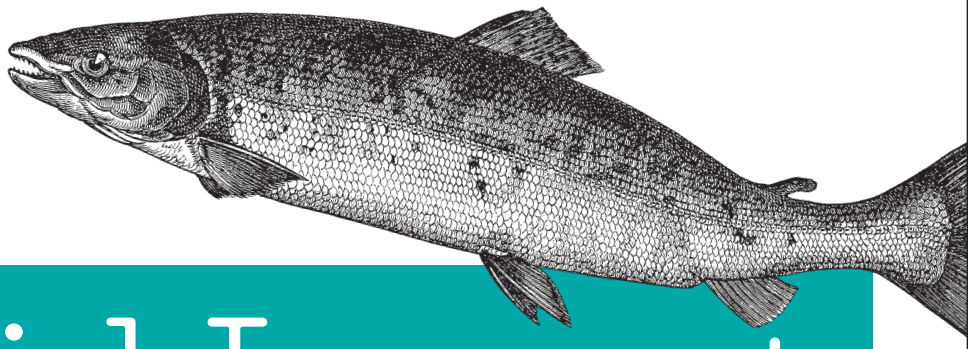
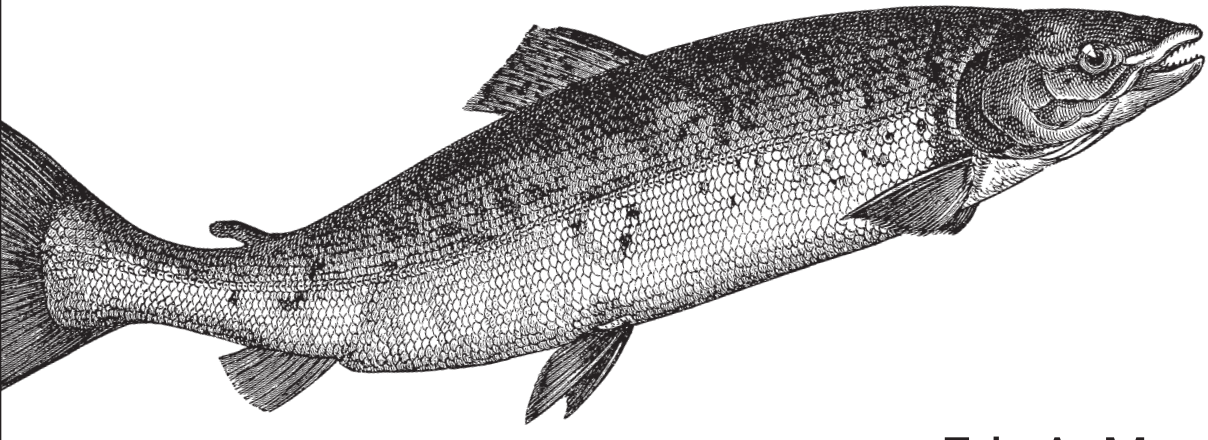
O'REILLY®



Grid Layout in CSS

INTERFACE LAYOUT FOR THE WEB



Eric A. Meyer

O'REILLY®

Grid Layout in CSS

CSS has had a layout-shaped hole at its center since the beginning. Designers have bent features such as float and clear to help fill that hole, but nothing has quite done the job. Now that's about to change. With this concise guide, you'll learn how to use CSS grid layout, a generalized system that lets you lay out pieces of your design independent of their document source order and with full awareness of the overall design.

Short and deep, this book is an excerpt from the upcoming fourth edition of *CSS: The Definitive Guide*. When you purchase either the print or the ebook edition of *Grid Layout in CSS*, you'll receive a discount on the entire *Definitive Guide* once it's released. Why wait? Learn how to make your web pages come alive today.

- Explore the differences between grid boxes and block containers
- Create block-level grids, inline grids, and even nest grids inside grids
- Learn best practices for attaching elements to your layout, using explicitly defined grid lines or grid area
- Understand how the implicit grid automatically adjusts for oversized elements
- Create gutters between grid elements, and align and justify individual items

Eric A. Meyer is an author, speaker, blogger, sometime teacher, and cofounder of An Event Apart. He's a two-decade veteran of the Web and web standards, a past member of the W3C's Cascading Style Sheets Working Group, and the author of O'Reilly's *CSS: The Definitive Guide*

CSS/WEB DEVELOPMENT

US $7.99 **CAN $9.99**

ISBN: 978-1-491-93021-2

Twitter: @oreillymedia
facebook.com/oreilly

Grid Layout in CSS

Interface Layout for the Web

Eric A. Meyer

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Grid Layout in CSS
by Eric A. Meyer

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meg Foley
Production Editor: Colleen Lobner
Copyeditor: Molly Ives Brower
Proofreader: Sharon Wilkey

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

May 2016: First Edition

Revision History for the First Edition
2016-04-14: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491930212* for release details.

978-1-491-93021-2

[LSI]

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a general note.

This element indicates a warning or caution.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/grid-layout-in-css*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Grid Layout

For as long as CSS has existed—which is, believe it or not, two decades now—it's had a layout-shaped hole at its center. We've bent other features to the purposes of layout, most notably `float` and `clear`, and generally hacked our way around that hole. Flexbox layout helped to fill it, but flexbox is really meant for only specific use cases, like navigation bars (navbars).

Grid layout, by contrast, is a *generalized* layout system. With its emphasis on rows and columns, it might at first feel like a return to table layout—and in certain ways that's not too far off—but there is far, far more to grid layout than table layout. Grid allows pieces of the design to be laid out independently of their document source order, and even overlap pieces of the layout, if that's your wish. There are powerfully flexible methods for defining repeating patterns of grid lines, attaching elements to those grid lines, and more. You can nest grids inside grids, or for that matter, attach tables or flexbox containers to a grid. And much, much more.

In short, grid layout is the layout system we've long waited for. There's a lot to learn, and perhaps even more to unlearn, as we leave behind the clever hacks and workarounds that have gotten us through the past 20 years.

Before we get started, a word of caution: as I write this in April 2016, grid layout is still in a bit of flux. We literally delayed publication twice due to changes in the specification and supporting browsers, and there may be still more to come. This is why the book is only available electronically at this point. Therefore, keep in mind that while the overall features and capabilities discussed in this book will almost certainly remain, some details or even properties could change, or be dropped. If you have a habit of checking for updates to your ebooks, this title might warrant more frequent checking. If you don't have that habit, now is an excellent time to start.

Creating a Grid Container

The first step to creating a grid is defining a *grid container*. This is much like a containing block in positioning, or a flex container in flexible-box layout: a grid container is an element that defines a *grid formatting context* for its contents.

At this very basic level, grid layout is actually quite reminiscent of flexbox. For example, the child elements of a grid container become *grid items*, just as the child elements of a flex container become flex items. The children of those child elements do *not* become grid elements—although any grid item can itself be made a grid container, and thus have its child elements become grid items to the nested grid. It's possible to nest grids inside grids, until it's grids all the way down. (Grid layout also has a separate concept of *subgrids* that is distinct from nesting grid containers, but we'll get to that later.)

There are two kinds of grids: *regular* grids and *inline* grids. These are created with special values for the `display` property: `grid` and `inline-grid`. The first generates a block-level box, and the second an inline-level box. The difference is illustrated in Figure 1.

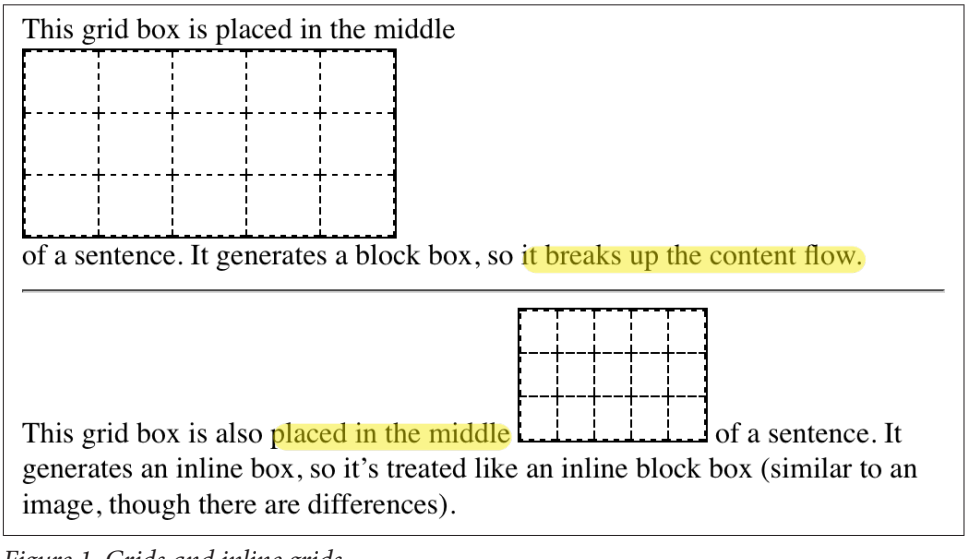


Figure 1. Grids and inline grids

These are very similar to the `block` and `inline-block` values for `display`. Most grids you create are likely to be block-level, though of course the ability to create inline grids is always there.

Although `display: grid` creates a block-level grid, the specification is careful to explicitly state that "grid containers are not block containers." What this means is that

although the grid box participates in layout much as a block container does, there are a number of differences between them.

First off, floated elements do not intrude into the grid container. What this means in practice is that a grid will not "slide under" a floated element, as a block container will do. See Figure 2 for a demonstration of the difference.

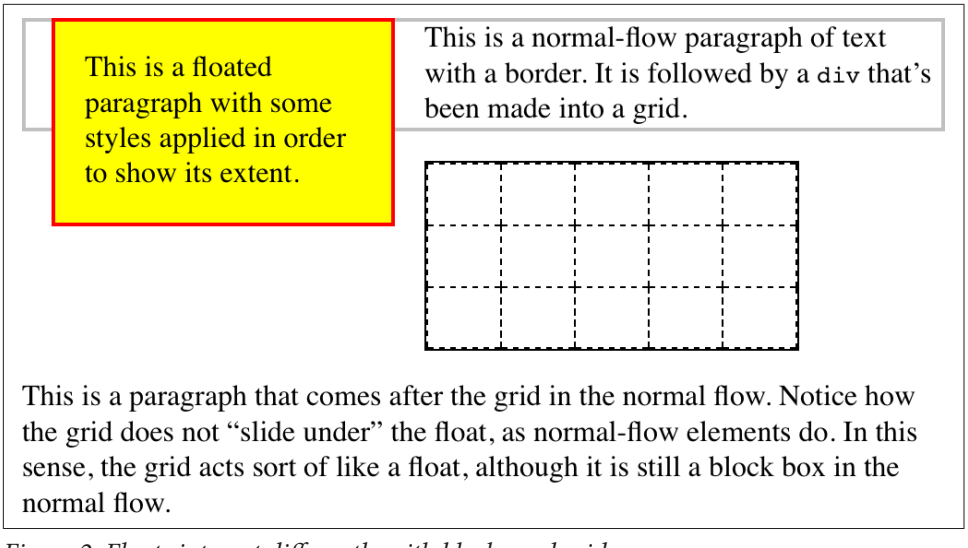


Figure 2. Floats interact differently with blocks and grids

Furthermore, the margins of a grid container do not collapse with the margins of its descendants. Again, this is distinct from block boxes, whose margins do (by default) collapse with descendants. For example, the first list item in an ordered list may have a top margin, but this margin will collapse with the list element's top margin. The top margin of a grid item will *never* collapse with the top margin of its grid container. Figure 3 illustrates the difference.

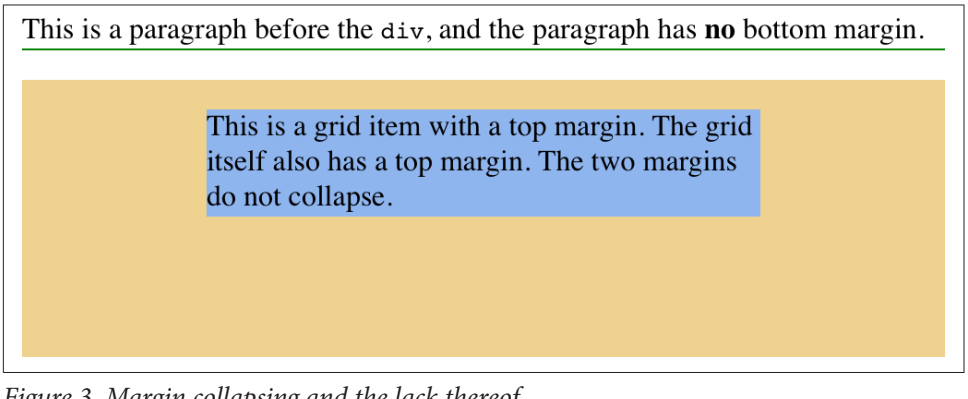


Figure 3. Margin collapsing and the lack thereof

There are a few CSS properties and features that do not apply to grid containers and grid items; specifically:

- All `column` properties (e.g., `column-count`, `columns`, etc.) are ignored when applied to a grid container.
- The `::first-line` and `::first-letter` pseudo-elements do not apply to grid containers and are ignored.
- `float` and `clear` are effectively ignored for grid items (though not grid containers). Despite this, the `float` property still helps determine the computed value of the `display` property for children of a grid container, because the `display` value of the grid items is resolved *before* they're made into grid items.
- The `vertical-align` property has no effect on grid items, though it may affect the content inside the grid item. (There are other, more powerful ways to align grid items, so don't worry.)

Lastly, if a grid container's declared `display` value is `inline-grid` *and* the element is either floated or absolutely positioned, the computed value of `display` becomes `grid` (thus dropping `inline-grid`).

Once you've defined a grid container, the next step is to set up the grid within. Before we explore how that works, though, it's necessary to cover some terminology.

Basic Grid Terminology

We've already talked about grid containers and grid items, but let's define them in a bit more detail. As was said before, a *grid container* is a box that establishes a *grid-formatting context*; that is, an area in which a grid is created and elements are laid out according the rules of grid layout instead of block layout. You can think of it the way an element set to `display: table` creates a table-formatting context within it. Given the grid-like nature of tables, this comparison is fairly apt, though be sure not to make the assumption that grids are just tables in another form. Grids are far more powerful than tables ever were.

A *grid item* is a thing that participates in grid layout within a grid-formatting context. This is usually a child element of a grid container, but it can also be the anonymous (that is, not contained within an element) bits of text that are part of an element's content. Consider the following, which has the result shown in Figure 4:

```
#warning {display: grid;
    background: #FCC; padding: 0.5em;
    grid-template-rows: 1fr;
    grid-template-columns: repeat(7, 1fr);}
```

```
<p id="warning"><img src="warning.svg"><strong>Note:</strong> This element is a
  <em>grid container</em> with several <em>grid items</em> inside it.</p>
```

Note: This element is a *grid container* with several *grid items* inside it.

Figure 4. Grid items

Notice how each element, *and* each bit of text between them, has become a grid item. The image is a grid item, just as much as the elements and text runs—seven grid items in all. Each of these will participate in the grid layout, although the anonymous text runs will be much more difficult (or impossible) to affect with the various grid properties we'll discuss.

If you're wondering about `grid-template-rows` and `grid-template-columns`, we'll tackle them in the next section.

In the course of using those properties, you'll create or reference several core components of grid layout. These are summarized in Figure 5.

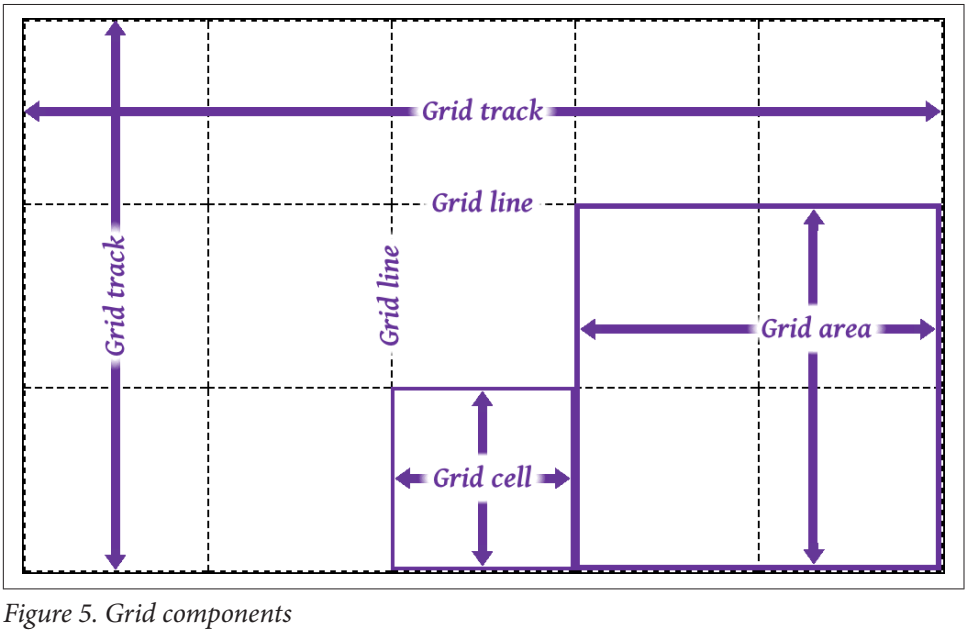


Figure 5. Grid components

The most fundamental unit is the *grid line*. By defining the placement of one or more grid lines, you implicitly create the rest of the grid's components:

- A *grid track* is a continuous run between two adjacent grid lines—in other words, a *grid column* or a *grid row*. It goes from one edge of the grid container to the other. The size of a grid track is dependent on the placement of the grid lines that define it. These are analogous to table columns and rows. More generically, these can be referred to as *block axis* and *inline axis* tracks, where (in Western languages) column tracks are on the block axis and row tracks are on the inline axis.
- A *grid cell* is any space bounded by four grid lines, with no grid lines running through it, analogous to a table cell. This is the smallest unit of area in grid layout. Grid cells cannot be directly addressed with CSS grid properties; that is, no property allows you to say a grid item should be associated with a given cell. (But see the next point for more details.)
- A *grid area* is any rectangular area bounded by four grid lines, and made up of one or more grid cells. An area can be as small as a single cell, or as large as all the cells in the grid. Grid areas *are* directly addressable by CSS grid properties, which allow you to define the areas and then associate grid items with them.

An important thing to note is that these grid tracks, cells, and areas are entirely constructed of grid lines—and more importantly, do not have to correspond to grid items. There is no requirement that all grid areas be filled with an item; it is perfectly possible to have some or even most of a grid's cells be empty of any content. It's also possible to have grid items overlap each other, either by defining overlapping grid areas or by using grid-line references that create overlapping situations.

Another thing to keep in mind is that you can define as many or as few grid lines as you wish. You could literally define just a set of vertical grid lines, thus creating a bunch of columns and only one row. Or you could go the other way, creating a bunch of row tracks and no column tracks (though there would be one, stretching from one side of the grid container to the other).

The flip side to that is if you create a condition where a grid item can't be placed within the column and row tracks you define, or if you explicitly place a grid item outside those tracks, new grid lines and tracks will be automatically added to the grid to accommodate.

Placing Grid Lines

It turns out that placing grid lines can get fairly complex. That's not so much because the concept is difficult; there are just so many different ways to get it done, and each uses its own subtly different syntax.

We'll get started by looking at two closely related properties.

grid-template-rows, grid-template-columns

Values:	none | <track-list> | <auto-track-list> | subgrid <line-name-list>?
Initial value:	none
Applies to:	Grid containers
Inherited:	No
Percentages:	Refer to the inline size (usually width) of the grid container for grid-template-columns, and to the block size (usually height) of the grid container for grid-template-rows
Computed value:	As declared, with lengths made absolute

With these properties, you can define the grid lines in your overall *grid template*, or what the CSS specification calls the *explicit grid*. Everything depends on these grid lines; fail to place them properly, and the whole layout can very easily fall apart.

When you're starting out with CSS grid layout, it's probably a very good idea to sketch out where the grid lines need to be on paper first, or in some close digital analogue. Having a visual reference for where lines should be, and how they should behave, will make writing your grid CSS a lot easier.

There are a lot of ways to specify your grid lines' placement, so before we get started on learning those patterns, there are some basic things to establish.

First, grid lines can always be referred to by number, and can also be named by the author. Take the grid shown in Figure 6, for example. From your CSS, you can use any of the numbers to refer to a grid line, or you can use the defined names, or you can mix them together. Thus, you could say that a grid item stretches from column line `3` to line `steve`, and from row line `skylight` to line `2`.

Note that a grid line can have more than one name. You can use any of them to refer to a given grid line, though you can't combine them the way you can multiple class names. You might think that means it's a good idea to avoid repeating grid-line names, but that's not always the case, as we'll soon see.

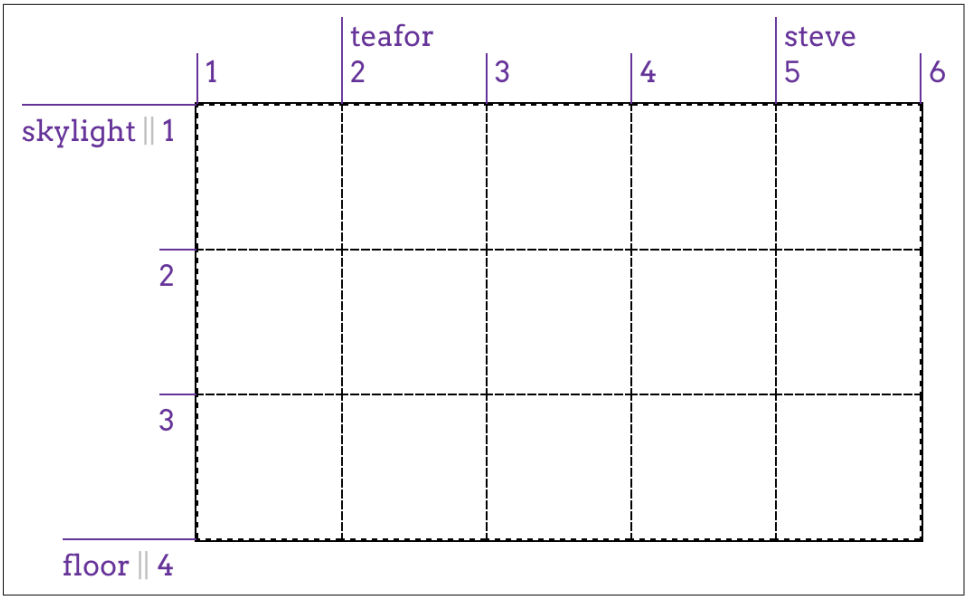


Figure 6. Grid-line numbers and names

I used intentionally silly grid-line names in Figure 6 to illustrate that you can pick any name you like, and also to avoid the implication that there are "default" names. If you'd seen `start` for the first line, you might have assumed that the first line is always called that. Nope. If you want to stretch an element from `start` to `end`, you'll need to define those names yourself. Fortunately, that's simple to do.

As I've said, many value patterns can be used to define the grid template. We'll start with the simpler ones and work our way toward the more complex.

Fixed-Width Grid Tracks

Our first step is to create a grid whose grid tracks are a fixed width. We don't necessarily mean a fixed length like pixels or ems; percentages also count as fixed-width here. In this context, "fixed-width" means the grid lines are placed such that the distance between them does not change due to changes of content.

So, as an example, this counts as a definition of three fixed-width grid columns:

```
#grid {display: grid;
    grid-template-columns: 200px 50% 100px;}
```

That will place a line 200 pixels from the start of the grid container (by default, the left side); a second grid line half the width of the grid container away from the first; and a third line 100 pixels away from the second. This is illustrated in Figure 7.

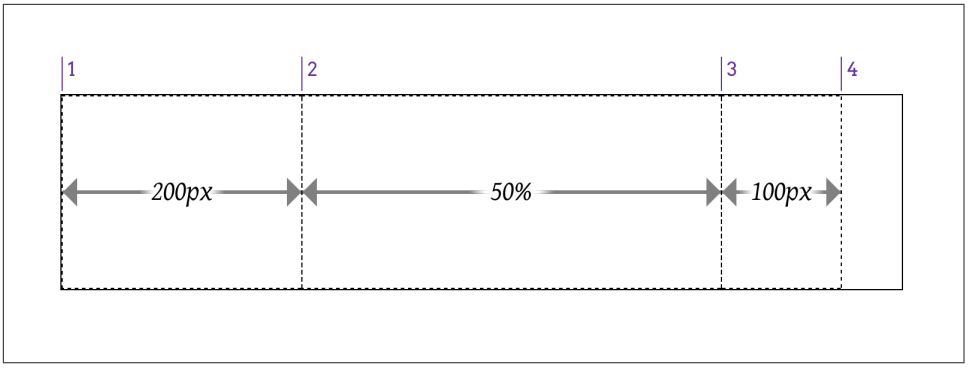


Figure 7. Grid-line placement

While it's true that the second column can change in size if the grid container's size changes, it will *not* change based on the content of the grid items. However wide or narrow the content placed in that second column, the column's width will always be half the width of the grid container.

It's also true that the last grid line doesn't reach the right edge of the grid container. That's fine; it doesn't have to. If you want it to—and you probably will—we'll see various ways to deal with that in just a bit.

This is all lovely, of course, but what if you want to name your grid lines? Just place any grid-line name you want, and as many as you want, in the appropriate place in the value, surrounded by square brackets. It really is as simple as that! Let's add some names to our previous example, with the result shown in Figure 8:

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    }
```

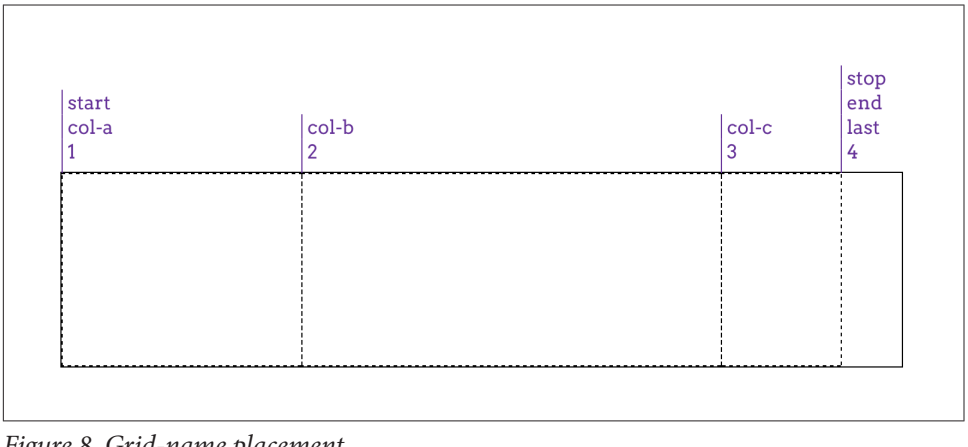


Figure 8. Grid-name placement

What's nice is that adding the names makes clear that each value is actually specifying a grid track's width, which means there is always a grid line to either side of a width value. Thus, for the three widths we have, there are actually four grid lines created.

Row grid lines are placed in exactly the same way as columns, as Figure 9 shows.

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    grid-template-rows:
        [start masthead] 3em [content] 80% [footer] 2em [stop end];
    }
```

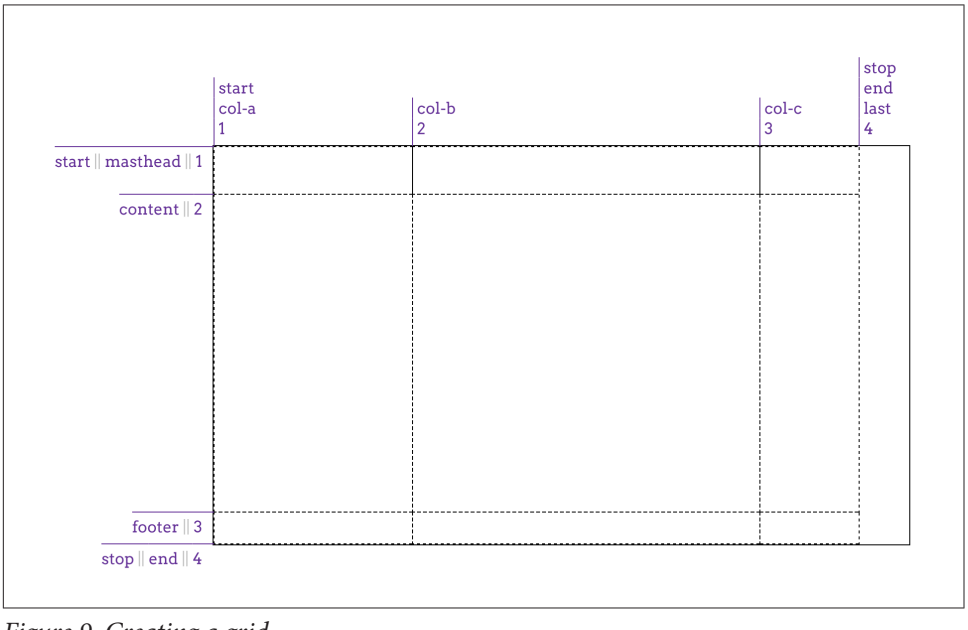


Figure 9. Creating a grid

There are a couple of things to point out here. First, there are both column and row lines with the names `start` and `end`. This is perfectly okay. Grids and columns don't share the same namespace, so you can reuse names like these in the two contexts.

Second is the percentage value for the `content` row track. This is calculated with respect to the height of the grid container; thus, a container 500 pixels tall would yield a `content` row that's 400 pixels tall. This obviously requires that you know ahead of time how tall the grid container will be, which won't always be the case.

You might think we could just say `100%` and have it fill out the space, but that doesn't work, as Figure 10 illustrates: the `content` row track will be as tall as the grid container itself, thus pushing the `footer` row track out of the container altogether.

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    grid-template-rows:
        [start masthead] 3em [content] 100% [footer] 2em [stop end];
    }
```

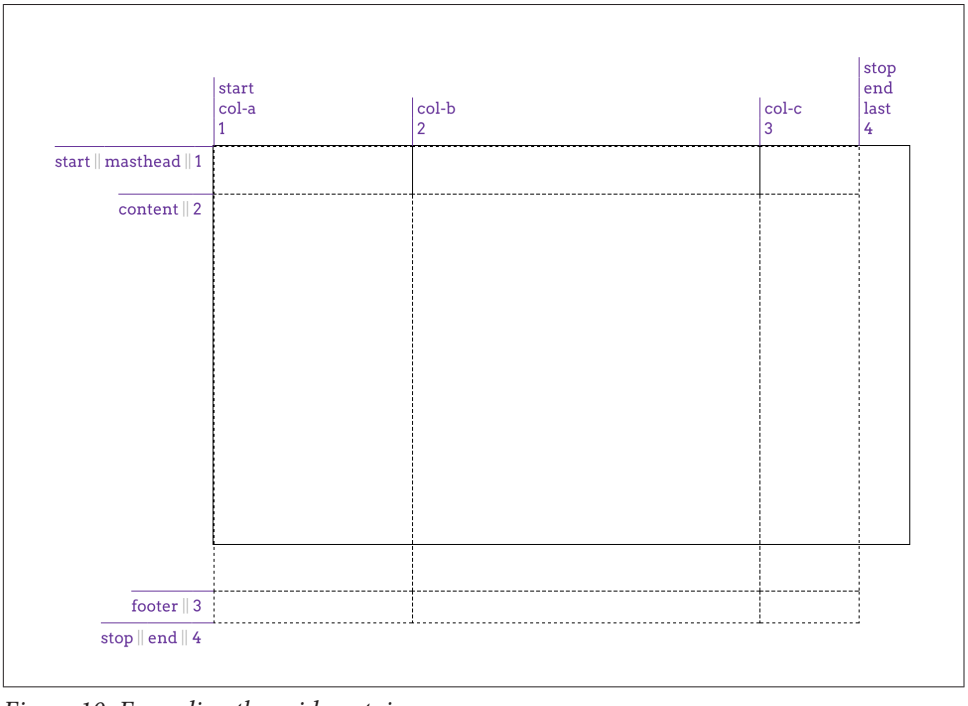


Figure 10. Exceeding the grid container

One way (not necessarily the best way) to handle this scenario is to *minmax* the row's value, telling the browser that you want the row no shorter than one amount and no taller than another, leaving the browser to fill in the exact value. This is done with the `minmax(a,b)` pattern, where `a` is the minimum size and `b` is the maximum size:

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    grid-template-rows:
        [start masthead] 3em [content] minmax(3em,100%) [footer] 2em [stop end];
    }
```

What we've said there is to make the `content` row never shorter than 3 ems tall, and never taller than the grid container itself. This allows the browser to bring up the size until it's tall enough to fit the space left over from the `masthead` and `footer` tracks, and no more. Of course, it also allows the browser to make it shorter than that, as

long as it's not shorter than `3em`, so this is not a guaranteed result. Figure 11 shows one possible outcome of this approach.

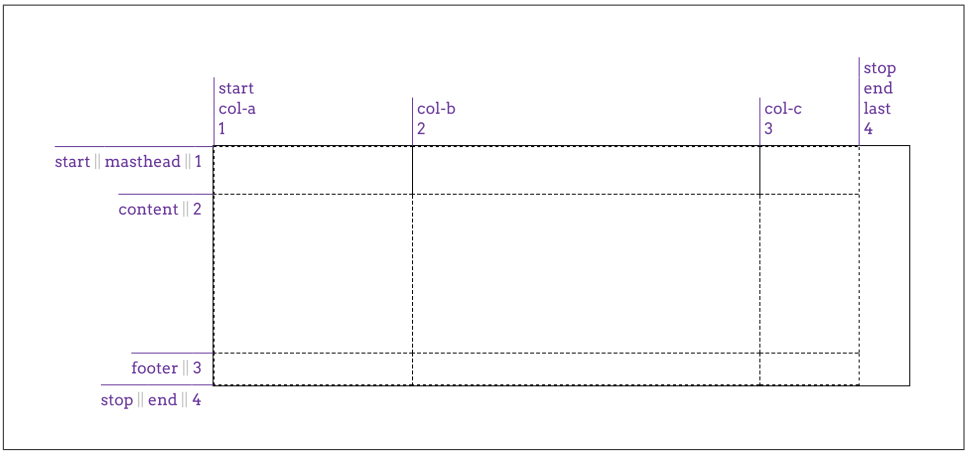


Figure 11. Adapting to the grid container

In like fashion, with the same caveats, `minmax()` could have been used to help the `col-b` column fill out the space across the grid container. The thing to remember with `minmax()` is that if the *max* is smaller than the *min*, then the *max* value is thrown out and the *min* value is used as a fixed-width track length. Thus, `minmax(100px, 2em)` would resolve to `100px` for any font-size value smaller than `50px`.

If the vagueness of `minmax()`'s behavior unsettles you, there are alternatives to this scenario. We could also have used the `calc()` value pattern to come up with a track's height (or width). For example:

```
grid-template-rows:
    [start masthead] 3em [content] calc(100%-5em) [footer] 2em [stop end];
```

That would yield a `content` row exactly as tall as the grid container minus the sum of the `masthead` and `footer` heights, as we saw in the previous figure.

That works as far as it goes, but is a somewhat fragile solution, since any changes to the `masthead` or `footer` heights will also require an adjustment of the calculation. It also becomes a lot more difficult (or impossible) if you want more than one column to flex in this fashion. As it happens, there are much more robust ways to deal with this sort of situation, as we'll soon see.

Flexible Grid Tracks

Thus far, all our grid tracks have been *inflexible*—their size determined by a length measure or the grid container's dimensions, but unaffected by any other considerations. *Flexible* grid tracks, by contrast, can be based on the amount of space in the

grid container not consumed by inflexible tracks, or alternatively, can be based on the actual content of the entire track.

Fractional units

If you want to divide up whatever space is available by some fraction and distribute the fractions to various columns, the `fr` unit is here for you.

In the simplest case, you can divide up the whole container by equal fractions. For example, if you want four columns, you could say

```
grid-template-columns: 1fr 1fr 1fr 1fr;
```

In this very limited case, that's equivalent to saying

```
grid-template-columns: 25% 25% 25% 25%;
```

The result either way is shown in Figure 12.

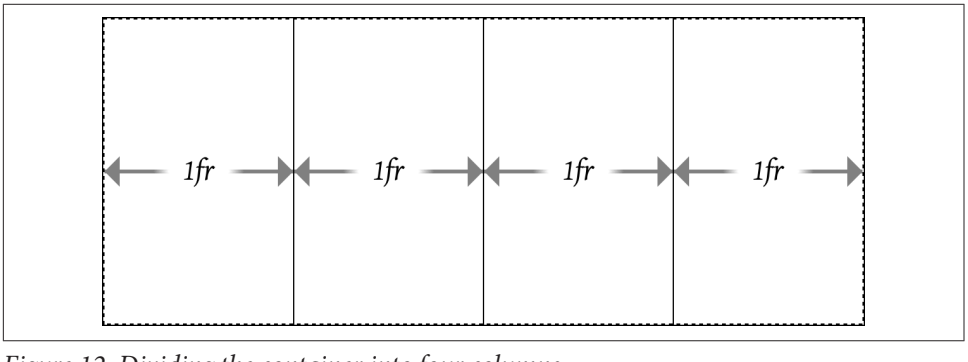


Figure 12. Dividing the container into four columns

Now suppose we want to add a fifth column, and redistribute the column size so they're all still equal. With percentages, we'd have to rewrite the entire value to be five instances of `20%`. With `fr`, though, we can just add another `1fr` to the value and have everything done for us automatically:

```
grid-template-columns: 1fr 1fr 1fr 1fr 1fr;
```

The way `fr` units work is that all of the `fr` values are added together, with the available space divided by that total. Then each track gets the number of those fractions indicated by its number.

What that meant for the first of the previous examples is that when there were four `fr` values, their numbers were added together to get a total of four. The available space was thus divided by four, and each column got one of those fourths. When we added a fifth `1fr`, the space was divided by five, and each column got one of those fifths.

You are not required to always use `1` with your `fr` units! Suppose you want to divide up a space such that there are three columns, with the middle column twice as wide as the other two. That would look like this:

```
grid-template-columns: 1fr 2fr 1fr;
```

Again, these are added up and then 1 is divided by that total, so the base `fr` in this case is `0.25`. The first and third tracks are thus 25% the width of the container, whereas the middle column is half the container's width, because it's `2fr`, which is twice `0.25`, or `0.5`.

You aren't limited to integers, either. A recipe card for apple pie could be laid out using these columns:

```
grid-template-columns: 1fr 3.14159fr 1fr;
```

I'll leave the math on that one as an exercise for the reader. (Lucky you! Just remember to start with `1 + 3.14159 + 1`, and you'll have a good head start.)

This is a convenient way to slice up a container, obviously, but there's more here than just replacing percentages with something more intuitive. Fractional units really come into their own when there are some fixed columns and some flexible space. Consider, for example, the following, which is illustrated in Figure 13:

```
grid-template-columns: 15em 1fr 10%;
```

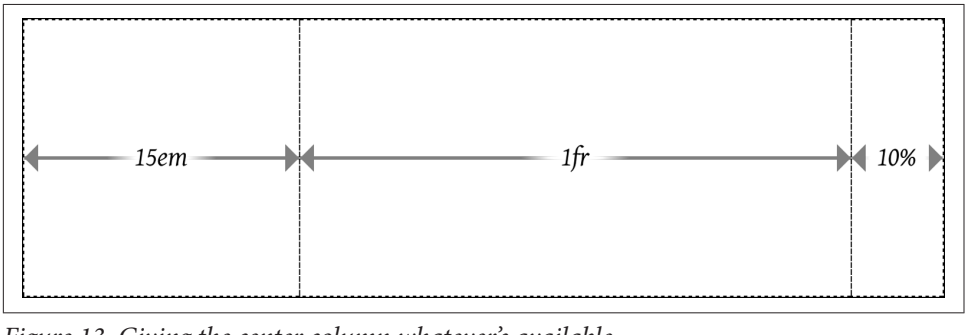


Figure 13. Giving the center column whatever's available

What happened there is that the browser assigned the first and third tracks to their inflexible widths, and then gave whatever was left in the grid container to the center track. This means that for a 1,000-pixel-wide grid container whose `font-size` is the usual browser default of `16px`, the first column will be 240 pixels wide and the third will be 100 pixels wide. That totals 340 pixels, leaving 660 pixels unassigned to inflexible tracks. The fractional units total one, so 660 is divided by one, yielding 660 pixels, all of which are given to the single `1fr` track. If the grid container's width is increased to 1,400 pixels, the third column will be 140 pixels wide and the center column 1,020 pixels wide.

Just like that, we have a mixture of fixed and flexible columns. We can keep this going, splitting up any flexible space into as many fractions as we like. Consider this:

```
width: 100em; grid-template-columns: 15em 4.5fr 3fr 10%;
```

In this case, the columns will be sized as shown in Figure 14.

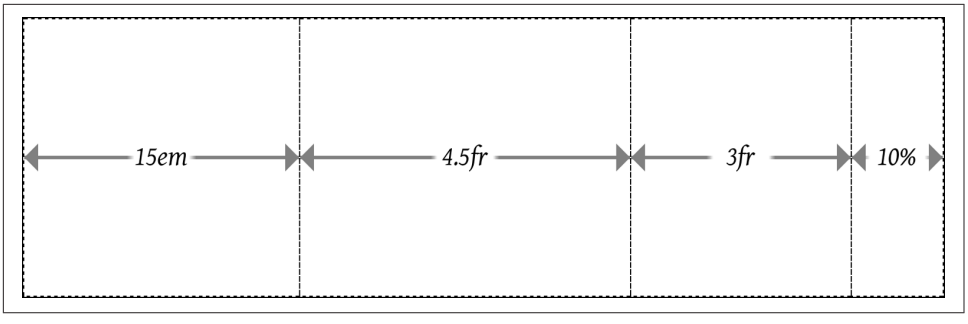


Figure 14. Flexible column sizing

Yes, admittedly, I put a thumb on the scales for Figure 14: the `fr` total and `width` value were engineered to yield nice, round numbers for the various columns. This is purely to aid understanding, of course. If you want to work through the process with less tidy numbers, consider using `92.5em` or `1234px` for the `width` value in the previous example.

In cases where you want to define a minimum or maximum size for a given track, `minxmax()` can be quite useful in some situations. To extend the previous example, suppose the third column should never be less than `5em` wide, no matter what. The CSS would then be

```
grid-template-columns: 15em 4.5fr minmax(5em,3fr) 10%;
```

Now the layout will have two flexible columns at its middle, down to the point that the third column reaches `5em` wide. Below that point, the layout will have three inflexible columns (`15em`, `5em`, and `10%` wide, respectively) and a single flexible column that will get all the leftover space, if there is any. Once you run the math, it turns out that up to `30.5556em` wide, the grid will have one flexible column. Above that width, there will be two such columns.

You might think that this works the other way—for example, if you wanted to make a column track flexible up to a certain point, and then become fixed after, you would declare a minimum `fr` value. This won't work, sadly, because `fr` units are not allowed in the *min* position of a `minmax()` expression. So any `fr` value provided as a minimum will invalidate the declaration. (In earlier implementations, a minimum `fr` was set to zero. In future implentations, `fr` values will likely be allowed to be minimum sizes.)

Speaking of setting to zero, let's look at a situation where the minimum value is explicitly set to `0`, like this:

```
grid-template-columns: 15em 1fr minmax(0,500px) 10%;
```

Figure 15 illustrates the narrowest grid width at which the third column can remain 500 pixels wide. Any narrower, and the minmaxed column will be narrower than 500 pixels. Any wider, and the second column, the `fr` column, will grow beyond zero width while the third column stays at 500 pixels wide.

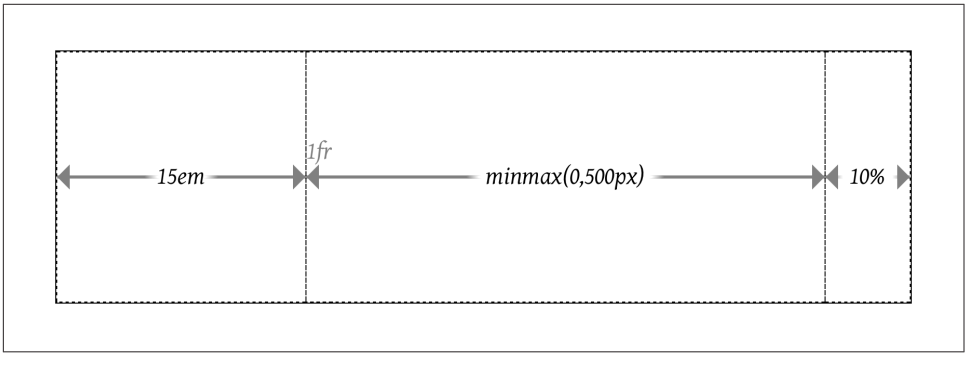


Figure 15. Minmaxed column sizing

If you look closely, you'll see the `1fr` label next to the boundary between the `15em` and `minmax(0,500px)` columns. That's there because the `1fr` is placed with its left edge on the second column grid line, and has no width, because there is no space left to flex. Similarly, the `minmax` is placed on the third column grid line. It's just that, in this specific situation, the second and third column grid lines are in the same place (which is why the `1fr` column has zero width).

If you ever run into a case where the minimum value is greater than the maximum value, then the whole thing is replaced with the minimum value. Thus, `minmax(500px,200px)` would be treated as a simple `500px`. You probably wouldn't do this so obviously, but this feature is useful when mixing things like percentages and fractions. Thus, you could have a column that's `minmax(10%,1fr)` that would be flexible down to the point where the flexible column was less than 10% of the grid container's width, at which point it would stick at `10%`.

Fractional units and minmaxes are usable on rows just as easily as columns; it's just that rows are rarely sized in this way. You could easily imagine setting up a layout where the masthead and footer are fixed tracks, while the content is flexible down to a certain point. That might look something like this:

```
grid-template-rows: 3em minmax(5em,1fr) 2em;
```

That works okay, but it's a lot more likely that you'll want to size that row by the height of its content, not some fraction of the grid container's height. The next section shows exactly how to make that happen.

Content-aware tracks

It's one thing to set up grid tracks that take up fractions of the space available to them, or that occupy fixed amounts of space. But what if you want to line up a bunch of pieces of a page and you can't guarantee how wide or tall they might get? This is where `min-content` and `max-content` come in.

What these keywords mean is simple to state, but not necessarily simple to describe in full. `max-content` means, in effect, "take up the maximum amount of space needed for this content." For large blocks of text (like a blog post), this would generally mean taking as much room as is available, in order to maximize the space for that content.

`min-content`, by contrast, means "take up the bare minimum space needed for this content." With text, that means squeezing the width down to the point that the longest word (or widest inline element, if there are things like images or form inputs) sits on a line by itself. That would lead to a lot of linebreaks in a very skinny, very tall grid element.

What's so powerful about these sizing keywords is that they apply to the entire grid track they define. For example, if you size a column to be `max-content`, then the entire column track will be as wide as the widest content within it. This is easiest to illustrate with a grid of images (12 in this case) with the grid declared as follows and shown in Figure 16.

```
#gallery {display: grid;
    grid-template-columns: max-content max-content max-content max-content;
    grid-template-rows: max-content max-content max-content;}
```

Looking at the columns, we can see that each column track is as wide as the widest image within that track. Where a bunch of portrait images happened to line up, the column is more narrow; where a landscape image showed up, the column was made wide enough to fit it. The same thing happened with the rows. Each row is as tall as the tallest image within it, so wherever a row happened to have all short images, the row is also short.

The advantage here is that this works for any sort of content, no matter what's in there. So let's say we add captions to the photos. All of the columns and rows will resize themselves as needed to handle both text and images, as shown in Figure 17.

Obviously, this isn't a full-fledged design—the images are out of place, and there's no attempt to constrain the caption widths. In fact, that's exactly what we should expect from `max-content` values for the column widths. Since it means "make this column wide enough to hold all its content," that's what we got.

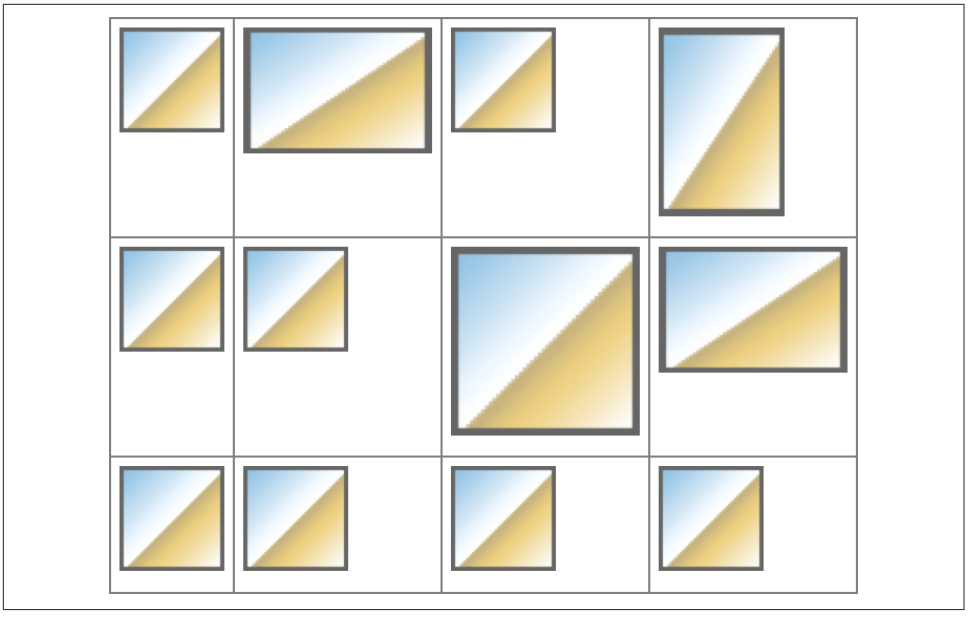


Figure 16. Sizing grid tracks by content

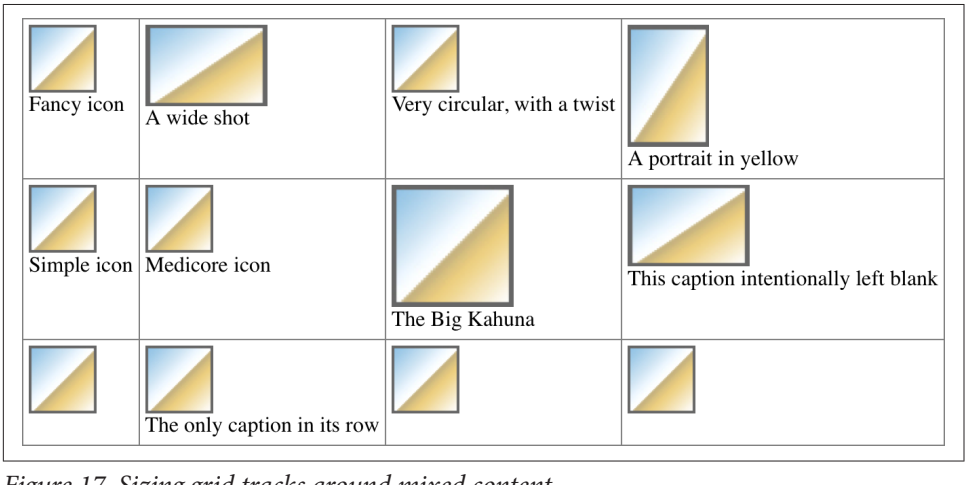


Figure 17. Sizing grid tracks around mixed content

What's important to realize is that this will hold even if the grid tracks have to spill out of the grid container. That means that even if we'd assigned something like `width: 250px` to the grid container, the images and captions would be laid out just the same. That's why things like `max-content` tend to appear in `minmax()` statements. Consider the following, where grids with and without `minmax()` appear side by side. In both cases, the grid container is represented by an orange background (see Figure 18):

```
#g1 {display: grid;
    grid-template-columns: max-content max-content max-content max-content;
    }
#g2 {display: grid;
    grid-template-columns: minmax(0,max-content)  minmax(0,max-content)
          minmax(0,max-content) minmax(0,max-content);
    }
```

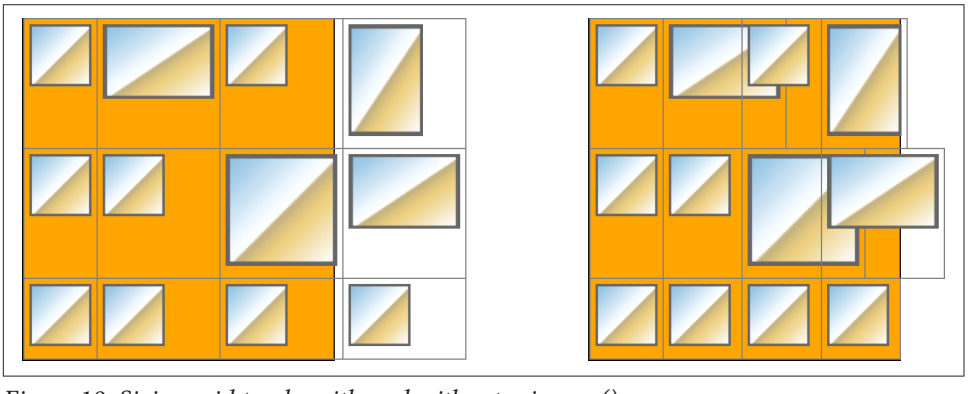


Figure 18. Sizing grid tracks with and without minmax()

In the first instance, the grid items completely contain their contents, but they spill out of the grid container. In the second, the `minmax()` directs the browser to keep the columns within the range of `0` and `max-content`, so they'll all be fit into the grid container if possible. A variant on this would be to declare `minmax(min-content, max-content)`, which leads to a slightly different result than the `0, max-content` approach.

If you're wondering what happens if you `min-content` both the columns and the rows, it's pretty much the same as applying `min-content` to the columns and leaving the rows alone. This happens because the grid specification directs browsers to resolve column sizing first, and row sizing after that.

There's one more keyword you can use with grid track sizing, which is `auto`. As a minimum, it's treated as the minimum size for the grid item, as defined by `min-width` or `min-height`. As a maximum, it's treated the same as `max-content`. You might think this means it can be used only in `minmax()` statements, but this is not the case. You can use it anywhere, and it will take on either a minimum or maximum role. Which one it takes on depends on the other track values around it, in ways that are frankly too complicated to get into here. As with so many other aspects of CSS, using `auto` is essentially letting the browser do what it wants. Sometimes that's fine, but in general you'll probably want to avoid it.

There is a caveat to that last statement: `auto` values allow grid items to be resized by the `align-content` and `justify-content` properties, a topic we'll discuss in a later section. Since `auto` values are the only track-sizing values that permit this, there may be very good reasons to use `auto` after all.

Speaking of things to avoid, you've probably been wondering about the repetitive grid template values, and what happens if you need more than three or four grid tracks. Will you have to write out every single track width individually? Indeed not, as we'll see in the next section.

Repeating Grid Lines

If you have a situation where you want to set up a bunch of grid lines, you probably don't want to have to type out every single one of them. Fortunately, `repeat()` is here to make sure you don't have to.

Let's say we want to set up a column grid line every 5 ems, and set up 10 column tracks. Here's how to do that:

```
#grid {display: grid;
    grid-template-columns: repeat(10, 5em);}
```

That's it. Done. Ten column tracks, each one `5em` wide, for a total of 50 ems of column tracks. It sure beats typing `5em` 10 times!

Any track-sizing value can be used in a repeat, from `min-content` and `max-content` to `fr` values to `auto`, and so on, and you can put together more than one sizing value. Suppose we want to define a column structure such that there's a `2em` track, then a `1fr` track, and then another `1fr` track—and, furthermore, we want to repeat that pattern three times. Here's how to do that, with the result shown in Figure 19:

```
#grid {display: grid;
    grid-template-columns: repeat(3, 2em 1fr 1fr);}
```

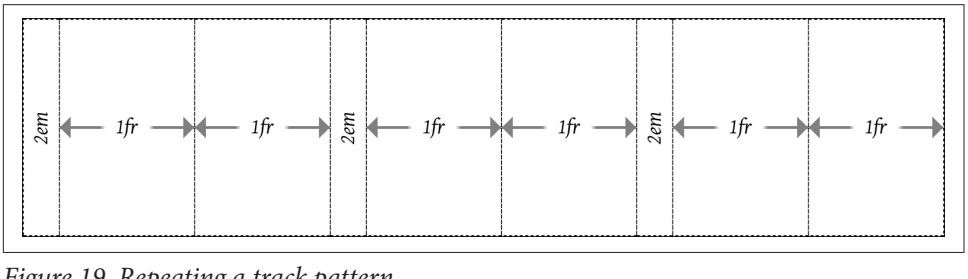


Figure 19. Repeating a track pattern

Notice how the last column track is a `1fr` track, whereas the first column track is `2em` wide. This is an effect of the way the `repeat()` was written. It's easy to add another `2em` track at the end, in order to balance things out, simply by making this change:

```
#grid {display: grid;
    grid-template-columns: repeat(3, 2em 1fr 1fr) 2em;}
```

See that extra `2em` at the end of the value? That adds one more column track after the three repeated patterns. This highlights the fact that `repeat` can be combined with any other track-sizing values—even other repeats—in the construction of a grid. The one thing you *can't* do is nest a repeat inside another repeat.

Other than that, just about anything goes within a `repeat()` value. Here's an example taken straight from the grid specification:

```
#grid {
    display: grid;
    grid-template-columns: repeat(4, 10px [col-start] 250px [col-end]) 10px;}
```

In this case, there are four repetitions of a 10-pixel track, a named grid line, a 250-pixel track, and then another named grid line. Then, after the four repetitions, a final 10-pixel column track. Yes, that means there will be four column grid lines named `col-start`, and another four named `col-end`, as shown in Figure 20. This is acceptable; grid-line names are not required to be unique. (We'll see how to handle this sort of thing in a later section.)

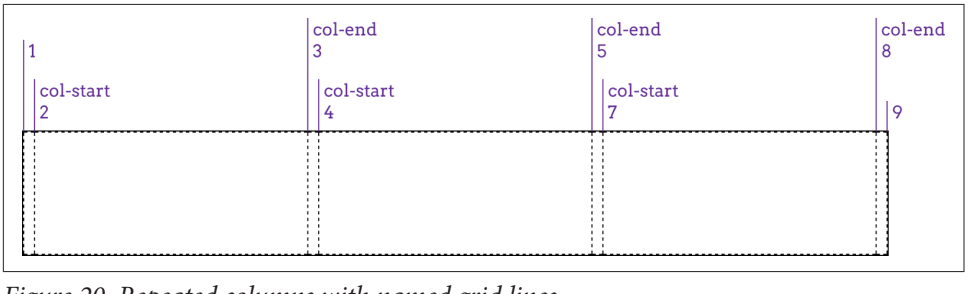


Figure 20. Repeated columns with named grid lines

One thing to remember, if you're going to repeat named lines, is that if you place two named lines next to each other, they'll be merged into a single, double-named grid line. In other words, the following two declarations are equivalent:

```
grid-template-rows: repeat(3, [top] 5em [bottom]);
grid-template-rows: [top] 5em [bottom top] 5em [top bottom] 5em [bottom];
```

If you're concerned about having the same name applied to multiple grid lines, don't be: there's nothing preventing it, and it can even be helpful in some cases. We'll explore ways to handle such situations in a later section.

Auto-filling tracks

There's even a way to set up a simple pattern and repeat it until the grid container is filled. This doesn't have quite the same complexity as regular `repeat()` (at least not yet) but it can still be pretty handy.

For example, suppose we want to have the previous row pattern repeat as many times as the grid container will comfortably accept:

```
grid-template-rows: repeat(auto-fill, [top] 5em [bottom]);
```

That will define a row line every 5 ems until there's no more room. Thus, for a grid container that's 11 ems tall, the following is equivalent:

```
grid-template-rows: [top] 5em [bottom top] 5em [bottom];
```

If the grid container's height is increased past 15 ems, but is less than 20 ems, then this is an equivalent declaration:

```
grid-template-rows: [top] 5em [bottom top] 5em [top bottom] 5em [bottom];
```

The limitation with auto-repeating is that it can take only an optional grid-line name, a fixed track size, and another optional grid-line name. So `[top] 5em [bottom]` represents about the maximum value pattern. You can drop the named lines and just repeat `5em`, or just drop one of the names. It's not possible to repeat multiple fixed track sizes, nor can you repeat flexible track sizes. (Which makes sense: how many times would a browser repeat `1fr` to fill out a grid container?)

You might wish you could auto-repeat multiple track sizes in order to define "gutters" around your content columns. This is unnecessary because grids have a concept of (and properties to define) track gutters, which we'll cover in a later section.

Furthermore, you can have only one auto-repeat in a given track template. Thus, the following would *not* be permissible:

```
grid-template-columns: repeat(auto-fill, 4em) repeat(auto-fill, 100px);
```

It makes sense that this isn't permitted, once you look at it. If the first fills up the grid container with 4-em column tracks, where is there room for any repeated 100-pixel column tracks?

However, you *can* combine fixed-repeat tracks with auto-fill tracks. For example, you could start with three wide columns, and then fill the rest of the grid container with narrow tracks (assuming there's space for them). That would look something like this:

```
grid-template-columns: repeat(3, 20em) repeat(auto-fill, 2em);
```

You can flip that around, too:

```
grid-template-columns: repeat(auto-fill, 2em) repeat(3, 20em);
```

That works because the grid layout algorithm assigns space to the fixed tracks first, and then fills up whatever space is left with auto-repeated tracks. The end result of that example is to have one or more auto-filled 2-em tracks, and then three 20-em tracks.

With `auto-fill`, you will always get at least one column, even if it won't fit into the grid container for some reason. You'll also get as many tracks as will fit, even if some of the tracks don't have content in them. As an example, suppose you set up an auto-fill that placed five columns, but only the first three of them actually ended up with grid items in them. The other two would remain in place, holding open layout space.

If you use `auto-fit`, on the other hand, then tracks that don't contain any grid items will be dropped. Suppose the following:

```
grid-template-columns: repeat(auto-fit, 20em);
```

If there's room for five column tracks in the grid container (i.e., it's more than 100 ems wide), but two tracks don't have any grid items to go into them, those empty grid tracks will be dropped, leaving the three column tracks that *do* contain grid items. The leftover space is handled in accordance with the values of `align-content` and `justify-content` (discussed in a later section).

There's a reason this section hasn't had any figures: as of early 2016, `auto-fill` and `auto-fit` had not been publicly implemented, though most grid-supporting browsers had announced plans to support them soon. Test thoroughly before use!

Grid Areas

Do you remember ASCII art? Well, it's back, and thanks to the `grid-template-areas` property, you can use it to create complete grid templates!

grid-template-areas

Values:	none | <string>
Initial value:	none
Applies to:	Grid containers

Inherited:	No
Computed value:	As declared

We could go through a wordy description of how this works, but it's a lot more fun to just show it. The following rule has the result shown in Figure 21:

```
#grid {display: grid;
    grid-template-areas:
        "h h h h"
        "l c c r"
        "l f f f";}
```

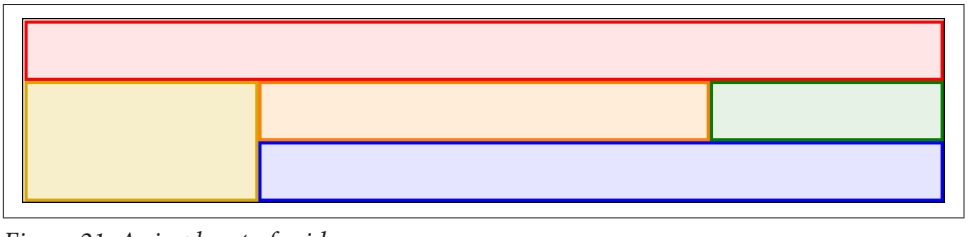


Figure 21. A simple set of grid areas

That's right: the letters in the string values are used to define how areas of the grid are shaped. Really! And you aren't even restricted to single letters! For example, we could expand the previous example like so:

```
#grid {display: grid;
    grid-template-areas:
        "header    header   header   header"
        "leftside  content  content  rightside"
        "leftside  footer   footer   footer";}
```

The grid layout would be the same as that shown in Figure 21, though the name of each area would be different (e.g., `footer` instead of `f`).

In defining template areas, the whitespace is collapsed, so you can use it (as I did in the previous example) to visually line up columns of names in the value of `grid-template-areas`. You could line them up with spaces or tabs, whichever will annoy your coworkers the most. Or you can just use a single space to separate each identifier, and not worry about the names lining up with each other. You don't even have to linebreak between strings; the following works just as well as a pretty-printed version:

```
grid-template-areas: "h h h h" "l c c r" "l f f f";
```

What you can't do is merge those separate strings into a single string and have it mean the same thing. Every new string (as delimited by the double quote marks)

defines a new row in the grid. So the previous example, like the examples before it, defines three rows. If we merged them all into a single string, like so:

```
grid-template-areas:
    "h h h h
     l c c r
     l f f f";
```

…then we'd have a single row of twelve columns, starting with the four-column area `h` and ending with the three-column area `f`. The linebreaks aren't significant in any way, except as whitespace that separates one identifier from another.

If you look at these values closely, you may come to realize that each individual identifier represents a grid cell. Let's bring back our first example from this section, and consider the result shown in Figure 22:

```
#grid {display: grid;
    grid-template-areas:
        "h h h h"
        "l c c r"
        "l f f f";}
```

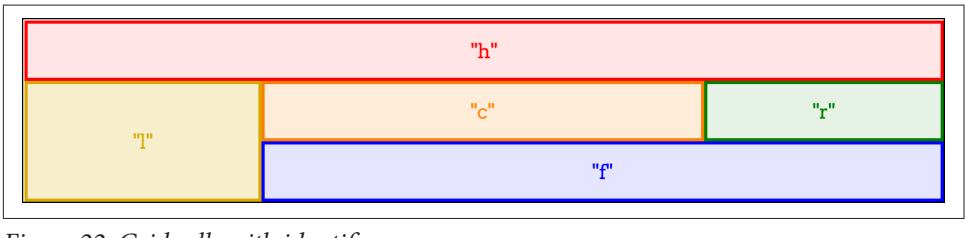


Figure 22. Grid cells with identifiers

This is exactly the same layout result, but here, we've shown how each grid identifier in the `grid-template-areas` value corresponds to a grid cell. Once all the cells are identified, the browser merges any adjacent cells with the same name into a single area that encloses all of them—as long as they describe a rectangular shape! If you try to set up more complicated areas, the entire template is invalid. Thus, the following would result in no grid areas being defined:

```
#grid {display: grid;
    grid-template-areas:
        "h h h h"
        "l c c r"
        "l l f f";}
```

See how `l` outlines an "L" shape? That simple change causes the entire `grid-template-areas` value to be dropped as invalid. A future version of grid layout may allow for nonrectangular shapes, but for now, this is what we have.

If you have a situation where you want to only define some grid cells to be part of grid areas, but leave others unlabeled, you can use one or more `.` characters to fill in for those unnamed cells. Let's say you just want to define some header, footer, and sidebar areas, and leave the rest unnamed. That would look something like this, with the result shown in Figure 23:

```
#grid {display: grid;
    grid-template-areas:
        "header  header  header  header"
        "left    ...     ...     right"
        "footer  footer  footer  footer";}
```

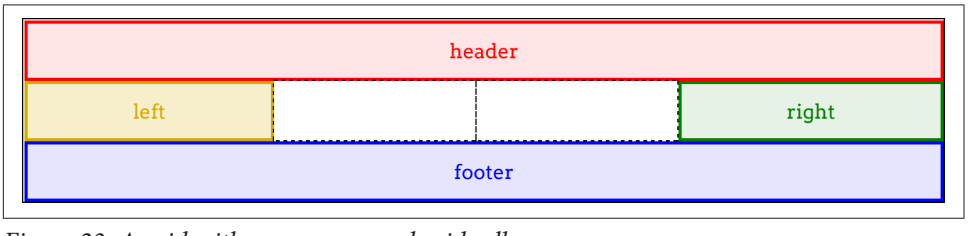


Figure 23. A grid with some unnamed grid cells

The two grid cells in the center of the grid are not part of a named area, having been represented in the template by *null cell tokens* (the `.` identifiers). Where each of those `...` sequences appears, we could have used one or more null tokens—so `left . . right` or `left ..... ..... right` would have worked just as well.

You can be as simple or creative with your cell names as you like. If you want to call your header `steve` and your footer `podiatrist`, go for it. You can even use any Unicode character above codepoint U+0080, so `ConHugeCo©®™` and `åwësømë` are completely valid area identifiers.

Now, to size the grid tracks created by these areas, we bring in our old friends `grid-template-columns` and `grid-template-rows`. Let's add both to the previous example, with the result shown in Figure 24:

```
#grid {display: grid;
    grid-template-areas:
        "header  header  header  header"
        "left    ...     ...     right"
        "footer  footer  footer  footer";
    grid-template-columns: 1fr 20em 20em 1fr;
    grid-template-rows: 40px 10em 3em;}
```

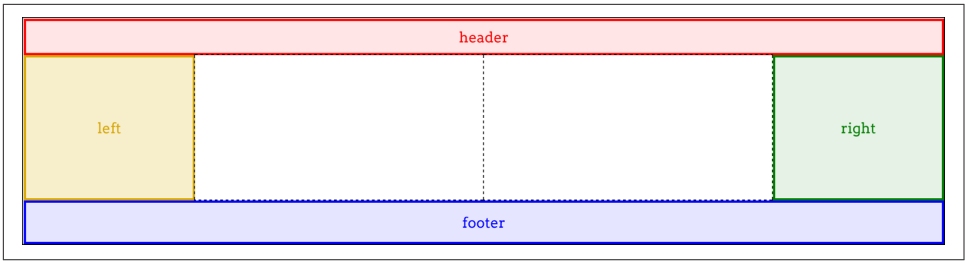


Figure 24. Named areas and sized tracks

Thus, the columns and rows created by naming the grid areas are given track sizes. If we give more track sizes than there are area tracks, that will simply add more tracks past the named areas. Therefore, the following CSS will lead to the result shown in Figure 25:

```
#grid {display: grid;
    grid-template-areas:
        "header  header  header  header"
        "left    ...     ...     right"
        "footer  footer  footer  footer";
    grid-template-columns: 1fr 20em 20em 1fr 1fr;
    grid-template-rows: 40px 10em 3em 20px;}
```

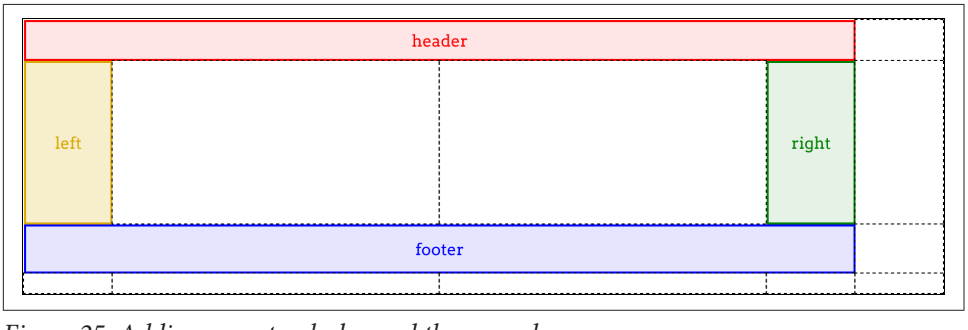


Figure 25. Adding more tracks beyond the named areas

So, given that we're naming areas, how about mixing in some named grid lines? As it happens, we already have: naming a grid area automatically adds names to the grid lines at its start and end. For the `header` area, there's an implicit `header-start` name on its first column-grid line *and* its first row-grid line, and `header-end` for its second column- and row-grid lines. For the `footer` area, the `footer-start` and `footer-end` names were automatically assigned to its grid lines.

Grid lines extend throughout the whole grid area, so a lot of these names are coincident. Figure 26 shows the naming of the lines created by the following template:

```
    grid-template-areas:
        "header    header    header    header"
```

```
"left      ...       ...       right"
"footer    footer    footer    footer";
```

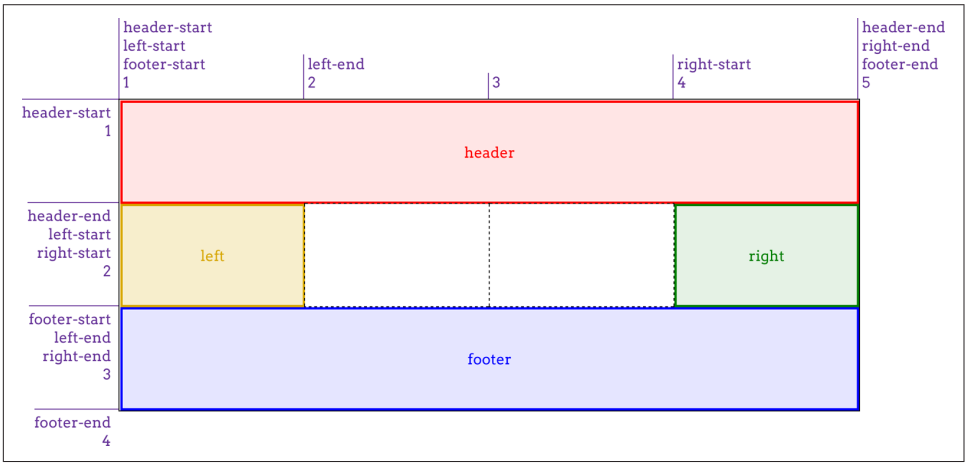


Figure 26. Implicit grid-line names made explicit

Now let's mix it up even more by adding a couple of explicit grid-line names to our CSS. Given the following rules, the first column-grid line in the grid would add the name `begin`, and the second row-grid line in the grid would add the name `content`.

```
#grid {display: grid;
    grid-template-areas:
        "header  header  header  header"
        "left    ...     ...     right"
        "footer  footer  footer  footer";
    grid-template-columns: [begin] 1fr 20em 20em 1fr 1fr;
    grid-template-rows: 40px [content] 1fr 3em 20px;}
```

Again: those grid-line names are *added* to the implicit grid-line names created by the named areas. Interestingly enough, grid-line names never replace other grid-line names. Instead, they just keep piling up.

Even more interesting, this implicit-name mechanism runs in reverse. Suppose you don't use `grid-template-areas` at all, but instead set up some named grid lines like so, as illustrated in Figure 27:

```
grid-template-columns:
     [header-start footer-start] 1fr
     [content-start] 1fr [content-end] 1fr
     [header-end footer-end];
grid-template-rows:
    [header-start] 3em
    [header-end content-start] 1fr
    [content-end footer-start] 3em
    [footer-end];
```

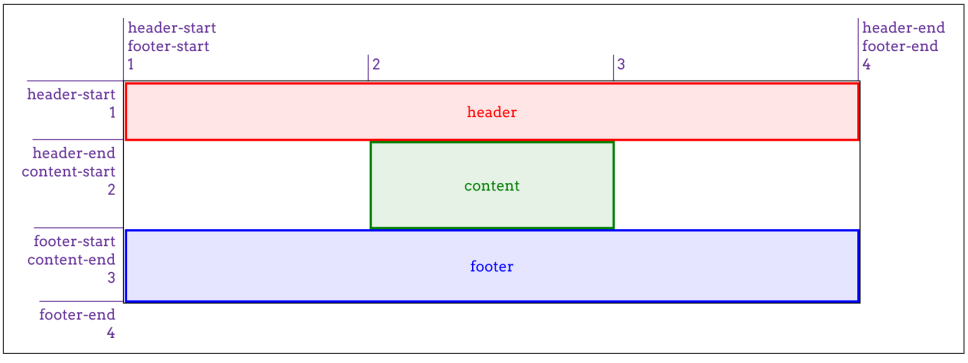


Figure 27. Implicit grid-area names made explicit

Because the grid lines use the form of `name-start`/`name-end`, the grid areas they define are implicitly named. To be frank, it's clumsier than doing it the other way, but the capability is there in case you ever want it.

Bear in mind that you don't need all four grid lines to be named in order to create a named grid area, though you probably do need them all to create a named grid area where you want it to be. Consider the following simple example:

```
grid-template-columns: 1fr [content-start] 1fr [content-end] 1fr;
grid-template-rows: 3em 1fr 3em;
```

This will still create a grid area named `content`. It's just that the named area will be placed into a new row after all the defined rows. What's odd is that an extra, empty row will appear after the defined rows but before the row containing `content`. This has been confirmed to be the intended behavior. Thus, if you try to create a named area by naming the grid lines and miss one or more of them, then your named area will effectively hang off to one side of the grid instead of being a part of the overall grid structure.

So, again, you should probably stick to explicitly naming grid areas and letting the grid-line names happen implicitly, as opposed to the other way around.

Attaching Elements to the Grid

Believe it or not, we've gotten this far without talking about how grid items are actually attached to a grid, once it's been defined.

Using Column and Row Lines

There are a couple of ways to go about this, depending on whether you want to refer to grid lines or grid areas. We'll start with four simple properties that attach an element to grid lines.

grid-row-start, grid-row-end, grid-column-start, grid-column-end

Values:	`auto` | <custom-ident> | [<integer> && <custom-ident>?] | [`span` && [<integer> || <custom-ident>]]
Initial value:	`auto`
Applies to:	Grid items and absolutely positioned elements, if their containing block is a grid container
Inherited:	No
Computed value:	As declared

What these properties do is let you say, "I want the edge of the element to be attached to grid line such-and-so." As with so much of grid layout, it's a lot easier to show than to describe, so ponder the following styles and their result (Figure 28):

```
.grid {display: grid; width: 50em;
    grid-template-rows: repeat(5, 5em);
    grid-template-columns: repeat(10, 5em);}
.one {
    grid-row-start: 2; grid-row-end: 4;
    grid-column-start: 2; grid-column-end: 4;}
.two {
    grid-row-start: 1; grid-row-end: 3;
    grid-column-start: 5; grid-column-end: 10;}
.three {
    grid-row-start: 4;
    grid-column-start: 6;}
```

Here, we're using grid-line numbers to say where and how the elements should be placed within the grid. Column numbers count from left to right, and row numbers from top to bottom. Note that if you omit ending grid lines, as was the case for `.three`, then the next grid lines in sequence are used for the end lines.

Thus, the rule for `.three` in the previous example is exactly equivalent to the following:

```
.three {
    grid-row-start: 4; grid-row-end: 5;
    grid-column-start: 6; grid-column-end: 7;}
```

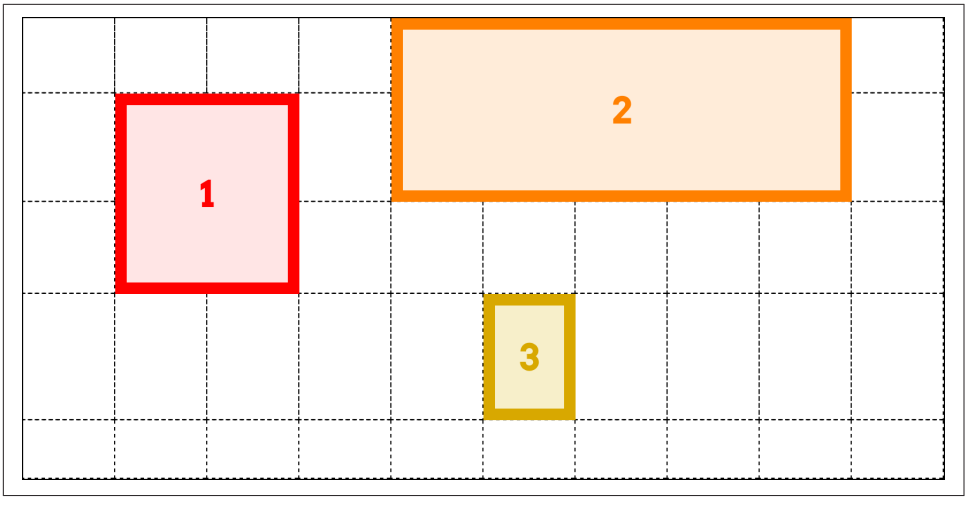


Figure 28. Attaching elements to grid lines

There's another way to say that same thing, as it happens: you could replace the ending values with `span 1`, or even just plain `span`, like this:

```
.three {
    grid-row-start: 4; grid-row-end: span 1;
    grid-column-start: 6; grid-column-end: span;}
```

If you supply `span` with a number, you're saying, "span across this many grid cells." So we could rewrite our earlier example like this, and get exactly the same result:

```
#grid {display: grid;
    grid-template-rows: repeat(5, 5em);
    grid-template-columns: repeat(10, 5em);}
.one {
    grid-row-start: 2; grid-row-end: span 2;
    grid-column-start: 2; grid-column-end: span 2;}
.two {
    grid-row-start: 1; grid-row-end: span 2;
    grid-column-start: 5; grid-column-end: span 5;}
.three {
    grid-row-start: 4; grid-row-end: span 1;
    grid-column-start: 6; grid-column-end: span;}
```

If you leave out a number for `span`, it's set to be `1`. You can't use zero or negative numbers for `span`; only positive integers.

An interesting feature of `span` is that you can use it for both ending *and* starting grid lines. The precise behavior of `span` is that it counts grid lines in the direction "away" from the grid line where it starts. In other words, if you define a start grid line and set the ending grid line to be a `span` value, it will search toward the end of the grid. Con-

versely, if you define an ending grid line and make the start line a `span` value, then it will search toward the start of the grid.

That means the following rules will have the result shown in Figure 29:

```
#grid {display: grid;
    grid-rows: repeat(4, 2em); grid-columns: repeat(5, 5em);}
.box01 {grid-row-start: 1; grid-column-start: 3; grid-column-end: span 2;}
.box02 {grid-row-start: 2; grid-column-start: span 2; grid-column-end: 3;}
.box03 {grid-row-start: 3; grid-column-start: 1; grid-column-end: span 5;}
.box04 {grid-row-start: 4; grid-column-start: span 1; grid-column-end: 5;}
```

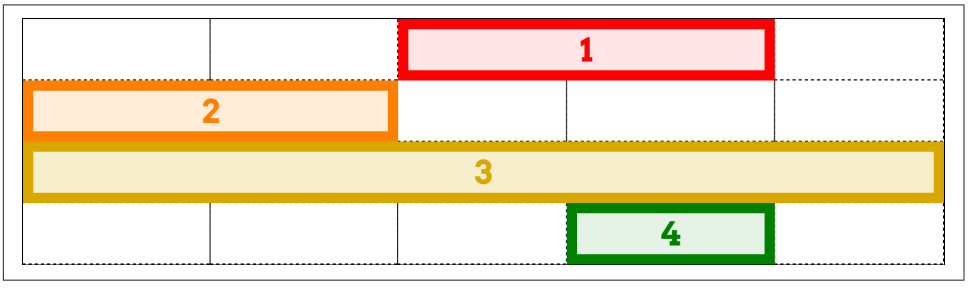


Figure 29. Spanning grid lines

In contrast to `span` numbering, you aren't restricted to positive integers for your actual grid-line values. Negative numbers will simply count backward from the end of explicitly defined grid lines. Thus, to place an element into the bottom-right grid cell of a defined grid, regardless of how many columns or rows it might have, you can just say this:

```
grid-column-start: -1;
grid-row-start: -1;
```

Note that this doesn't apply to any implicit grid tracks, a concept we'll get to in a bit, but only to the grid lines you explicitly define via one of the `grid-template-*` properties (e.g., `grid-template-rows`).

We aren't restricted to grid-line numbers, as it happens. If there are named grid lines, we can refer to those instead of (or in conjunction with) numbers. If you have multiple instances of a grid-line name, then you can use numbers to identify which instance of the grid-line name you're talking about. Thus, to start from the fourth instance of a row grid named `mast-slice`, you can say `mast-slice 4`. Take a look at the following, illustrated in Figure 30, for an idea of how this works.

```
#grid {display: grid;
    grid-template-rows: repeat(5, [R] 4em);
    grid-template-columns: 2em repeat(5, [col-A] 5em [col-B] 5em) 2em;}
.one {
    grid-row-start: R 2; grid-row-end: 5;
    grid-column-start: col-B; grid-column-end: span 2;}
```

```
.two {
    grid-row-start: R; grid-row-end: span R 2;
    grid-column-start: col-A 3; grid-column-end: span 2 col-A;}
.three {
    grid-row-start: 9;
    grid-column-start: col-A -2;}
```

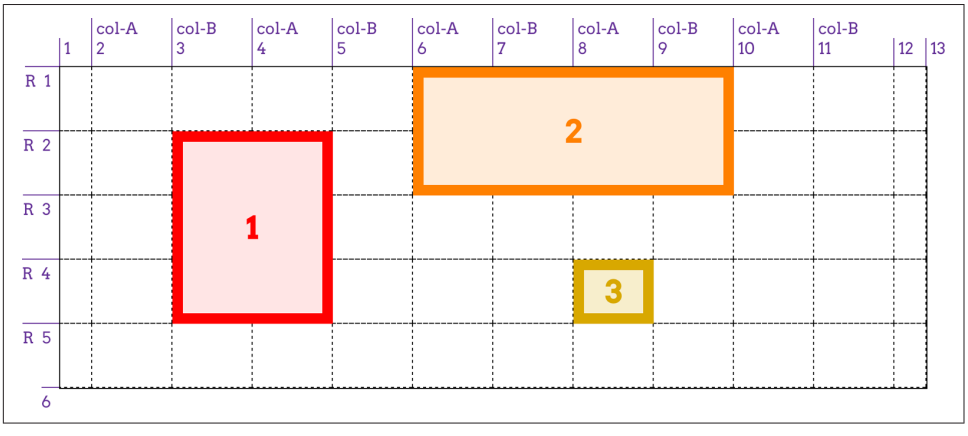


Figure 30. Attaching elements to named grid lines

Notice how `span` changes when we add a name: where we said `span 2 col-A`, that caused the grid item to span from its starting point (the third `col-A`) across another `col-A` and end at the `col-A` after that. This means the grid item actually spans four grid cells, since `col-A` appears on every other column grid line.

Again, negative numbers count backward from the end of a sequence, so `col-A -2` gets us the second-from-last instance of a grid line named `col-A`. Because there are no end-line values declared for `.three`, they're both set to `span 1`. That means the following is exactly equivalent to the `.three` in the previous example:

```
.three {
    grid-row-start: 9; grid-row-end: span 1;
    grid-column-start: col-A -2; grid-row-end: span 1;}
```

There's an alternative way to use names with named grid lines—specifically, the named grid lines that are implicitly created by grid areas. For example, consider the following styles, illustrated in Figure 31:

```
grid-template-areas:
    "header     header    header    header"
    "leftside   content   content   rightside"
    "leftside   footer    footer    footer";
#masthead {grid-row-start: header;
        grid-column-start: header; grid-row-end: header;}
#sidebar {grid-row-start: 2; grid-row-end: 4;
        grid-column-start: leftside / span 1;}
```

```
#main {grid-row-start: content; grid-row-end: content;
       grid-column-start: content;}
#navbar {grid-row-start: rightside; grd-row-end: 3;
       grid-column-start: rightside;}
#footer {grid-row-start: 3; grid-row-end: span 1;
       grid-column-start: footer; grid-row-end: footer;}
```

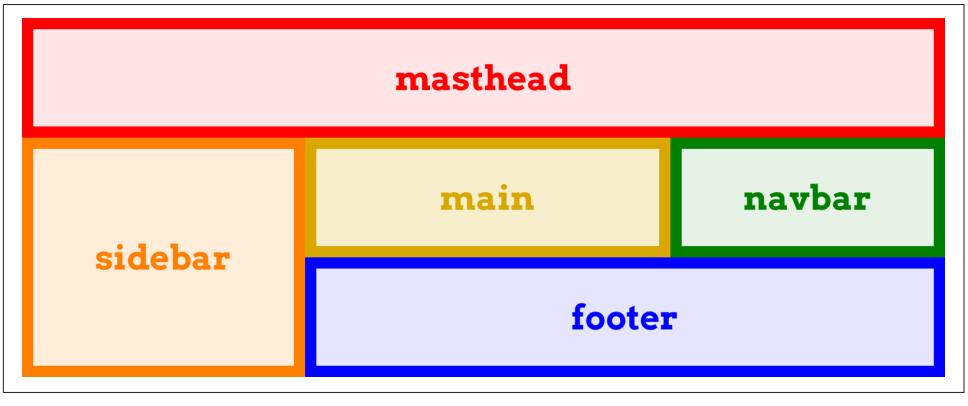


Figure 31. Another way of attaching elements to named grid lines

What happens if you supply a custom identifier (i.e., a name you defined) is that the browser looks for a grid line with that name *plus* either `-start` or `-end` added on, depending on whether you're assigning a start line or an end line. Thus, the following are equivalent:

```
grid-column-start: header; grid-column-end: header;
grid-column-start: header-start; grid-column-end: header-end;
```

This works because, as was mentioned with `grid-template-areas`, explicitly creating a grid area implicitly creates the named `-start` and `-end` grid lines that surround it.

The final value possibility, `auto`, is kind of interesting. According to the Grid Layout specification, if one of the grid-line start/end properties is set to `auto`, that indicates "auto-placement, an automatic span, or a default span of one." In practice, what this tends to mean is that the grid line that gets picked is governed by the *grid flow*, a concept we have yet to cover (but will soon!). For a start line, `auto` usually means that the next available column or row line will be used. For an end line, `auto` usually means a one-cell span. In both cases, the word "usually" is used intentionally: as with any automatic mechanism, there are no absolutes.

Row and Column Shorthands

There are two shorthand properties that allow you to more compactly attach an element to grid lines.

grid-row, grid-column

Values:	<grid-line> [/ <grid-line>]?
Initial value:	`auto`
Applies to:	Grid items and absolutely positioned elements, if their containing block is a grid container
Inherited:	No
Computed value:	As declared
Note:	Is equivalent to the value syntax for `grid-column-start` et al.

The primary benefit of these properties is that they make it a lot simpler to declare the start and end grid lines to be used for laying out a grid item. For example:

```
#grid {display: grid;
    grid-template-rows: repeat(10, [R] 1.5em);
    grid-template-columns: 2em repeat(5, [col-A] 5em [col-B] 5em) 2em;}
.one {
    grid-row: R 3 / 7;
    grid-column: col-B / span 2;}
.two {
    grid-row: R / span R 2;
    grid-column: col-A 3 / span 2 col-A;}
.three {
    grid-row: 9;
    grid-column: col-A -2;}
```

That's a whole lot easier to read than having each start and end value in its own property, honestly. Other than being more compact, the behavior of these properties is more or less what you'd expect. If you have two bits separated by a solidus, the first part defines the starting grid line and the second part defines the ending grid line.

If you have only one bit with no solidus, it defines the starting grid line. The ending grid line depends on what you said for the starting line. If you supply a name for the starting grid line, then the ending grid line is given that same name. Thus, the following are equivalent:

```
grid-column: col-B;
grid-column: col-B / col-B;
```

That will, of course, span from one instance of that grid-line name to the next, regardless of how many grid cells are spanned.

In cases where only one name is given for `grid-column` or `grid-row`, it is also used for the second (the end line). If a single number is given, then the second number (the end line) is set to `auto`. That means the following pairs are equivalent:

```
grid-row: 2;
grid-row: 2 / auto;

grid-column: header;
grid-column: header / header;
```

There's a subtle behavior built into the handling of grid-line names in `grid-row` and `grid-column` that pertains to implicitly named grid lines. If you recall, defining a named grid area creates `-start` and `-end` grid lines. That is, given a grid area with a name of `footer`, there are implicitly created `footer-start` grid lines to its top and left, and `footer-end` grid lines to its bottom and right.

In that case, if you refer to those grid lines by the area's name, the element will still be placed properly. Thus, the following styles have the result shown in Figure 32:

```
#grid {display: grid;
    grid-template-areas:
        "header header"
        "sidebar content"
        "footer footer";
     grid-template-rows: auto 1fr auto;
     grid-template-columns: 25% 75%;}
#header {grid-row: header / header; grid-column: header;}
#footer {grid-row: footer; grid-column: footer-start / footer-end;}
```

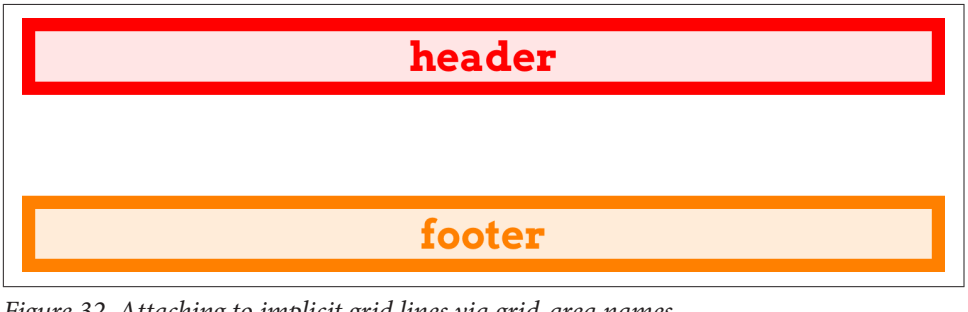


Figure 32. Attaching to implicit grid lines via grid-area names

You can always explicitly refer to the implicitly named grid lines, but if you just refer to the grid area, things still work out. If you refer to a grid-line name that doesn't correspond to a grid area, then it falls back to the behavior discussed previously. In detail, it's the same as saying `line-name 1`, so the following two are equivalent:

```
grid-column: jane / doe;
grid-column: jane 1 / doe 1;
```

This is why it's risky to name grid lines the same as grid areas. Consider the following:

```
    grid-template-areas:
        "header header"
        "sidebar content"
        "footer footer"
        "legal legal";
    grid-template-rows: auto 1fr [footer] auto [footer];
    grid-template-columns: 25% 75%;
```

This explicitly sets grid lines named `footer` above the "footer" row and below the "legal" row…and now there's trouble ahead. Suppose we add this:

```
#footer {grid-column: footer; grid-row: footer;}
```

For the column lines, there's no problem. `footer` gets expanded to `footer / footer`. The browser looks for a grid area with that name and finds it, so it translates `footer / footer` to `footer-start / footer-end`. The `#footer` element is attached to those implicit grid lines.

For `grid-row`, everything starts out the same. `footer` becomes `footer / footer`, which is translated to `footer-start / footer-end`. But that means the `#footer` will only be as tall as the "footer" row. It will *not* stretch to the second explicitly named `footer` grid line below the "legal" row, because the translation of `footer` to `footer-end` (due to the match between the grid-line name and the grid-area name) takes precedence.

The upshot of all this: it's generally a bad idea to use the same name for grid areas and grid lines. You might be able to get away with it in some scenarios, but you're almost always better off keeping your line and area names distinct, so as to avoid tripping over name-resolution conflicts.

The Implicit Grid

Up to this point, we've concerned ourselves solely with explicitly defined grids: we've talked about the row and column tracks we define via properties like `grid-template-columns`, and how to attach grid items to the cells in those tracks.

But what happens if we try to place a grid item, or even just part of a grid item, beyond that explicitly created grid? For example, consider the following simple grid:

```
#grid {display: grid;
    grid-template-rows: 2em 2em;
    grid-template-columns: repeat(6, 4em);}
```

Two rows, six columns. Simple enough. But suppose we define a grid item to sit in the first column and go from the first grid line to the fourth:

```
.box01 {grid-column: 1; grid-row: 1 / 3;}
```

Now what? There are only two rows bounded by three grid lines, and we've told the browser to go beyond that, from line 1 to line 4.

What happens is that another row line is created to handle the situation. This grid line, and the new row track it creates, are both part of the *implicit grid*. Here are a few examples of grid items that create implicit grid lines (and tracks) and how they're laid out (see Figure 33):

```
.box01 {grid-column: 1; grid-row: 1 / 3;}
.box02 {grid-column: 2; grid-row: 3 / span 2;}
.box03 {grid-column: 3; grid-row: span 2 / 3;}
.box04 {grid-column: 4; grid-row: span 2 / 5;}
.box05 {grid-column: 5; grid-row: span 4 / 5;}
.box06 {grid-column: 6; grid-row: -1 / span 3;}
.box07 {grid-column: 7; grid-row: span 3 / -1;}
```

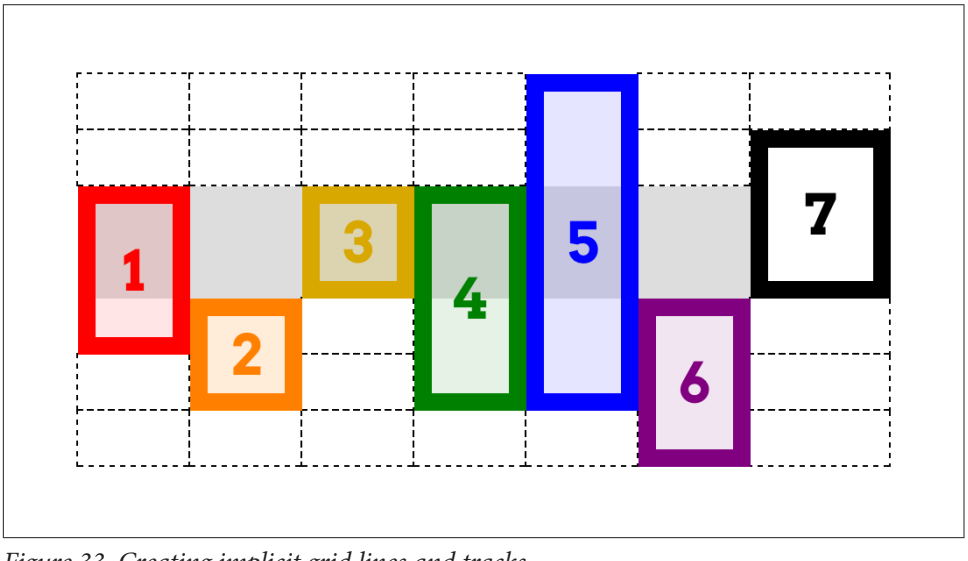


Figure 33. Creating implicit grid lines and tracks

There's a lot going on there, so let's break it down. First off, the explicit grid is represented by the light-gray box behind the various numbered boxes; all the dashed lines represent the implicit grid.

So, what about those numbered boxes? `box1` adds an extra grid line after the end of the explicit grid, as we discussed before. `box2` starts on the last line of the explicit grid, and spans forward two lines, so it adds yet another implicit grid line. `box3` ends on

the last explicit grid line (line 3) and spans back two lines, thus starting on the first explicit grid line.

box4 is where things really get interesting. It ends on the fifth row line, which is to say the second implicit grid line. It spans back three lines—and yet, it still starts on the same grid line as box3. This happens because spans have to start counting within the explicit grid. Once they start, they can continue on into the implicit grid (as happened with box2), but they *cannot* start counting within the implicit grid.

Thus, box4 ends on row-line 5, but its span starts with grid-line 3 and counts back two lines (`span 2`) to arrive at line 1. Similarly, box5 ends on line 5, and spans back four lines, which means it starts on row-line –2. Remember: span counting must *start* in the explicit grid. It doesn't have to end there.

After those, box6 is pretty straightforward: it starts on the last explicit row line (line 3), and spans out to the sixth row line—adding yet another implicit row line. The point of having it here is to show that negative grid-line references are with respect to the explicit grid, and count back from its end. They do *not* refer to negatively indexed implicit lines that are placed before the start of the explicit grid.

If you want to start an element on an implicit grid line before the explicit grid's start, then the way to do that is shown by box7: put its end line somewhere in the explicit grid, and span back past the beginning of the explicit grid. And you may have noticed: box7 occupies an implicit column track. The original grid was set up to create six columns, which means seven column lines, the seventh being the end of the explicit grid. When box7 was given `grid-column: 7`, that was equivalent to `grid-column: 7 / span 1` (since a missing end line is always assumed to be `span 1`). That necessitated the creation of an implicit column line in order to hold the grid item in the implicit seventh column.

Now let's take those principles and add named grid lines to the mix. Consider the following, illustrated in Figure 34:

```
#grid {display: grid;
    grid-template-rows: [begin] 2em [middle] 2em [end];
    grid-template-columns: repeat(5, 5em);}
.box01 {grid-column: 1; grid-row: 2 / span end 2;}
.box02 {grid-column: 2; grid-row: 2 / span final;}
.box03 {grid-column: 3; grid-row: 1 / span 3 middle;}
.box04 {grid-column: 4; grid-row: span begin 2 / end;}
.box05 {grid-column: 5; grid-row: span 2 middle / begin;}
```

What you can see at work there, in several of the examples, is what happens with grid-line names in the implicit grid: every implicitly created line has the name that's being hunted. Take box2, for example. It's given an end line of `final`, but there is no line with that name. Thus the span-search goes to the end of the explicit grid and,

having not found the name it's looking for, creates a new grid line, to which it attaches the name `final`.

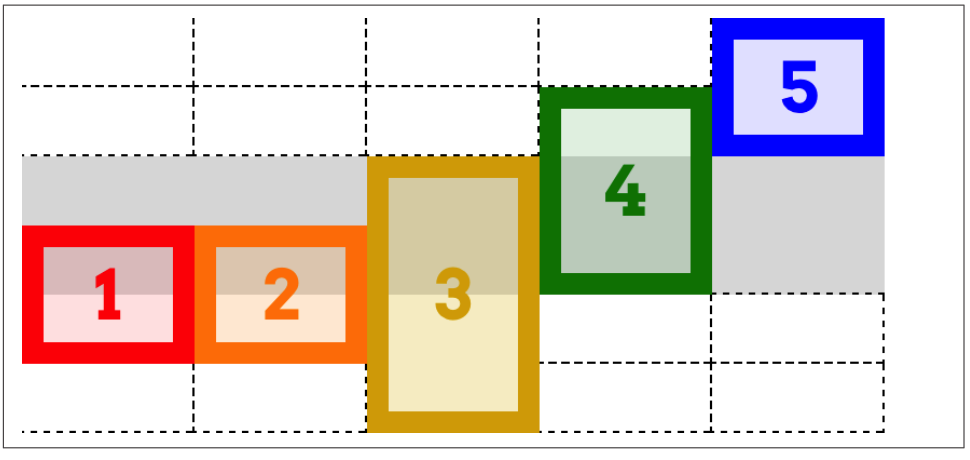


Figure 34. Named implicit grid lines and tracks

Similarly, `box3` starts on the first explicit row line, and then needs to span three `middle` named lines. It searches forward and finds one, then goes looking for the other two. Not finding any, it attaches the name `middle` to the first implicit grid line, and then does the same for the second implicit grid line. Thus, it ends two implicit grid lines past the end of the explicit grid.

When you get right down to it, the implicit grid is a delightfully baroque fallback mechanism. It's generally best practice to stick to the explicit grid, and to make sure the explicit grid covers everything you want to do. If you find you need another row, don't just run off the edge of the grid—adjust your grid template's values instead!

Error Handling

There are a few cases that need to be covered, as they fall under the general umbrella of "what grids do when things go pear-shaped."

First, what if you accidentally put the start line after the end line? Say, something like this:

```
grid-row-start: 5;
grid-row-end: 2;
```

All that happens is probably what was meant in the first place: the values are swapped. Thus, you end up with this:

```
grid-row-start: 2;
grid-row-end: 5;
```

Second, what if both the start and the end lines are declared to be spans of some variety? For example:

```
grid-column-start: span;
grid-column-end: span 3;
```

If this happens, the end value is dropped and replaced with `auto`. That means you'd end up with this:

```
grid-column-start: span;  /* 'span' is equal to 'span 1' */
grid-column-end: auto;
```

That would cause the grid item to have its ending edge placed automatically, according to the current grid flow (a subject we'll soon explore), and the starting edge to be placed one grid line earlier.

Third, what if the only thing directing placement of the grid item is a named span? In other words:

```
grid-row-start: span footer;
grid-row-end: auto;
```

This is not permitted, so the `span footer` in this case is replaced with `span 1`.

Using Areas

Attaching by row lines and column lines is great, but what if you could refer to a grid area with a single property? Behold: `grid-area`.

grid-area

Values:	<grid-line> [/ <grid-line>]{0,3}
Initial value:	See individual properties
Applies to:	Grid items and absolutely positioned elements, if their containing block is a grid container
Inherited:	No
Computed value:	As declared
Note:	Is equivalent to the value syntax for `grid-column-start` et al.

Let's start with the easier use of `grid-area`: assigning an element to a previously defined grid area. Makes sense, right? Let's bring back our old friend `grid-template-areas`, put it together with `grid-area` and some markup, and see what magic results (as shown in Figure 35):

```
#grid {display: grid;
    grid-template-areas:
        "header    header   header   header"
        "leftside  content  content  rightside"
        "leftside  footer   footer   footer";}
#masthead {grid-area: header;}
#sidebar {grid-area: leftside;}
#main {grid-area: content;}
#navbar {grid-area: rightside;}
#footer {grid-area: footer;}

<div id="grid">
    <div id="masthead">…</div>
    <div id="main">…</div>
    <div id="navbar">…</div>
    <div id="sidebar">…</div>
    <div id="footer">…</div>
</div>
```

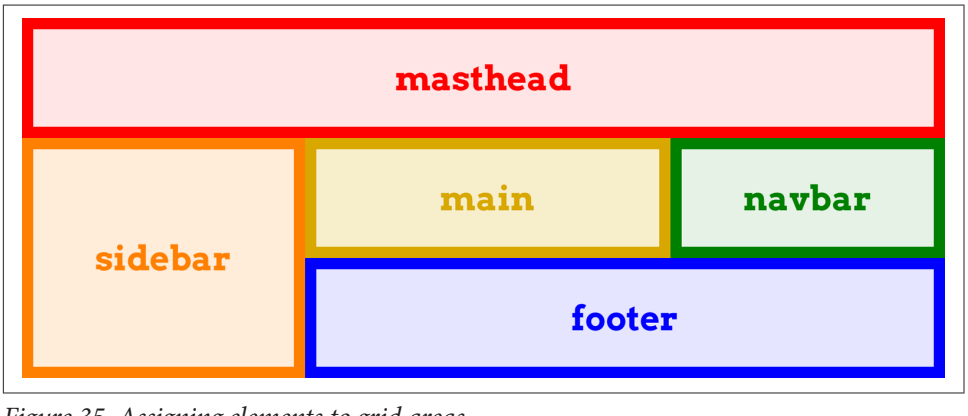


Figure 35. Assigning elements to grid areas

That's all it takes: set up some named grid areas to define your layout, and then drop grid items into them with `grid-area`. So simple, and yet so powerful.

As you might have noticed, the sizing of the column and row tracks was omitted from that CSS. This was done entirely for clarity's sake. In an actual design, the rule probably would look more like this:

```
grid-template-areas:
    "header   header  header  header"
    "leftside content content rightside"
    "leftside footer  footer  footer";
```

```
    grid-template-rows: 200px 1fr 3em;
    grid-template-columns: 20em 1fr 1fr 10em;
```

There is another way to use `grid-area` that refers to grid lines instead of grid areas. Fair warning: it's likely to be confusing at first, for a couple of reasons.

Here's an example of a grid template that defines some grid lines, and some `grid-area` rules that reference the lines, as illustrated in Figure 36:

```
#grid {display: grid;
    grid-template-rows:
        [r1-start] 1fr [r1-end r2-start] 2fr [r2-end];
    grid-template-columns:
        [col-start] 1fr [col-end main-start] 1fr [main-end];}
.box01 {grid-area: r1 / main / r1 / main;}
.box02 {grid-area: r2-start / col-start / r2-end / main-end;}
.box03 {grid-area: 1 / 1 / 2 / 2;}
```

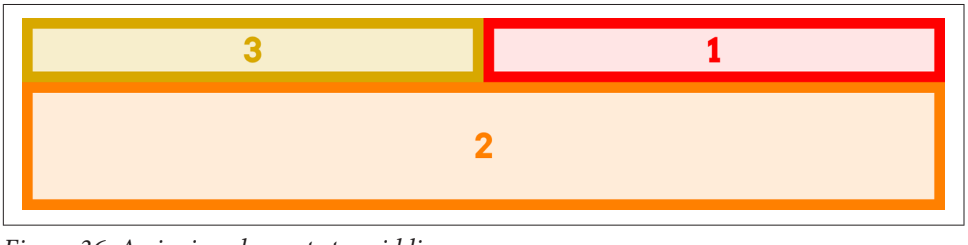


Figure 36. Assigning elements to grid lines

As you can see, the elements were placed as directed. Note the ordering of the grid-line values, however. They're listed in the order `row-start`, `column-start`, `row-end`, `column-end`. If you diagram that in your head, you'll quickly realize that the values go anticlockwise around the grid item—the exact opposite of the TRBL (Top, Right, Bottom, Left) pattern we're used to from margins, padding, borders, and so on. Furthermore, this means the column and row references are not grouped together, but are instead split up.

Yes, this is intentional. No, I don't know why.

If you supply fewer than four values, the missing values are taken from those you do supply. If there are only three values, then the missing `grid-column-end` is the same as `grid-column-start` if it's a name; if the start line is a number, the end line is set to `auto`. The same holds true if you give only two values, except that the now-missing `grid-row-end` is copied from `grid-row-start` if it's a name; otherwise, it's set to `auto`.

From that, you can probably guess what happens if only one value is supplied: if it's a name, use it for all four values; if it's a number, the rest are set to `auto`.

This one-to-four replication pattern is actually how giving a single grid-area name translates into having the grid item fill that area. The following are equivalent:

```
grid-area: footer;
grid-area: footer / footer / footer / footer;
```

Now recall the behavior discussed in the previous section about `grid-column` and `grid-row`: if a grid line's name matches the name of a grid area, then it's translated into a `-start` or `-end` variant, as appropriate. That means the previous example is translated to the following:

```
grid-area: footer-start / footer-start / footer-end / footer-end;
```

And that's how a single grid-area name causes an element to be placed into the corresponding grid area.

Grid Item Overlap

One thing we've been very careful to do in our grid layouts thus far is to avoid overlap. Rather like positioning, it's absolutely (get it?) possible to make grid items overlap each other. Let's take a simple case, illustrated in Figure 37:

```
#grid {display: grid;
    grid-template-rows: 50% 50%;
    grid-template-columns: 50% 50%;}
.box01 {grid-area: 1 / 1 / 2 / 3;}
.box02 {grid-area: 1 / 2 / 3 / 2;}
```

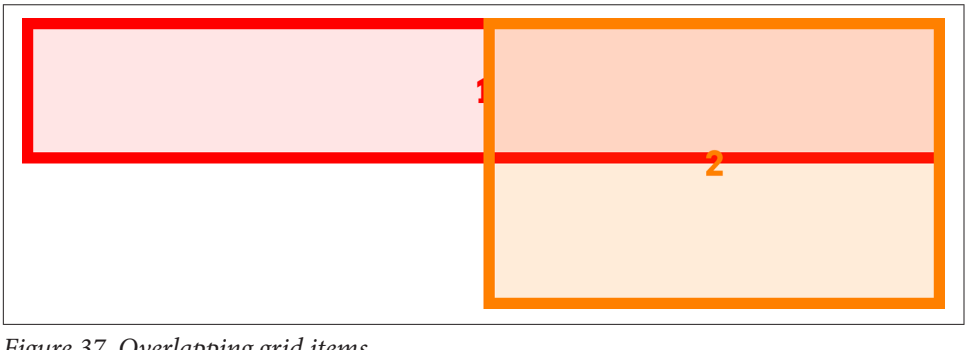


Figure 37. Overlapping grid items

Thanks to the grid numbers that were supplied, the two grid items overlap in the upper-right grid cell. Which is on top of the other depends on the layering behavior discussed later, but for now, just take it as given that they do layer when overlapping.

Overlap isn't restricted to situations involving raw grid numbers, of course. In the following case, the sidebar and the footer will overlap, as shown in Figure 38. (Assuming the footer comes later than the sidebar in the markup, then in the absence of other styles, the footer will be on top of the sidebar.)

```
#grid {display: grid;
    grid-template-areas:
        "header header"
        "sidebar content"
        "footer footer";}
#header {grid-area: header;}
#sidebar {grid-area: sidebar / sidebar / footer-end / sidebar;}
#footer {grid-area: footer;}
```

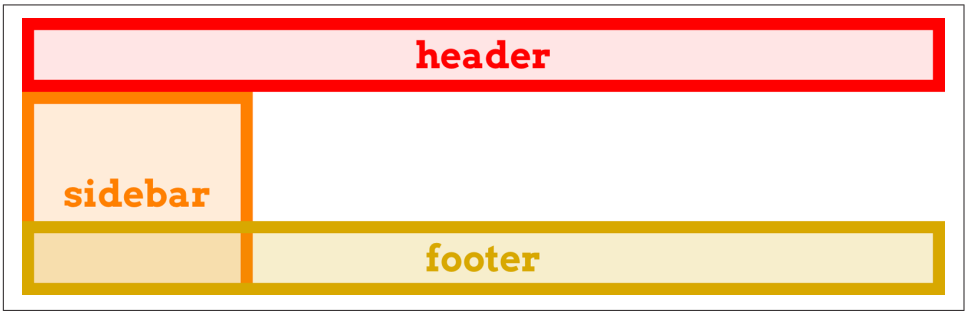


Figure 38. Overlapping sidebar and footer

I bring this up in part to warn you about the possibility of overlap, and also to serve as a transition to the next topic. It's a feature that sets grid layout apart from positioning, in that it can sometimes help avoid overlap: the concept of *grid flow*.

Grid Flow

For the most part, we've been explicitly placing grid items on the grid. If items aren't explicitly placed, then they're automatically placed into the grid. Following the grid flow in effect, an item is placed in the first area that will fit it. The simplest case is simply filling a grid track in sequence, one grid item after another, but things can get a lot more complex than that, expecially if there is a mixture of explicitly and automatically placed grid items—the latter must work around the former.

There are primarily two grid-flow models, *row-first* and *column-first*, though you can enhance either by specifying a *dense* flow. All this is done with the property `grid-auto-flow`.

grid-auto-flow

Values: [`row` | `column`] || `dense`

Initial value: `row`

Applies to:	Grid containers
Inherited:	No
Computed value:	As declared

To see how these values work, consider the following markup:

```
<ol id="grid">
<li>1</li>
<li>2</li>
<li>3</li>
<li>4</li>
<li>5</li>
</ol>
```

To that markup, let's apply the following styles:

```
#grid {display: grid; width: 45em; height: 8em;
    grid-auto-flow: row;}
#grid li {grid-row: auto; grid-column: auto;}
```

Assuming a grid with a column line every 15 ems and a row line every 4 ems, we get the result shown in Figure 39.

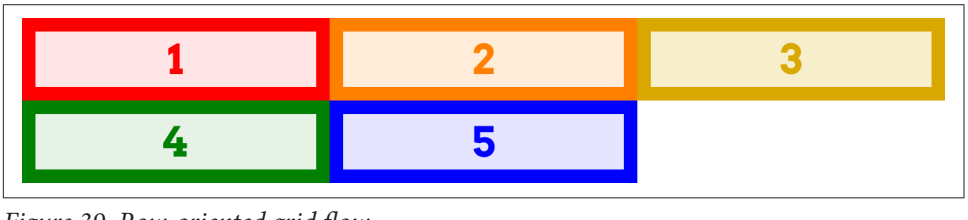


Figure 39. Row-oriented grid flow

This probably seems pretty normal, the same sort of thing you'd get if you floated all the boxes, or if all of them were inline blocks. That's why `row` is the default value. Now, let's try switching the `grid-auto-flow` value to `column`, as shown in Figure 40:

```
#grid {display: grid; width: 45em; height: 8em;
    grid-auto-flow: column;}
#grid li {grid-row: auto; grid-column: auto;}
```

So with `grid-auto-flow: row`, each row is filled in before starting on the next row. With `grid-auto-flow: column`, each column is filled first.

Figure 40. Column-oriented grid flow

What needs to be stressed here is that the list items weren't explicitly sized. By default, they were resized to attach to the defined grid lines. This can be overridden by assigning explicit sizing to the elements. For example, if we make the list items be 7 ems wide and 1.5 ems tall, we'll get the result shown in Figure 41:

```
#grid {display: grid; width: 45em; height: 8em;
    grid-auto-flow: column;}
#grid li {grid-row: auto; grid-column: auto;
    width: 7em; height: 1.5em;}
```

Figure 41. Explicitly sized grid items

If you compare that to the previous figure, you'll see that the corresponding grid items start in the same place in each figure; they just don't end in the same places. This illustrates that what's really placed in grid flow is grid areas, to which the grid items are then attached.

This is important to keep in mind if you auto-flow elements that are wider than their assigned column or taller than their assigned row, as can very easily happen when turning images or other intrinsically sized elements into grid items. Let's say we want to put a bunch of images, each a different size, into a grid that's set up to have a column line every 50 horizontal pixels, and a row line every 50 vertical pixels. This grid is illustrated in Figure 42, along with the results of flowing a series of images into that grid by either row or column.

```
#grid {display: grid;
    grid-template-rows: repeat(3, 50px);
    grid-template-columns: repeat(4, 50px);
    grid-auto-rows: 50px;
    grid-auto-columns: 50px;
}
img {grid-row: auto; grid-column: auto;}
```

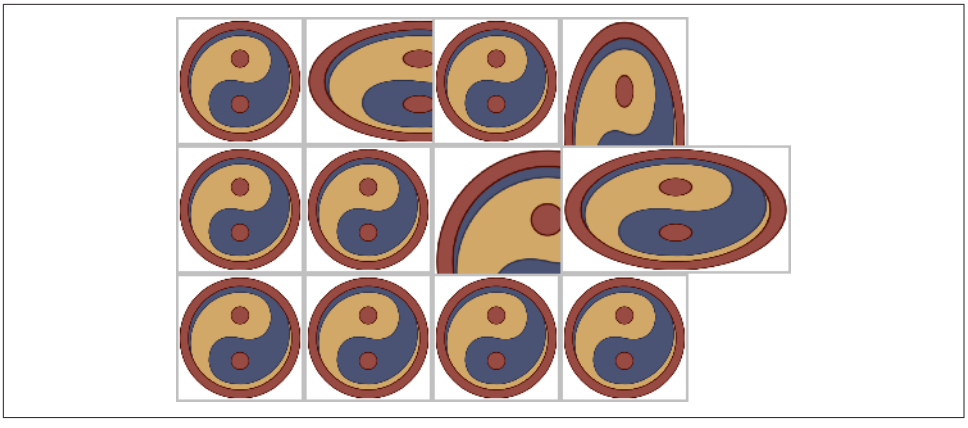


Figure 42. Flowing images in grids

Notice how some of the images overlap others? That's because each image is attached to the next grid line in the flow, without taking into account the presence of other grid items. We didn't set up images to span more than one grid track when they needed it, so overlap occurred.

This can be managed with class names or other identifiers. We could class images as `tall` or `wide` (or both) and specify that they get more grid tracks. Here's some CSS to add to the previous example, with the result shown in Figure 43:

```
img.wide {grid-column: auto / span 2;}
img.tall {grid-row: auto / span 2;}
```

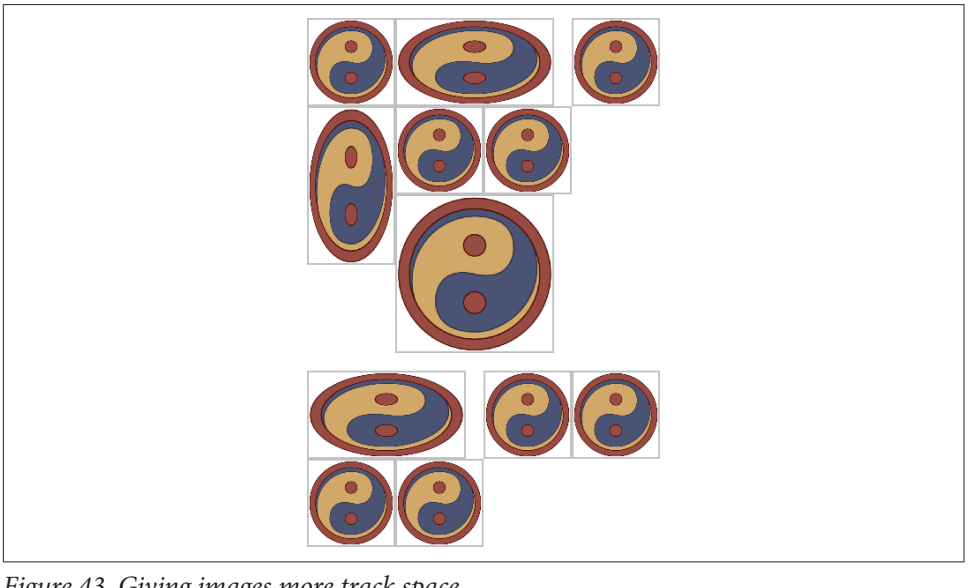


Figure 43. Giving images more track space

This does cause the images to keep spilling down the page, but there's no overlapping.

However, notice how there are gaps in that last grid? That happened because the placement of some grid items across grid lines didn't leave enough room for other items in the flow. In order to illustrate this, and the two flow patterns, more clearly, let's try an example with numbered boxes (Figure 44).

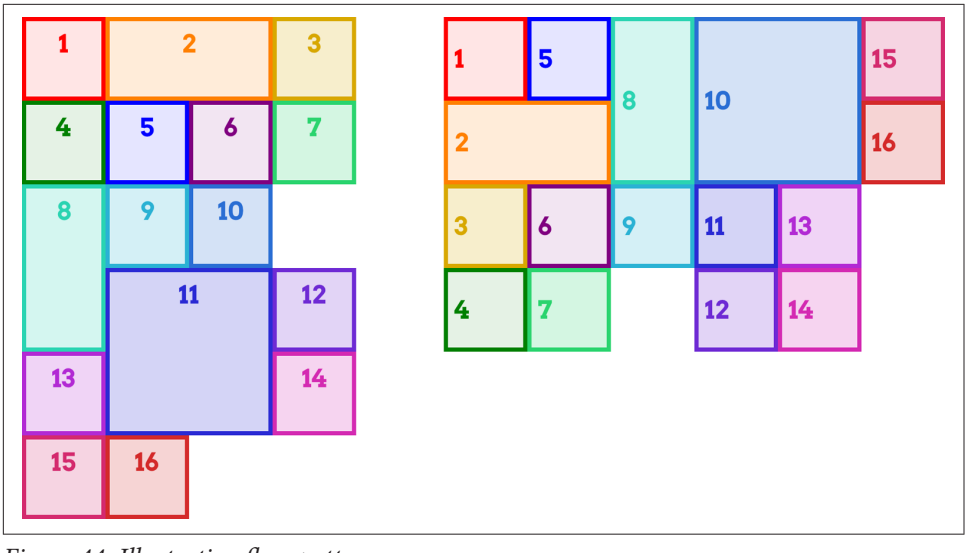


Figure 44. Illustrating flow patterns

Follow across the rows of the first grid, counting along with the numbers. In this particular flow, the grid items are laid out almost as if they were leftward floats. Almost, but not quite: notice that grid item 13 is actually to the left of grid item 11. That would never happen with floats, but it can with grid flow. The way row flow (if we may call it that) works is that you go across each row from left to right, and if there's room for a grid item, you put it there. If a grid cell has been occupied by another grid item, you skip over it. So the cell next to item 10 didn't get filled, because there wasn't room for item 11. Item 13 went to the left of item 11 because there was room for it there when the row was reached.

The same basic mechanisms hold true for column flow, except in this case you work from top to bottom. Thus, the cell below item 9 is empty because item 10 wouldn't fit there. It went into the next column and spanned four grid cells. The items after it, since they were just one grid cell in size, filled in the cells after it in column order.

Grid flow works left-to-right, top-to-bottom in languages that have that writing pattern. In right-to-left languages, such as Arabic and Hebrew, the row-oriented flow would be right-to-left, not left-to-right.

If you were just now wishing for a way to pack grid items as densely as possible, regardless of how that affected the ordering, good news: you can! Just add the keyword `dense` to your `grid-auto-flow` value, and that's exactly what will happen. We can see the result in Figure 45, which shows the results of `grid-auto-flow: row dense` and `grid-auto-flow: dense column` side by side.

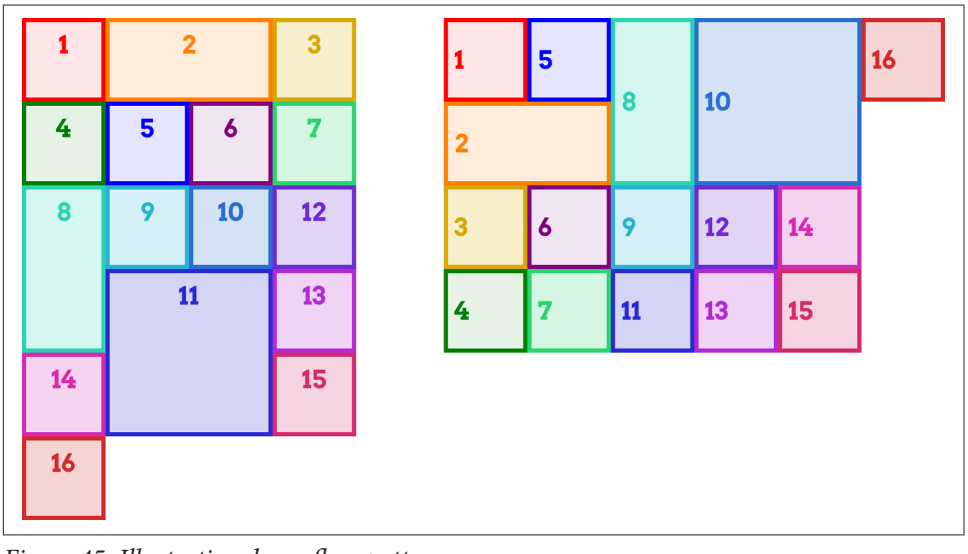


Figure 45. Illustrating dense flow patterns

In the first grid, item 12 appears in the row above item 11 because there was a cell that fit it. For the same reason, item 11 appears to the left of item 10 in the second grid.

In effect, what happens with `dense` grid flow is that for each grid item, the browser scans through the *entire* grid in the given flow direction (`row` or `column`), starting from the flow's starting point (the top-left corner, in LTR—left-to-right—languages), until it finds a place where that grid item will fit. This can make things like photo galleries more compact, and works great as long as you don't have a specific order in which the images need to appear.

Now that we've explored grid flow, I have a confession to make: in order to make the last couple of grid items look right, I included some CSS that I didn't show you. Without it, the items hanging off the edge of the grid would have looked quite a bit different than the other items—much shorter in row-oriented flow, and much narrower in column-oriented flow. We'll see why, and the CSS I used, in the next section.

Automatic Grid Lines

So far, we've almost entirely seen grid items placed into a grid that was explicitly defined. But in the last section we had situations where grid items ran off the edge of the explicitly defined grid. What happens when a grid item goes off the edge? Rows or columns are added as needed to satisfy the layout directives of the items in question (see "The Implicit Grid" on page 37). So, if an item with a row span of 3 is added after the end of a row-oriented grid, three new rows are added after the explicit grid.

By default, these automatically added rows are the absolute minimum size needed. If you want to exert a little more control over their sizing, then `grid-auto-rows` and `grid-auto-columns` are for you.

grid-auto-rows, grid-auto-columns

Values:	<track-breadth> | `minmax(` <track-breadth> `,` <track-breadth> `)`
Initial value:	`auto`
Applies to:	Grid containers
Inherited:	No
Computed value:	Depends on the specific track sizing
Note	<track-breadth> is a stand-in for <length> | <percentage> | <flex> | `min-content` | `max-content`

For any automatically created row or column tracks, you can provide a single track size or a minmaxed pair of track sizes. Let's take a look at a reduced version of the grid-flow example from the previous section: we'll set up a 2 x 2 grid, and try to put five items into it. In fact, let's do it twice: once with `grid-auto-rows`, and once without, as illustrated in Figure 46:

```
.grid {display: grid;
    grid-template-rows: 80px 80px;
    grid-template-columns: 80px 80px;}
#g1 {grid-auto-rows: 80px;}
```

As you can see, without sizing the automatically created row, the grid item is placed in a row that's exactly as tall as the grid item's content, and not a pixel more. It's still

just as wide as the column into which it's placed, because that has a size (`80px`). The row, lacking an explicit height, defaults to `auto`, with the result shown.

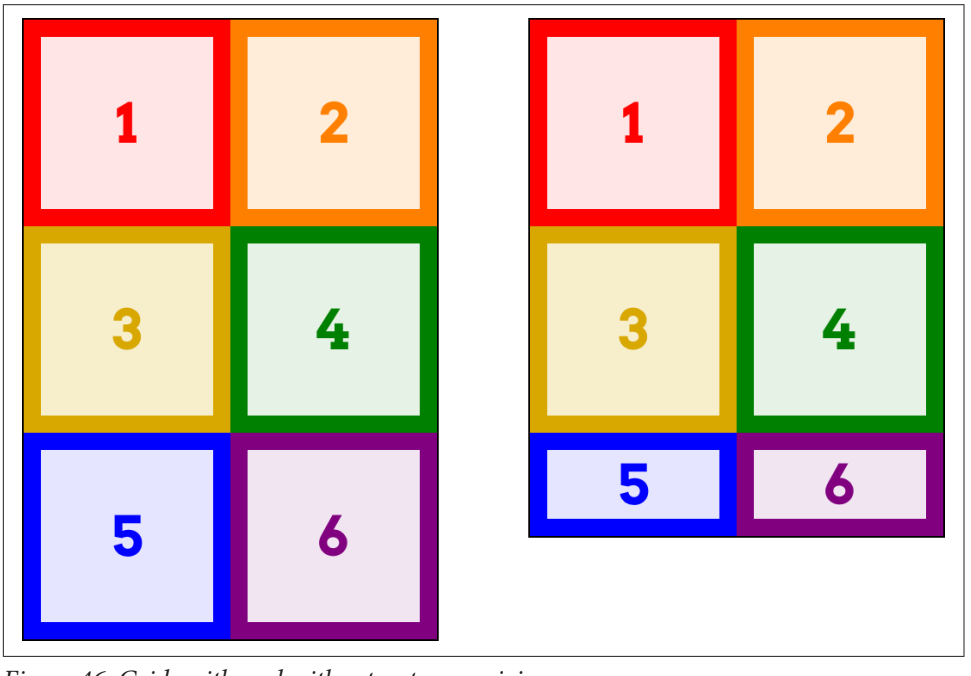


Figure 46. Grids with and without auto-row sizing

If we flip it around to columns, the same basic principles apply (see Figure 47):

```
.grid {display: grid; grid-auto-flow: column;
    grid-template-rows: 80px 80px;
    grid-template-columns: 80px 80px;}
#g1 {grid-auto-columns: 80px;}
```

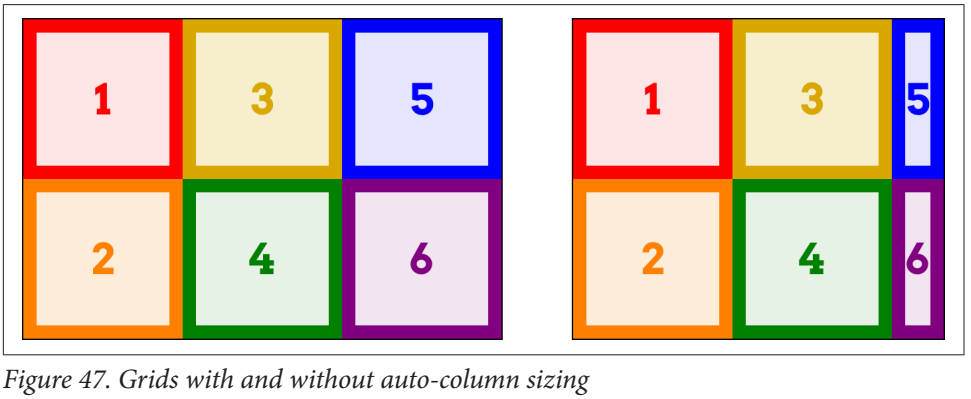


Figure 47. Grids with and without auto-column sizing

In this case, because the flow is column-oriented, the last grid item is placed into a new column past the end of the explicit grid. In the second grid, where there's no `grid-auto-columns`, that fifth item is as tall as its row (`80px`), but has an `auto` width, so it's just as wide as it needs to be and no wider. Of course, if a sixth item were added and it had wider content, then the column would be sized to fit that content, thus widening the fifth item.

So now you know what I used in the `grid-auto-flow` figures in the previous section: I silently made the auto-rows and auto-columns the same size as the explicitly sized columns, in order to not have the last couple of items look weird. Let's bring back one of those figures, only this time the `grid-auto-rows` and `grid-auto-columns` styles will be removed. As you can see in Figure 48, the last few items in each grid are shorter or narrower than the rest, due to the lack of auto-track sizing.

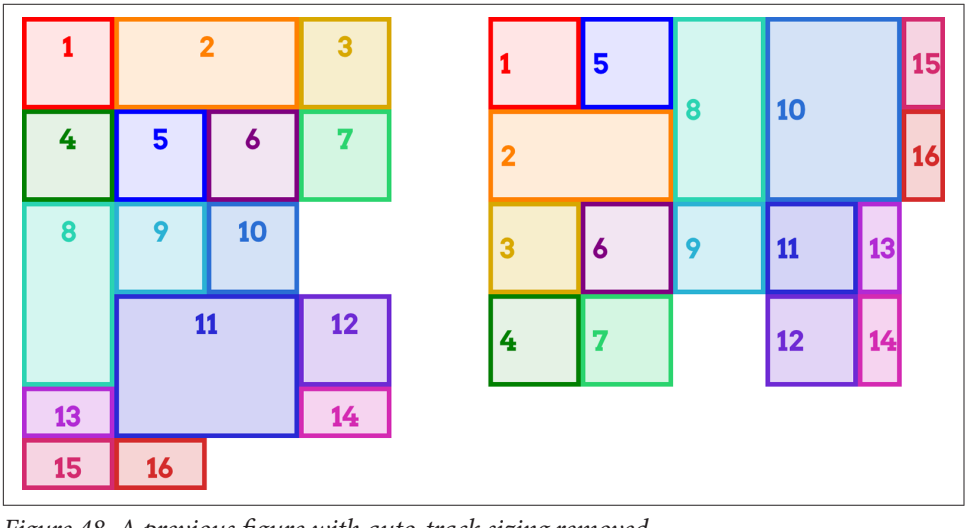


Figure 48. A previous figure with auto-track sizing removed

And now you know…the rest of the story.

The grid Shorthand

At long last, we've come to the shorthand property `grid`. It might just surprise you, though, because it's not like other shorthand properties.

grid

Values:	none | subgrid | [<grid-template-rows> / <grid-template-columns>] | [<line-names>? <string> <track-size>? <line-names>?]+ [/ <track-list>]? | [<grid-auto-flow> [<grid-auto-rows> [/ <grid-auto-columns>]?]?]]
Initial value:	See individual properties
Applies to:	Grid containers
Inherited:	No
Computed value:	See individual properties

The syntax is a little bit migraine-inducing, I admit, but we'll step through it a piece at a time.

Let's get to the elephant in the room right away: `grid` allows you to either define a grid template *or* you can set the grid's flow and auto-track sizing in a compact syntax. You can't do both at the same time.

Furthermore, whichever you don't define is reset to its defaults, as is normal for a shorthand property. So if you define the grid template, then the flow and auto tracks will be returned to their default values. This includes grid gutters, a topic we haven't even covered yet. You can't set the gutters with `grid`, but it will reset them anyway.

Yes, this is intentional. No, I don't know why.

So let's talk about creating a grid template using `grid`. The values can get fiendishly complex, and take on some fascinating patterns, but can be very handy. As an example, the following rule is equivalent to the set of rules that follows it:

```
grid:
    "header header header header" 3em
    ". content sidebar ." 1fr
    "footer footer footer footer" 5em /
    2em 3fr minmax(10em,1fr) 2em;

grid-template-areas:
    "header header header header"
    ". content sidebar ."
    "footer footer footer footer";
grid-template-rows: 3em 1fr 5em;
grid-template-columns: 2em 3fr minmax(10em,1fr) 2em;
```

Notice how the value of `grid-template-rows` is broken up and scattered around the strings of `grid-template-areas`. That's how row sizing is handled in `grid` when you have grid-area strings present. Take those strings out, and you end up with the following:

```
grid:
     3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;
```

In other words, the row tracks are separated by a solidus (`/`) from the column tracks.

Remember that with `grid`, undeclared shorthands are reset to their defaults. That means the following two rules are equivalent:

```
#layout {display: grid;
    grid: 3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;}

#layout {display: grid;
    grid: 3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;
    grid-auto-rows: auto;
    grid-auto-columns: auto;
    grid-auto-flow: row;}
```

Therefore, make sure your `grid` declaration comes before anything else related to defining the grid. That means that if we wanted a dense column flow, we'd write something like this:

```
#layout {display: grid;
    grid: 3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;
    grid-auto-flow: dense column;}
```

Now, let's bring the named grid areas back, *and* add some extra row grid-line names to the mix. A named grid line that goes *above* a row track is written *before* the string, and a grid line that goes *below* the row track comes *after* the string and any track sizing. So let's say we want to add `main-start` and `main-stop` above and below the middle row, and `page-end` at the very bottom:

```
grid:
    "header header header header" 3em
    [main-start] ". content sidebar ." 1fr [main-stop]
    "footer footer footer footer" 5em [page-end] /
    2em 3fr minmax(10em,1fr) 2em;
```

That creates the grid shown in Figure 49, with the implicitly created named grid lines (e.g., `footer-start`) along with the explicitly named grid lines we wrote into the CSS.

You can see how `grid` can get very complicated very quickly. It's a very powerful syntax, and it's surprisingly easy to get used to once you've had just a bit of practice. On the other hand, it's also easy to get things wrong and have the entire value be invalid, thus preventing the appearance of any grid at all.

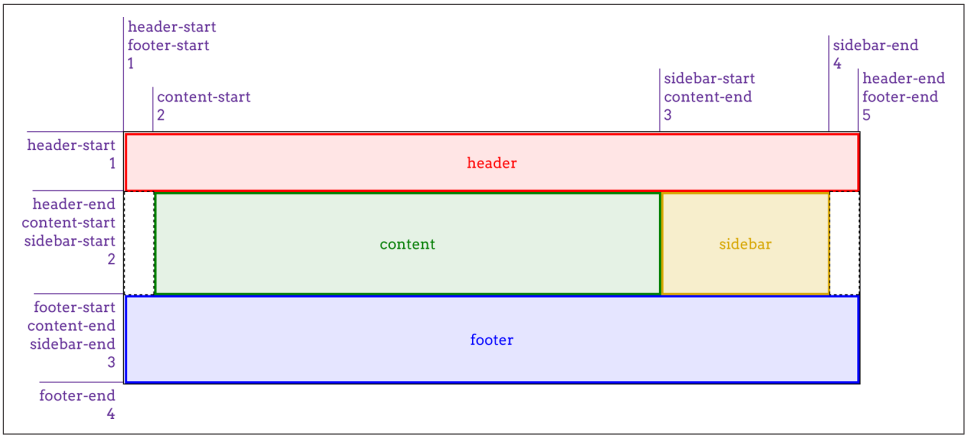


Figure 49. Creating a grid with the grid shorthand

For the other use of `grid`, it's a merging of `grid-auto-flow`, `grid-auto-rows`, and `grid-auto-columns`. The following rules are equivalent:

```
#layout {grid-auto-flow: dense rows;
    grid-auto-rows: 2em;
    grid-auto-columns: minmax(1em,3em);}

#layout {grid: dense rows 2em / minmax(1em,3em);}
```

That's certainly a lot less typing for the same result! But once again, I have to remind you: if you write this, then all the column and row track properties will be set to their defaults. Thus, the following rules are equivalent:

```
#layout {grid: dense rows 2em / minmax(1em,3em);}

#layout {grid: dense rows 2em / minmax(1em,3em);
        grid-template-rows: auto;
        grid-template-columns: auto;}
```

So once again, it's important to make sure your shorthand comes before any properties it might otherwise override.

Subgrids

There's another possible value for `grid`, which is `subgrid`. It might be used something like this:

```
#grid {display: grid;
    grid: repeat(auto-fill, 2em) / repeat(10, 1% 8% 1%);}
.module {display: grid;
    grid: subgrid;}
```

What happens inside each `module` element is that its grid items (i.e., its child elements) use the grid defined by `#grid` to align themselves.

This is potentially really useful, because you can imagine having a module that spans three of its parent's column patterns and containing child elements that are aligned to and laid out using the "master" grid. This is illustrated in Figure 50.

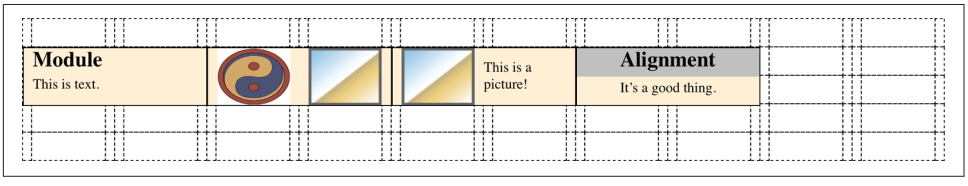


Figure 50. Aligning subgridded items

The problem is that, as of this writing, `subgrid` is an "at-risk" feature of grid layout, and may be dropped entirely. That's why it rates just this small section, instead of a more comprehensive examination.

Opening Grid Spaces

So far, we've seen a lot of grid items jammed right up against one another, with no space between them. There are a number of ways to mitigate this, as we'll talk about in this section, starting with gutters.

Grid Gutters (or Gaps)

Simply put, a *gutter* is a space between two grid tracks. It's created as if by expanding the grid line between them to have actual width. It's much like `border-spacing` in table styling—both because it creates space between grid cells and because you can set only a single spacing value for each axis, via the properties `grid-row-gap` and `grid-column-gap`.

grid-row-gap, grid-column-gap

Values:	<length>
Initial value:	`0`
Applies to:	Grid containers
Inherited:	No

Computed value: An absolute length

Right up front: as the value syntax shows, you can supply only a length for these properties; what it's less clear about is that the lengths must be non-negative. It's not possible to supply a percentage, a fractional value via `fr`, nor a minmax of some sort. If you want your columns to be separated by 1 em, then it's simple enough: `grid-column-gap: 1em`. That's pretty much as fancy as it gets. All the columns in the grid will be pushed apart by 1 em, as illustrated in Figure 51.

```
#grid {display: grid;
    grid-template-rows: 5em 5em;
    grid-template-columns: 15% 1fr 1fr;
    grid-column-gap: 1em;}
```

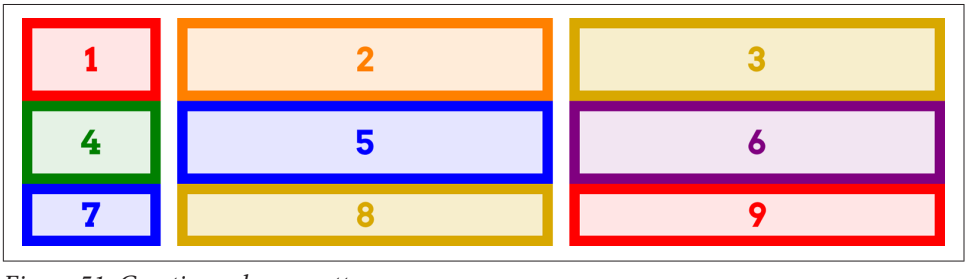


Figure 51. Creating column gutters

In terms of sizing the tracks in a grid, gutters are treated as if they're grid tracks. Thus, given the following styles, the fractional grid rows will each be 140 pixels tall.

```
#grid {display: grid; height: 500px;
    grid-template-rows: 100px 1fr 1fr 75px;
    grid-row-gap: 15px;}
```

We get 140 pixels for each fraction row's height because there are a total of 500 pixels of height. From that, we subtract the two row tracks (100 and 75) to get 325. From that result, we subtract the three 15-pixel gutters, which totals 45 pixels; this yields 280 pixels. That divided in half (because the fractional rows have equal fractions) gets us 140 pixels each. If the gutter value were increased to `25px`, then the fractional rows would have 250 pixels to divide between them, making each 125 pixels tall.

Of course, track sizing can be much more complicated than this; the example used all pixels because it makes the math simple. You can always mix units however you'd like, including minmaxing your actual grid tracks. This is one of the main strengths of grid layout.

Grid gutters can be changed from their declared size by the effects of `align-content` and `justify-content`. This will be covered in a later section.

There is, as you might have already suspected, a shorthand that combines row and column gap lengths into a single property.

grid-gap

Values:	<grid-row-gap> <grid-column-gap>
Initial value:	`0 0`
Applies to:	Grid containers
Inherited:	No
Computed value:	As declared

Not a lot more to say than that, really: supply two non-negative lengths, and you'll have defined the row gutters and column gutters, in that order. Here's an example, as shown in Figure 52:

```
#grid {display: grid;
    grid-template-rows: 5em 5em;
    grid-template-columns: 15% 1fr 1fr;
    grid-gap: 12px 2em;}
```

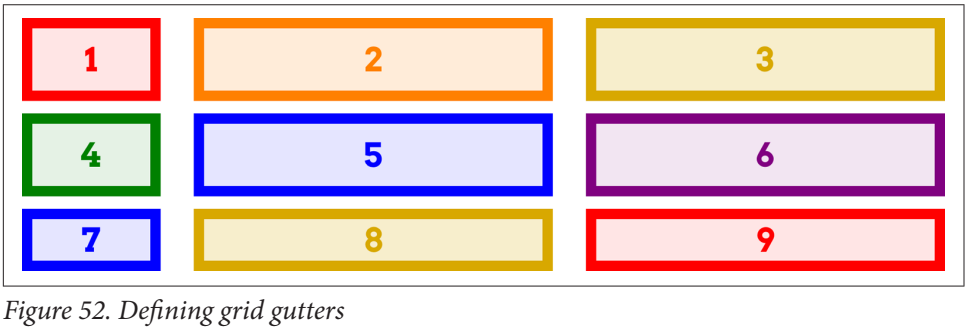


Figure 52. Defining grid gutters

Grid Items and the Box Model

Now we can create a grid, attach items to the grid, and even create gutters between the grid tracks. But what happens if we style the element that's attached to the grid with, say, margins? Or if it's absolutely positioned? How do these things interact with the grid?

Let's take margins first. The basic principle at work is that an element is attached to the grid by its margin edges. That means you can push the visible parts of the element inward from the grid area it occupies by setting positive margins—and pull it outward with negative margins. For example, these styles will have the result shown in Figure 53:

```
#grid {display: grid;
    grid-template-rows: repeat(2, 100px);
    grid-template-columns: repeat(2, 200px);}
.box02 {margin: 25px;}
.box03 {margin: -25px 0;}
```

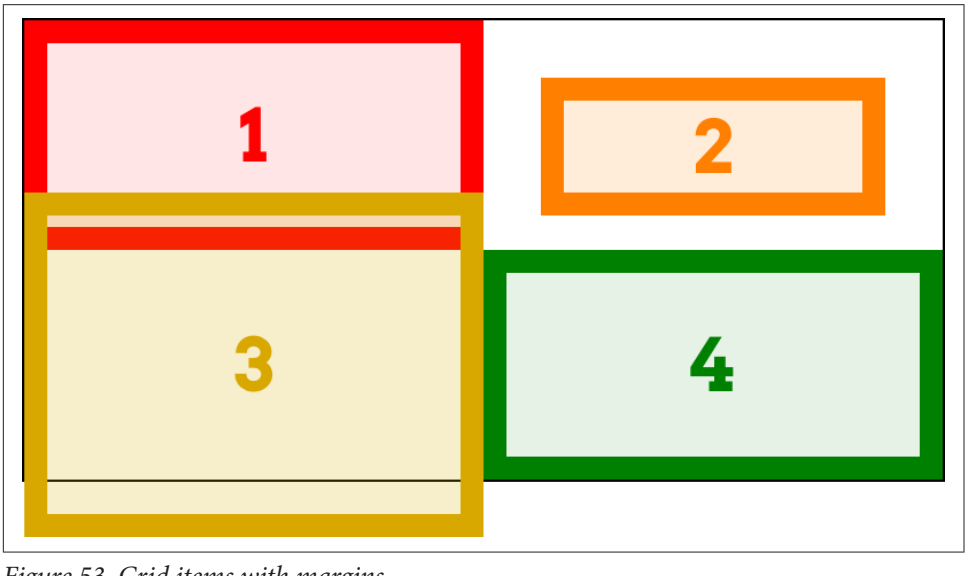


Figure 53. Grid items with margins

This worked as it did because the items had both their `width` and `height` set to `auto`, so they could be stretched as needed to make everything work out. If `width` and/or `height` have non-`auto` values, then they'll end up overriding margins to make all the math work out. This is much like what happens with right and left margins when element sizing is overconstrained: eventually, one of the margins gets overridden.

Consider an element with the following styles placed into a 200-pixel-wide by 100-pixel-tall grid area:

```
.exel {width: 150px; height: 100px;
    padding: 0; border: 0;
    margin: 10px;}
```

Going across the element first, it has 10 pixels of margin to either side, and its `width` is `150px`, giving a total of 170 pixels. Something's gotta give, and in this case it's the right margin (in left-to-right languages), which is changed to `40px` to make everything work—10 pixels on the left margin, 150 pixels on the content box, and 40 pixels on the right margin equals the 200 pixels of the grid area's width.

On the vertical axis, the bottom margin is reset to `-10px`. This compensates for the top margin and content height totalling 110 pixels, when the grid area is only 100 pixels tall.

Margins on grid items are ignored when calculating grid-track sizes. That means that no matter how big or small you make a grid item's margins, it won't change the sizing of a `min-content` column, for example, nor will increasing the margins on a grid item cause `fr`-sized grid tracks to change size.

As with block layout, you can selectively use `auto` margins to decide which margin will have its value changed to fit. Suppose we wanted the grid item to align to the right of its grid area. By setting the item's left margin to `auto`, that would happen:

```
.exel {width: 150px; height: 100px;
    padding: 0; border: 0;
    margin: 10px; margin-left: auto;}
```

Now the element will add up 160 pixels for the right margin and content box, and then give the difference between that and the grid area's width to the left margin, since it's been explicitly set to `auto`. This has the result shown in Figure 54, where there are 10 pixels of margin on each side of the `exel` item, except the left margin, which is (as we just calculated) 40 pixels.

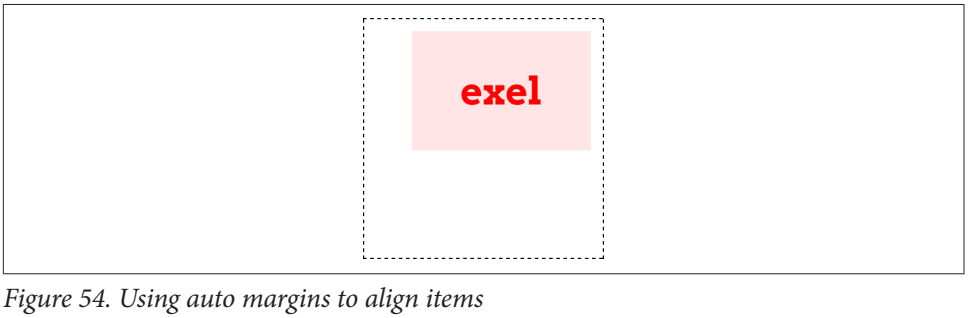


Figure 54. Using auto margins to align items

That might seem familiar from block-level layout, where you can use `auto` left and right margins to center an element in its containing block, as long as you've given it an explicit `width`. Where grid layout differs is that you can do the same thing on the vertical axis; that is, given an element with an absolute height, you can vertically center it by setting the top and bottom margins to `auto`. Figure 55 shows a variety of `auto` margin effects on elements that naturally have explicit heights and widths: images.

```
.i01 {margin: 10px;}
.i02 {margin: 10px; margin-left: auto;}
.i03 {margin: auto 10px auto auto;}
.i04 {margin: auto;}
.i05 {margin: auto auto 0 0;}
.i06 {margin: 0 auto;}
```

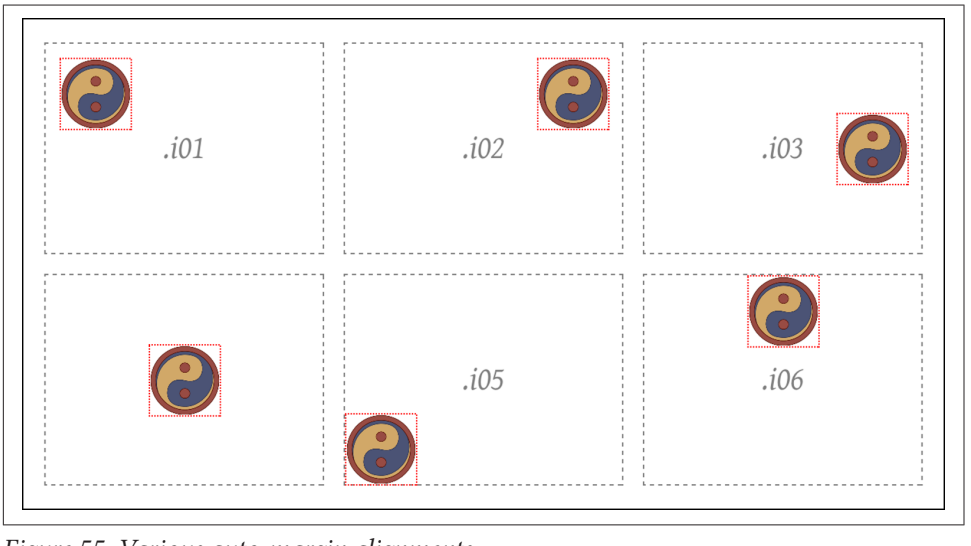


Figure 55. Various auto-margin alignments

There are other ways to align grid items, notably with properties like `justify-self`, which don't depend on having explicit `height` and `width` values. These will be covered in the next section.

This is a lot like how margins and element sizes operate when elements are absolutely positioned. Which leads us to the next question: what if a grid item is *also* absolutely positioned? For example:

```
.exel {grid-row: 2 / 4; grid-column: 2 / 5;
    position: absolute;
    top: 1em; bottom: 15%;
    left: 35px; right: 1rem;}
```

The answer is actually pretty elegant: if you've defined grid-line starts and ends, that grid area is used as the containing block and positioning context, and the grid item is positioned *within* that context. That means the offset properties (`top` et al.) are calculated in relation to the declared grid area. Thus, the previous CSS would have the result shown in Figure 56.

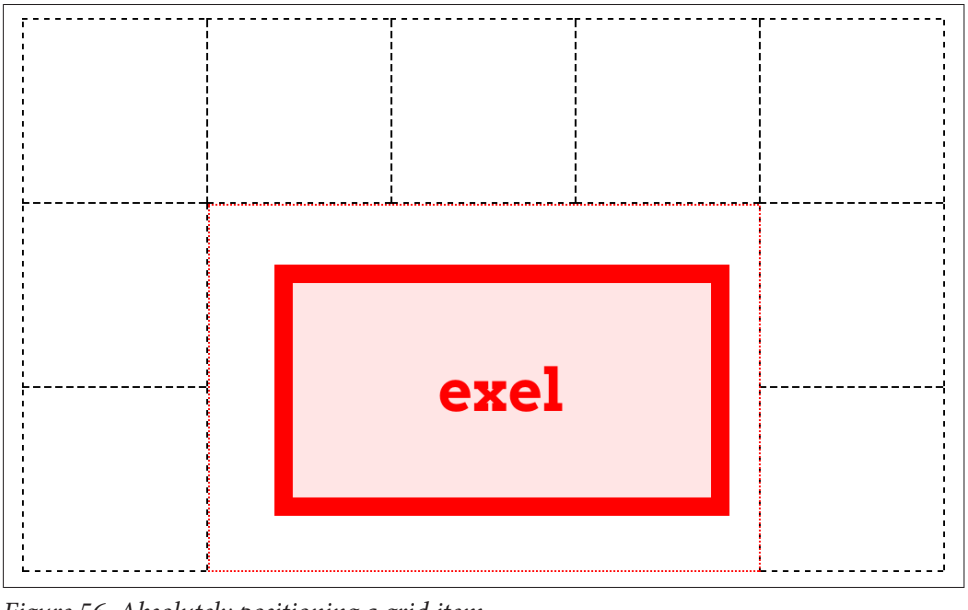


Figure 56. Absolutely positioning a grid item

Everything you know about absolutely positioned elements regarding offsets, margins, element sizing, and so on applies within this formatting context. It's just that in this case, the formatting context is defined by a grid area.

There is a wrinkle that absolute positioning introduces: it changes the behavior of the `auto` value for grid-line properties. If, for example, you set `grid-column-end: auto` for an absolutely positioned grid item, the ending grid line will actually create a new and special grid line that corresponds to the padding edge of the grid container itself. This is true even if the explicit grid is smaller than the grid container, as can happen.

To see this in action, we'll modify the previous example as follows, with the result shown in Figure 57:

```
.exel {grid-row: 1 / auto; grid-column: 2 / auto;
    position: absolute;
    top: 1em; bottom: 15%;
    left: 35px; right: 1rem;}
```

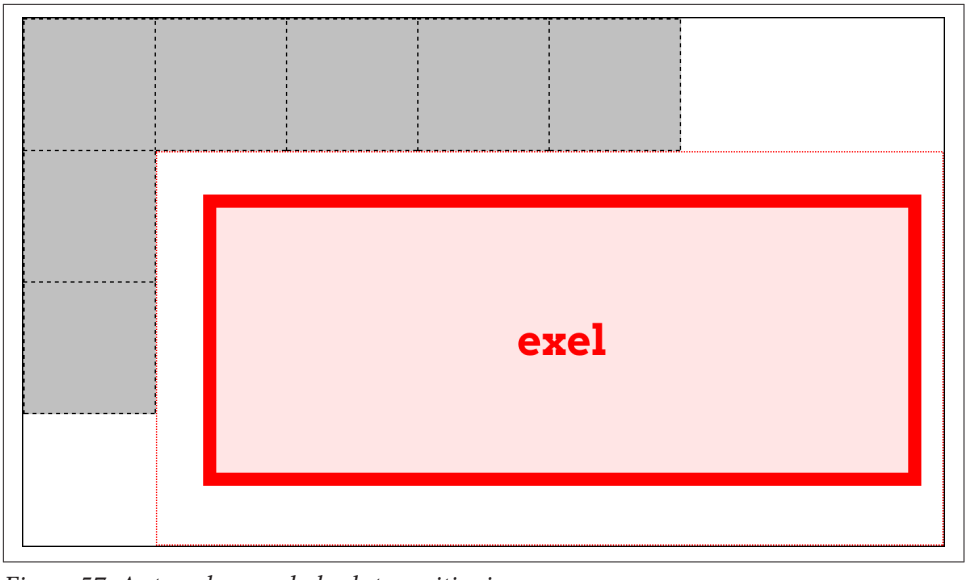


Figure 57. Auto values and absolute positioning

Note how the positioning context now starts at the top of the grid container, and stretches all the way to the right edge of the grid container, even though the grid itself ends well short of that edge.

One implication of this behavior is that if you absolutely position an element that's a grid item, but you don't give it any gridline start or end values, then it will use the inner padding edge of the grid container as its positioning context. It does this without having to set the grid container to `position: relative`, or any of the other usual tricks to establish a positioning context.

Note that absolutely positioned grid items do *not* participate in figuring out grid cell and track sizing. As far as the grid layout is concerned, the positioned grid item doesn't exist. Once the grid is set up, then the grid item is positioned with respect to the grid lines that define its positioning context.

As of early 2016, browsers did not support any of this absolute positioning behavior. The only way to re-create it was to relatively position the element establishing the grid area, and absolutely position a child element within it. That's how the absolute-positioning figures in this section were created. The special `auto` behavior was also not supported.

Aligning and Grids

If you have any familiarity with flexbox, you're probably aware of the various alignment properties and their values. Those same properties are also available in grid layout, and have very similar effects.

First, a quick refresher. The properties that are available and what they affect are summarized in Table 1.

Table 1. Justify and align values

Property	Aligns	Applied to
`justify-self`	A grid item in the inline (horizontal) direction	Grid items
`justify-items`	All grid items in the inline (horizontal) direction	Grid container
`justify-content`	The entire grid in the inline (horizontal) direction	Grid container
`align-self`	A grid item in the block (vertical) direction	Grid items
`align-items`	All grid items in the block (vertical) direction	Grid container
`align-content`	The entire grid in the block (vertical) direction	Grid container

As Table 1 shows, the various `justify-*` properties change alignment along the inline axis—in English, this will be the horizontal direction. The difference is whether a property applies to a single grid item, all the grid items in a grid, or the entire grid. Similarly, the `align-*` properties affect alignment along the block axis; in English, this is the vertical direction.

Aligning and Justifying Individual Items

It's easiest to start with the `*-self` properties, because we can have one grid show various `justify-self` property values, while a second grid shows the effects of those same values when used by `align-self`. (See Figure 58.)

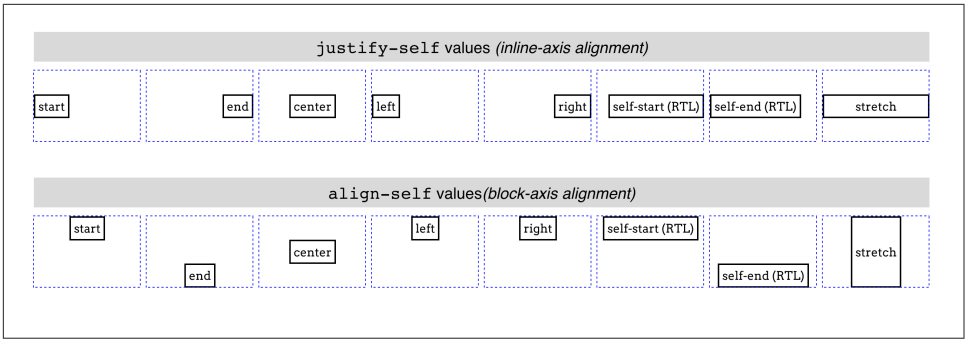


Figure 58. Self alignment in the inline and block directions

Each grid item in Figure 58 is shown with its grid area (the dashed blue line) and a label identifying the property value that's applied to it. Each deserves a bit of commentary.

First, though, realize that for all of these values, any element that doesn't have an explicit `width` or `height` will "shrink-wrap" its content, instead of using the default grid-item behavior of filling out the entire grid area.

`start` and `end` cause the grid item to be aligned to the start or end edge of its grid area, which makes sense. Similarly, `center` centers the grid item within its area along the alignment axis, *without* the need to declare margins or any other properties, including `height` and `width`.

`left` and `right` have the expected results for horizontal alignment, but if they're applied to elements via `align-self` (which is vertical alignment), they're treated as `start`.

`self-start` and `self-end` are more interesting. `self-start` aligns a grid item with the grid-area edge that corresponds to the grid *item's* start edge. So in Figure 58, the `self-start` and `self-end` boxes were set to `direction: rtl`. That set them to use right-to-left language direction, meaning their start edges were their right edges, and their end edges their left. You can see in the first grid that this right-aligned `self-start` and left-aligned `self-end`. In the second grid, however, the RTL direction is irrelevant to block-axis alignment. Thus, `self-start` was treated as `start`, and `self-end` was treated as `end`.

The last value, `stretch`, is interesting. To understand it, notice how the other boxes in each grid "shrink-wrap" themselves to their content. `stretch`, on the other hand, directs the element to stretch from edge to edge in the given direction—`align-self: stretch` causes the grid item to stretch vertically, and `justify-self: stretch` causes horizontal stretching. This is as you might expect, but bear in mind that it works only

if the element's size properties are set to `auto`. Thus, given the following styles, the first example will stretch vertically, but the second will not:

```
.exel01 {align-self: stretch; height: auto;}
.exel02 {align-self: stretch; height: 50%;}
```

Because the second example sets a `height` value that isn't `auto` (which is the default value), it cannot be resized by `stretch`. The same holds true for `justify-self` and `width`.

There are two more values that can be used to align grid items, but they are sufficiently interesting to merit their own explanation. These permit the alignment of a grid item's first or last baseline with the highest or lowest baseline in the grid track. For example, suppose you wanted a grid item to be aligned so the baseline of its last line was aligned with the last baseline in the tallest grid item sharing its row track. That would look like the following:

```
.exel {align-self: last-baseline;}
```

Conversely, to align its first baseline with the lowest first baseline in the same row track, you'd say this:

```
.exel {align-self: baseline;}
```

As of early 2016, no browsers supported `baseline` and `last-baseline` alignment in grid layout.

In a situation where a grid element doesn't have a baseline, or it's asked to baseline-align itself in a direction where baselines can't be compared, then `baseline` is treated as `start` and `last-baseline` is treated as `end`.

There are two values that were intentionally skipped in this section: `flex-start` and `flex-end`. These values are supposed to be used only in flexbox layout, and are defined to be equivalent to `start` and `end` in any other layout context, including grid layout.

Aligning and Justifying All Items

Now let's consider `align-items` and `justify-items`. These properties accept all the same values we saw in the previous section, and have the same effect, except they apply to all grid items in a given grid container, and must be applied to a grid container instead of to individual grid items.

Thus, you could set all of the grid items in a grid to be center-aligned within their grid areas as follows, with a result like that depicted in Figure 59:

```
#grid {display: grid;
    align-items: center; justify-items: center;}
```

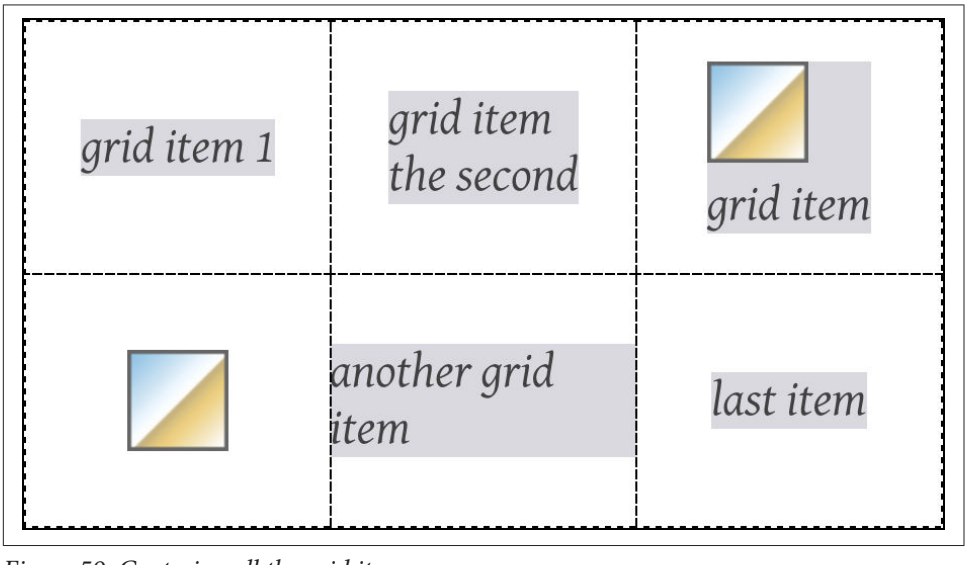


Figure 59. Centering all the grid items

As you can see, that horizontally *and* vertically centers every grid item within its given grid area. Furthermore, it causes any grid item without an explicit width and height to "shrink-wrap" its content rather than stretch out to fill their grid area. If a grid item has an explicit height and width, then those will be honored, and the item centered within its grid area.

Beyond aligning and justifying every grid item, it's possible to distribute the grid items, or even to justify or align the entire grid, using `align-content` and `justify-content`. There is a small set of distributive values for these properties. Figure 60 illustrates the effects of each value as applied to `justify-content`, with each grid sharing the following styles:

```
.grid {display: grid; padding: 0.5em; margin: 0.5em 1em; width: auto;
        grid-gap: 0.75em 0.5em; border: 1px solid;
        grid-template-rows: 4em;
        grid-template-columns: repeat(5, 6em);}
```

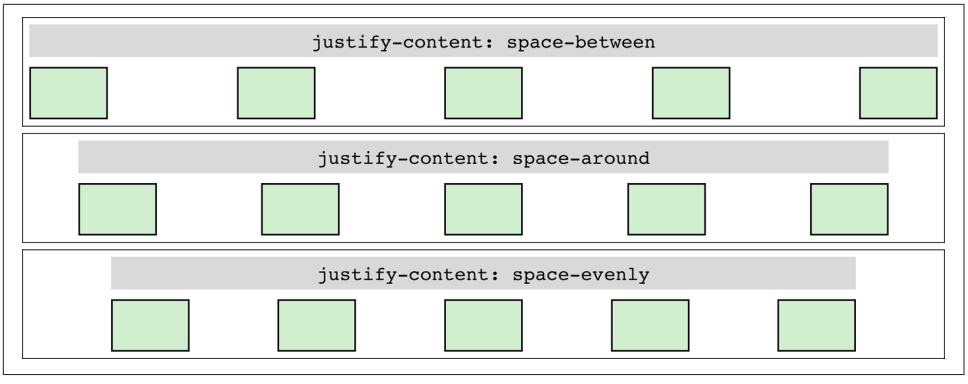


Figure 60. Distributing grid items horizontally

This works just as well in column tracks as it does in row tracks, as Figure 61 illustrates, as long as you switch to `align-content`. This time, the grids all share these styles:

```
.grid {display: grid; padding: 0.5em;
       grid-gap: 0.75em 0.5em; border: 1px solid;
       grid-template-rows: repeat(4, 3em);
       grid-template-columns: 5em;}
```

The way this distribution works is that the grid tracks, including any gutters, are all sized as usual. Then, if there is any leftover space within the grid container—that is, if the grid tracks don't reach all the way from one edge of the grid container to the other—then the remaining space is distributed according to the value of `justify-content` (in the horizontal) or `align-content` (in the vertical).

This space distribution is carried out by resizing the grid gutters. If there are no declared gutters, there will be gutters. If there are already gutters, their sizes are altered as required to distribute the grid tracks.

Note that because space is distributed only when the tracks don't fill out the grid container, the gutters can only increase in size. If the tracks are larger than the container, which can easily happen, there is no leftover space to distribute (negative space turns out to be indivisible).

There is another distribution value, very new as of this writing, which wasn't shown in the previous figures. `stretch` takes any leftover space and applies it equally to the grid tracks, not the gutters. So if there are 400 pixels of leftover space and eight grid tracks, each grid track is increased by 50 pixels. The grid tracks are *not* increased proportionally, but equally. As of early 2016, there was no browser support for this value in terms of grid distribution.

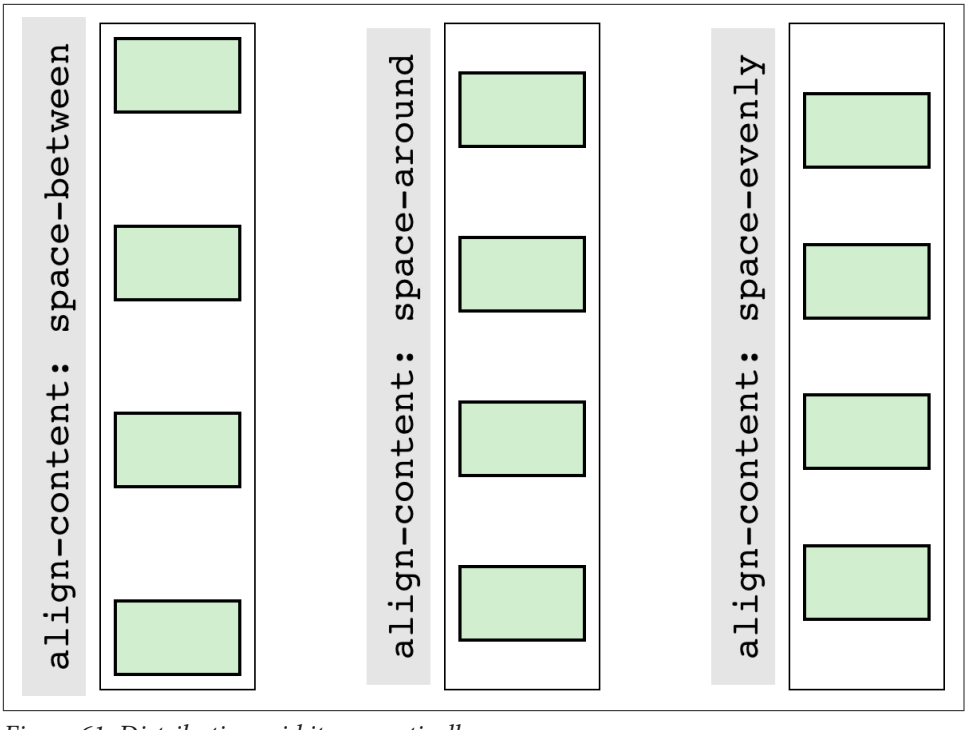


Figure 61. Distributing grid items vertically

We'll round out this section with examples of justifying, as opposed to distributing, grid tracks. Figure 62 shows the possibilities when justifying horizontally.

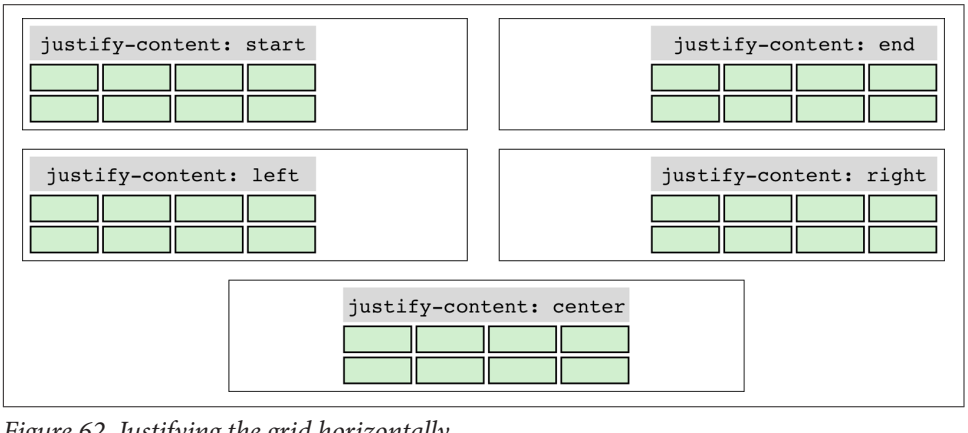


Figure 62. Justifying the grid horizontally

In these cases, the set of grid tracks is taken as a single unit, and justified by the value of `justify-content`. That alignment does not affect the alignment of individual grid

items; thus, you could end-justify the whole grid with `justify-content: end` while having individual grid items be left-, center-, or start-justified (among other options) within their grid areas.

As you might expect by now, being able to `justify-content` horizontally means you can `align-content` vertically. Figure 63 shows each value in action.

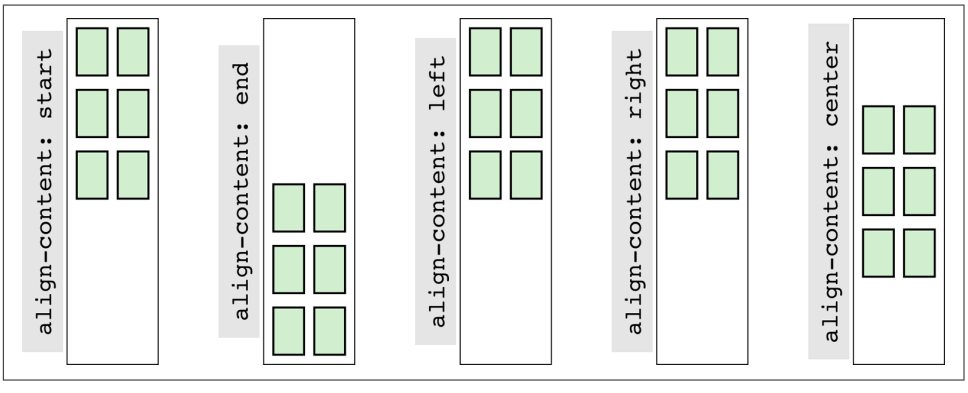


Figure 63. Aligning the grid vertically

Of course, `left` and `right` don't make sense in a vertical context, so they're treated as `start`. The others have the effect you'd expect from their names.

Layering and Ordering

As we saw in a previous section, it's entirely possible to have grid items overlap each other, whether because negative margins are used to pull a grid item beyond the edges of its grid area, or simply because the grid areas of two different grid items share grid cells. By default, the grid items will visually overlap in document source order: grid items later in the document source will appear in front of grid items earlier in the document source. Thus we see the following result in what's depicted in Figure 64. (Assume the number in each class name represents the grid item's source order.)

```
#grid {display: grid; width: 80%; height: 20em;
    grid-rows: repeat(10, 1fr); grid-columns: repeat(10, 1fr);}
.box01 {grid-row: 1 / span 4; grid-column: 1 / span 4;}
.box02 {grid-row: 4 / span 4; grid-column: 4 / span 4;}
.box03 {grid-row: 7 / span 4; grid-column: 7 / span 4;}
.box04 {grid-row: 4 / span 7; grid-column: 3 / span 2;}
.box05 {grid-row: 2 / span 3; grid-column: 4 / span 5;}
```

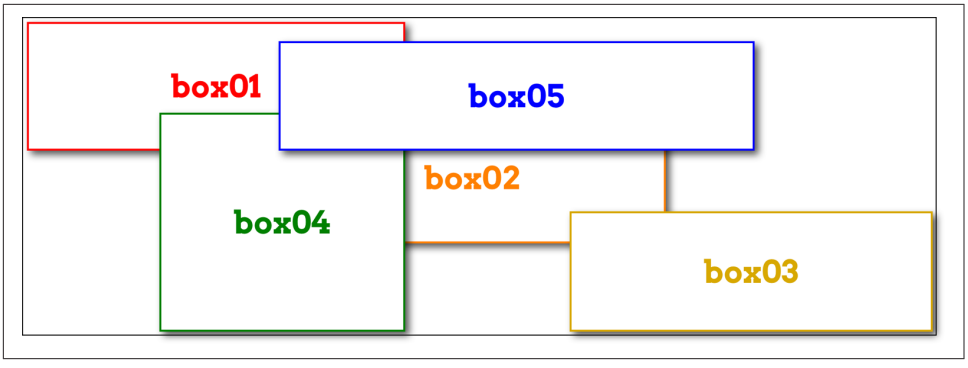


Figure 64. Grid items overlapping in source order

If you want to assert your own stacking order, then `z-index` is here to help. Just as in positioning, `z-index` places elements relative to each other on the z-axis, which is perpendicular to the display surface. Positive values are closer to you, and negative values further away. So to bring the second box to the "top," as it were, all you need is to give it a `z-index` value higher than any other (with the result shown in Figure 65):

```
.box02 {z-index: 10;}
```

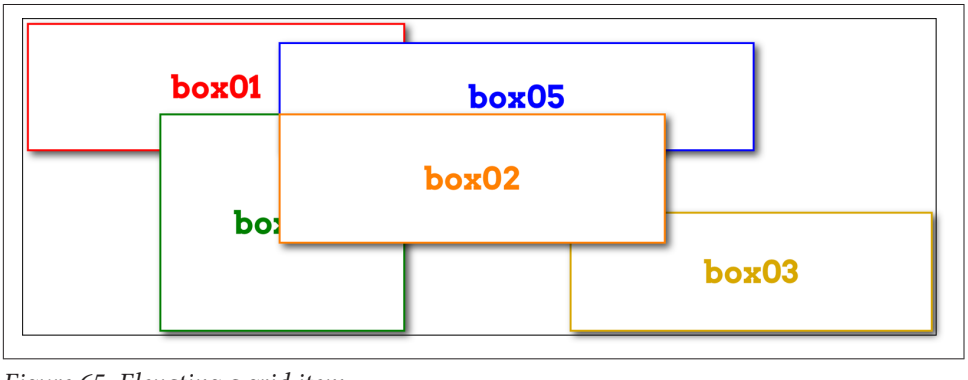


Figure 65. Elevating a grid item

Another way you can affect the ordering of grid items is by using the `order` property. Its effect is essentially the same as it is in flexbox—you can change the order of grid items within a grid track by giving them `order` values. This affects not only placement within the track, but also *paint order* if they should overlap. For example, we could change the previous example from `z-index` to `order`, as shown here, and get the same result shown in Figure 65:

```
.box02 {order: 10;}
```

In this case, `box02` appears "on top of" the other grid items because its order places it after the rest of them. Thus, it's drawn last. Similarly, if those grid items were all

placed in sequence in a grid track, the `order` value for `box02` would put it at the end of the sequence. This is depicted in Figure 66.

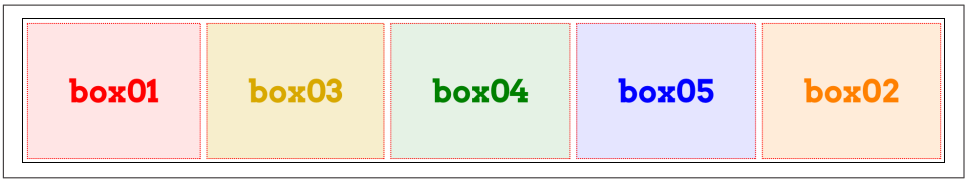


Figure 66. Changing grid-item order

Remember that just because you *can* rearrange the order of grid items this way, it doesn't necessarily mean you *should*. As the Grid Layout specification says (section 4.2):

> As with reordering flex items, the `order` property must only be used when the visual order needs to be *out-of-sync* with the speech and navigation order; otherwise the underlying document source should be reordered instead.

So the only reason to use `order` to rearrange grid item layout is if you need to have the document source in one order and layout in the other. This is already easily possible by assigning grid items to areas that don't match source order, of course.

This is not to say that `order` is useless and should always be shunned; there may well be times it makes sense. But unless you find yourself nearly forced into using it by specific circumstances, think very hard about whether it's the best solution.

Summary

Grid layout is complex and powerful, so don't be discouraged if you feel overwhelmed. It takes some time to get used to how grid operates, especially since so many of its features are nothing like what we've dealt with before. Much of those features' power comes directly from their novelty—but like any powerful tool, it can be difficult and frustrating to learn to use. I got frustrated and confused as I wrote about grid, going down blind alleys and falling victim to two decades of instincts that had been honed on a layout-less CSS.

I hope I was able to steer you past some of those pitfalls, but still, remember the wisdom of Master Yoda: "You must unlearn what you have learned." When coming to grid layout, there has never been greater need to put aside what you think you know about layout and learn anew. Over time, your patience and persistence will be rewarded.

About the Author

Eric A. Meyer has been working with the Web since late 1993 and is an internationally recognized expert on the subjects of HTML, CSS, and web standards. A widely read author, he is also the founder of Complex Spiral Consulting, which counts among its clients America Online; Apple Computer, Inc.; Wells Fargo Bank; and Macromedia, which described Eric as "a critical partner in our efforts to transform Macromedia Dreamweaver MX 2004 into a revolutionary tool for CSS-based design."

Beginning in early 1994, Eric was the visual designer and campus web coordinator for the Case Western Reserve University website, where he also authored a widely acclaimed series of three HTML tutorials and was project coordinator for the online version of the *Encyclopedia of Cleveland History* and the *Dictionary of Cleveland Biography*, the first encyclopedia of urban history published fully and freely on the Web.

Author of *Eric Meyer on CSS* and *More Eric Meyer on CSS* (New Riders), *CSS: The Definitive Guide* (O'Reilly), and *CSS 2.0 Programmer's Reference* (Osborne/McGraw-Hill), as well as numerous articles for the O'Reilly Network, Web Techniques, and Web Review, Eric also created the CSS Browser Compatibility Charts and coordinated the authoring and creation of the W3C's official CSS Test Suite. He has lectured to a wide variety of organizations, including Los Alamos National Laboratory, the New York Public Library, Cornell University, and the University of Northern Iowa. Eric has also delivered addresses and technical presentations at numerous conferences, among them An Event Apart (which he cofounded), the IW3C2 WWW series, Web Design World, CMP, SXSW, the User Interface conference series, and The Other Dreamweaver Conference.

In his personal time, Eric acts as list chaperone of the highly active css-discuss mailing list, which he cofounded with John Allsopp of Western Civilisation, and which is now supported by *evolt.org*. Eric lives in Cleveland, Ohio, which is a much nicer city than you've been led to believe. For nine years he was the host of "Your Father's Oldsmobile," a big-band radio show heard weekly on WRUW 91.1 FM in Cleveland.

You can find more detailed information on Eric's personal web page.

Colophon

The animals on the cover of *Grid Layout in CSS* are salmon (*salmonidae*), which is a family of fish consisting of many different species. Two of the most common salmon are the Pacific salmon and the Atlantic salmon.

Pacific salmon live in the northern Pacific Ocean off the coasts of North America and Asia. There are five subspecies of Pacific salmon, with an average weight of 10 to 30 pounds. Pacific salmon are born in the fall in freshwater stream gravel beds, where

they incubate through the winter and emerge as inch-long fish. They live for a year or two in streams or lakes and then head downstream to the ocean. There they live for a few years, before heading back upstream to their exact place of birth to spawn and then die.

Atlantic salmon live in the northern Atlantic Ocean off the coasts of North America and Europe. There are many subspecies of Atlantic salmon, including the trout and the char. Their average weight is 10 to 20 pounds. The Atlantic salmon family has a life cycle similar to that of its Pacific cousins, and also travels from freshwater gravel beds to the sea. A major difference between the two, however, is that the Atlantic salmon does not die after spawning; it can return to the ocean and then return to the stream to spawn again, usually two or three times.

Salmon, in general, are graceful, silver-colored fish with spots on their backs and fins. Their diet consists of plankton, insect larvae, shrimp, and smaller fish. Their unusually keen sense of smell is thought to help them navigate from the ocean back to the exact spot of their birth, upstream past many obstacles. Some species of salmon remain landlocked, living their entire lives in freshwater.

Salmon are an important part of the ecosystem, as their decaying bodies provide fertilizer for streambeds. Their numbers have been dwindling over the years, however. Factors in the declining salmon population include habitat destruction, fishing, dams that block spawning paths, acid rain, droughts, floods, and pollution.

The cover image is a 19th-century engraving from the Dover Pictorial Archive. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.