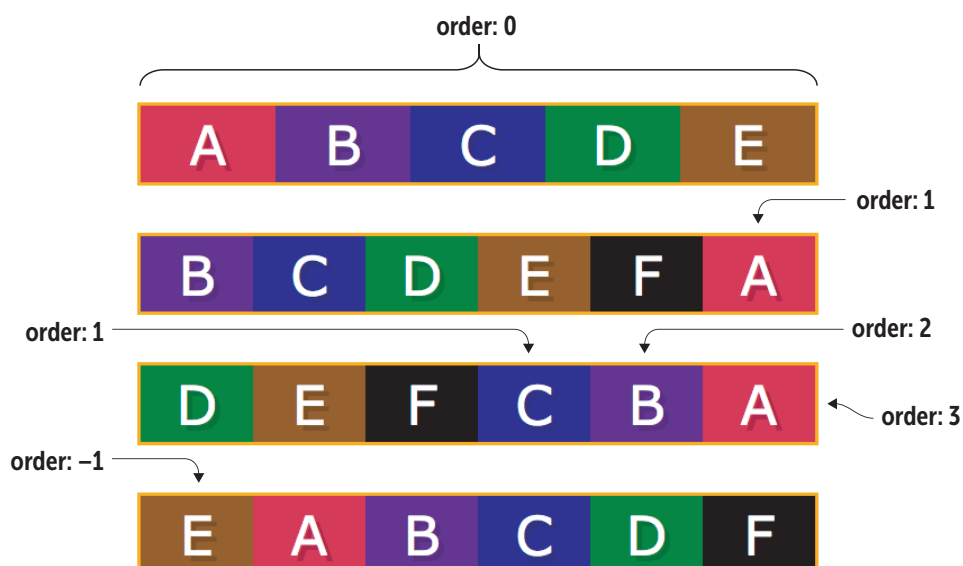
You change the order of a flex item by using the order property:

```
item {  
  order: value;  
}
```

By default, all the items in a flex container are given an order value of 0. You can manipulate the item order as follows:

- The higher an item's order value, the later it appears in the container.
- The item with the highest order value appears last in the container.
- The item with the lowest order value appears first in the container.

Figure 12.10 puts a few order values through their paces.



► Figure 12.10 The effect of different order values

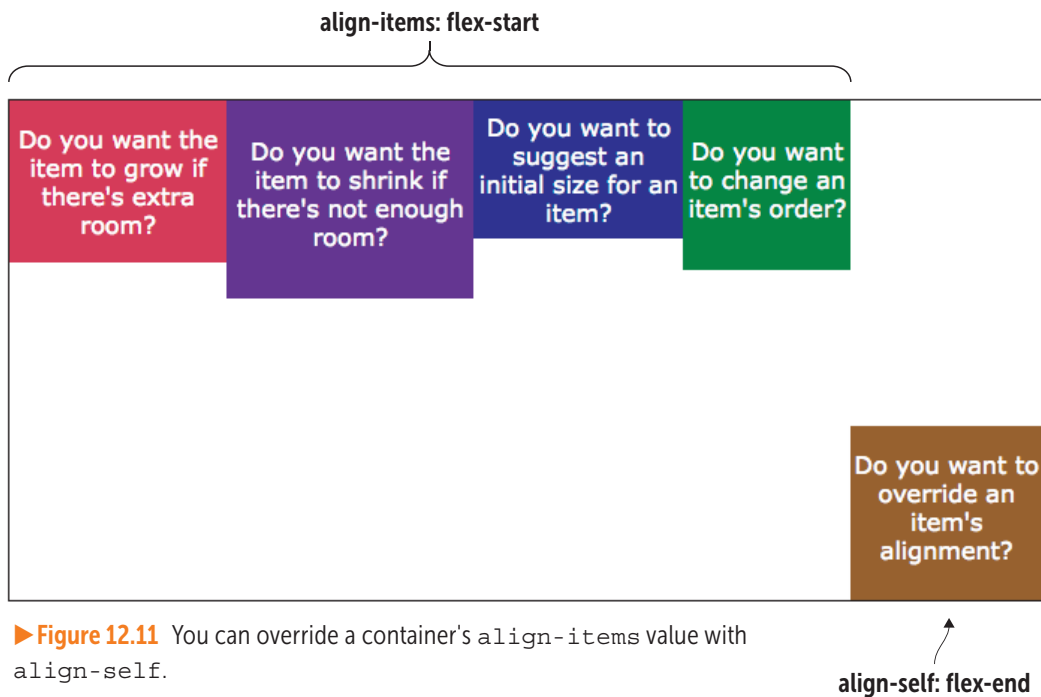
Do you want to override an item's alignment?

You saw earlier that you can use the align-items property to arrange items along a container's cross axis. Rather than align all the items the same way, you may prefer to override this global alignment and assign a different alignment to an item. You can do that by setting the item's align-self property:

```
item {  
  align-self: stretch|flex-start|flex-end|center|baseline;  
}
```




The possible values act in the same manner as I outlined earlier (see "How do you want the items arranged along the cross axis?"). You can also assign the value `auto` to revert the item to the current `align-items` value. Figure 12.11 shows a container with `align-items` set to `flex-start` but with the last item having `align-self` set to `flex-end`.



► **Figure 12.11** You can override a container's `align-items` value with `align-self`.

Flexbox Browser Support

The good news about flexbox browser support is that it works in all current browsers. In fact, it works even in the vast majority of recent browsers, so for the most part, you don't have to worry about using browser prefixes.

If you have to support old browsers, however, some prefixing is required to get flexbox to work. These prefixes can get complex because the flexbox syntax changed between versions, so supporting older browsers means supporting these older syntaxes. Rather than run through all these prefixes, I'm going to pass the buck to a fantastic tool called Autoprefixer (<http://autoprefixer.github.io>), shown in Figure 12.12. You paste your nonprefixed code into the left pane, and fully prefixed code appears automagically in the right



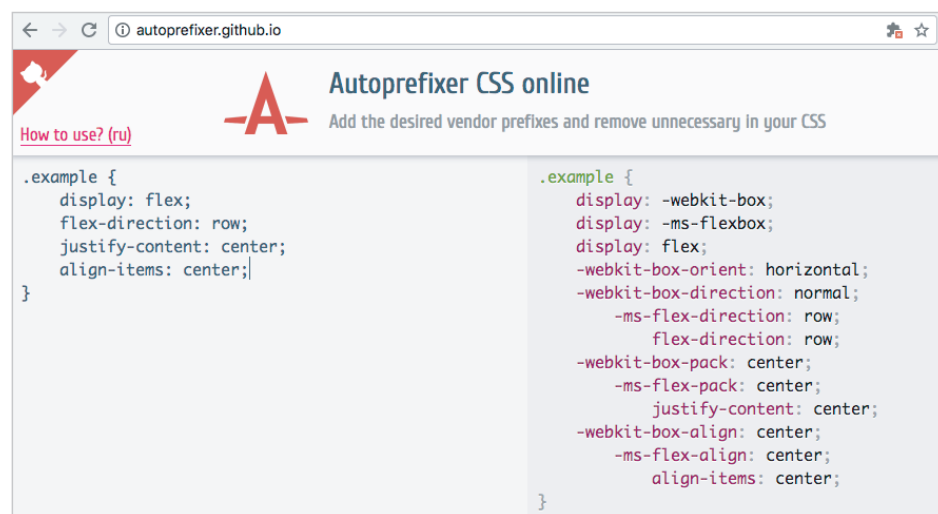
Creating Page Layouts with Flexbox

pane. It also comes with a Filter box that you can use to specify how far back you want to go with browser support:

- Type `last x versions` to support that most recent `x` versions of all browsers (such as `last 4 versions`).
- Type `> y%` to support only web browsers that have at least `y%` market share (such as `> .5%`).

► **Figure 12.12**

Use the online version of Autoprefixer to add browser vendor prefixes to your flexbox code.



Lesson 12.2: *Creating a Thumbnail List*

Covers: The `flex-grow` and `flex-shrink` properties

⇒ Online: wdpg.io/12-2-0

A common web page component is a simple thumbnail list that has a thumbnail image on the left and a description or other information on the right. These elements are used for photo galleries, user directories, book lists, project summaries, and much more. Getting the image and the text to behave is tricky with garden-variety CSS, but it's a breeze with flexbox, as shown in the following example.



► Example

⇒ Online: wdpg.io/12-2-1

This example shows you how to use flexbox to create a thumbnail list of items.

WEB PAGE



animal path

A footpath or track made by the constant and long-term walking of animals.



bridleway

A footpath that is also suitable for a horse and rider.



coffin trail

A footpath used for transporting a coffin to a cemetery for burial.



desire line

An informal path that pedestrians prefer to take to get from one location to another rather than using a sidewalk or other official route.

CSS

```
.dictionary-container {
  list-style-type: none;
}
.dictionary-item {
  display: flex;
}
.dictionary-image {
  flex-shrink: 0;
}
.dictionary-entry {
  flex-grow: 1;
}
```

Each li becomes a flex container.

Prevent the thumbnail from shrinking.

Allow the text to use the rest of the container.

HTML

```
<ul class="dictionary-container">
  <li class="dictionary-item">
    <div class="dictionary-image">
      
    </div>
    <div class="dictionary-entry">
      <h4>animal path</h4>
      <p>A footpath or track made by the constant and long-
term walking of animals.</p>
    </div>
  </li>
  etc.
</ul>
```



Creating Page Layouts with Flexbox

Lesson 12.3: Creating the Holy-Grail Layout with Flexbox

Covers: Layout with `flex` and other flexbox properties

⇒ Online: wdpg.io/12-3-0

Okay, now you can turn your attention to building the holy-grail layout with flexbox. As before, the holy grail includes three instances in which you need content side by side: the header, the navigation bar, and the content columns. In all three instances, you'll place the elements in a flexbox container with a horizontal main axis.

First, however, note that you want these elements stacked, which means that they need a flex container that uses a vertical main axis. The `<body>` tag does the job nicely, so set `body` as a flex container with a vertical main axis and the content starting at the top:

```
body {  
  display: flex;  
  flex-direction: column;  
  justify-content: flex-start;  
  max-width: 50em;  
  min-height: 100vh;  
}
```

Note, too, that I specified a maximum width for the container and a minimum height. You'll see why I used `100vh` when I talk about adding a footer a bit later.

Now do the header, as shown in the following example.

► Example

⇒ Online: wdpg.io/12-3-1

This example shows you how to use flexbox to get the header logo and title side by side.

WEB PAGE

YOUR
LOGO
HERE **Site Title**



CSS

```
header {
  display: flex;
  justify-content: flex-start;
  align-items: center;
  border: 1px solid black;
  padding: 1em;
}
header img {
  flex-shrink: 0;
}
h1 {
  flex-grow: 1;
  padding-left: .5em;
  font-size: 2.5em;
}
```

Display the header element as a flex container.

Prevent the logo from shrinking.

Let the h1 element use the rest of the header space.

HTML

```
<header>
  
  <h1>Site Title</h1>
</header>
```

In this code, I converted the header element to a flex container with the items arranged at the start of the main (horizontal) axis and centered on the cross (vertical) axis.

Now convert the navigation bar to a horizontal flex container, as shown in the following example.

► Example

➡ Online: wdpg.io/12-3-2

This example shows you how to use flexbox to get the navigation-bar items side by side.

WEB PAGE

Home Item Item Item

continued



Creating Page Layouts with Flexbox

CSS

```
nav {  
  padding: .5em;  
  border: 1px solid black;  
}  
nav ul {  
  display: flex;  
  justify-content: flex-start;  
  list-style-type: none;  
  padding-left: .5em;  
}  
nav li {  
  padding-right: 1.5em;  
}
```

Display the ul element
as a flex container.

HTML

```
<nav>  
  <ul>  
    <li>Home</li>  
    <li>Item</li>  
    <li>Item</li>  
    <li>Item</li>  
  </ul>  
</nav>
```

In this case, the `ul` element is converted to a flex container, meaning that the `li` elements become flex items arranged horizontally from the start of the container.

Next, convert the main element's `<article>` and `<aside>` tags to flex items, which gives you the two-column content layout. The following example shows how it's done.

► Example

🔗 Online: wdpg.io/12-3-3

This example shows you how to use flexbox to get the article and aside elements side by side in a two-column layout.

WEB PAGE

Article Title

Article paragraph 1

Article paragraph 2

Sidebar Title

Sidebar paragraph



CSS	<pre> main { display: flex; flex-grow: 1; } article { flex-grow: 3; border: 1px solid black; } aside { flex-grow: 1; border: 1px solid black; } </pre> <p>Display main as a flex container.</p> <p>Let it grow vertically.</p> <p>Let article use three units of space</p> <p>Let aside use one unit of space.</p>
HTML	<pre> <main> <article> <h2>Article Title</h2> <p>Article paragraph 1</p> <p>Article paragraph 2</p> </article> <aside> <h3>Sidebar Title</h3> <p>Sidebar paragraph</p> </aside> </main> </pre>

A couple of interesting things are going on here. First, note that the main element does double duty: It acts as the flex container for the article and aside elements, *and* it's a flex item in the body element's flex container. Setting flex-grow to 1 for the main element tells the browser to give main all the empty vertical space in the body container. Again, why you're doing this will become apparent when you get to the footer.

For the article and aside flex items, I assigned flex-grow values of 3 and 1, respectively, meaning that article gets 75 percent of the available horizontal space and aside gets the remaining 25 percent.

Finally, add the footer element in the same way that you did with the float and inline block layouts in Chapter 11. Figure 12.13 shows the result.

MASTER

Note, too, that the article and aside items are the same height—a pleasant bonus that comes courtesy of the body container's default stretch value for align-items. You get a true full-height sidebar and don't have to resort to a faux column.



Creating Page Layouts with Flexbox



► **Figure 12.13** The complete holy-grail layout using flexbox

PLAY

How would you modify this layout to display the sidebar on the left instead of the right? ➡ Online: wdpg.io/12-3-5

PLAY

How would you modify this layout to display three content columns: a sidebar to the left and to the right of the article element? ➡ Online: wdpg.io/12-3-6

Can you see what's different? That's right: The footer element appears at the bottom of the browser window, which is where it should be in a true holy-grail layout. You got that nice touch by doing three things:

- Turning the body element into a flex container with a vertical main axis
- Declaring `min-height: 100vh` on the body element, which forces the body element to always be at least the same height as the browser window
- Setting `flex-grow: 1` on the main element to force it to use any available empty vertical space in the body container



Summary

- In which direction do you want the container to run? Use `flex-direction`.
- How do you want the items arranged along the main axis? Use `justify-content`.
- How do you want the items arranged along the cross axis? Use the `align-items` property.
- Do you want the items to wrap? Use `flex-wrap`.
- How do you want multiple lines arranged along the cross axis? Add the `align-content` property.
- Do you want the item to grow if there's extra room? Use `flex-grow`.
- Do you want the item to shrink if there's not enough room? Use `flex-shrink`.
- Do you want to suggest an initial size for an item? You can use the `flex-basis` property.
- Do you want to change an item's order? You can use the `order` property.
- Do you want to override an item's alignment? Use `align-self`.