

CSS Grid Layout Module Level 3

Editor's Draft, 6 May 2026



▼ More details about this document

This version:

<https://drafts.csswg.org/css-grid-3/>

Latest published version:

<https://www.w3.org/TR/css-grid-3/>

Feedback:

[CSSWG Issues Repository](#)

[Inline In Spec](#)

Editors:

[Tab Atkins Jr.](#) (Google)

[Elika J. Etemad / fantasai](#) (Apple)

[Jen Simmons](#) (Apple)

[Brandon Stewart](#) (Apple)

Former Editor:

[Mats Palmgren](#) (Mozilla)

Suggest an Edit for this Spec:

[GitHub Editor](#)

Copyright © 2026 [World Wide Web Consortium](#). W3C[®] [liability](#), [trademark](#) and [permissive document license](#) rules apply.

Abstract

This module introduces a one-dimensional grid layout mode for [CSS Grid](#) containers.

[CSS](#) is a language for describing the rendering of structured documents (such as HTML and XML) on screen, on paper, etc.

Status of this document

This is a public copy of the editors' draft. It is provided for discussion only and may change at any moment. Its publication here does not imply endorsement of its contents by W3C. Don't cite this

document other than as work in progress.

Please send feedback by [filing issues in GitHub](#) (preferred), including the spec code “css-grid” in the title, like this: “[css-grid] ...*summary of comment*...”. All issues and comments are [archived](#). Alternately, feedback can be sent to the ([archived](#)) public mailing list www-style@w3.org.

This document is governed by the [18 August 2025 W3C Process Document](#).

Table of Contents

1 Introduction

- 1.1 Background and Motivation
 - 1.1.1 Waterfall Layout with Auto-placed Items
 - 1.1.2 One-dimensional Grid Layout
- 1.2 Value Definitions

2 Grid Lanes Layout Model

- 2.1 Reordering and Accessibility
- 2.2 Establishing Grid Lanes Layout
- 2.3 Orienting Grid Lanes Layout

3 Grid Lanes Track Specification

- 3.1 Declaring Grid Lanes Track Templates: the ‘[grid-template-*](#)’ properties
 - 3.1.1 Intrinsic Tracks and `repeat()`
- 3.2 Subgrids
- 3.3 Track Repetition: the ‘[repeat\(\)](#)’ notation
 - 3.3.1 `repeat(auto-fit)`
- 3.4 Grid Axis Track Sizing
 - 3.4.1 Subgrid Item Contributions
 - 3.4.2 Optimized Track Sizing

4 Grid Lanes Item Placement

- 4.1 Specifying Grid Axis Item Placement: the ‘[grid-column-*](#)’ and ‘[grid-row-*](#)’ properties
- 4.2 Placement Precision: the ‘[flow-tolerance](#)’ property
- 4.3 Dense Placement: the ‘[dense](#)’ keyword
- 4.4 Grid Lanes Layout and Placement Algorithm
 - 4.4.1 Containing Block
 - 4.4.2 Placement and Writing Modes

5 Sizing Grid Containers

6 Alignment and Spacing

6.1 Gutters: the ‘[row-gap](#)’, ‘[column-gap](#)’, and ‘[gap](#)’ properties

6.2 Grid-axis Alignment: the ‘[align-content](#)’/‘[justify-content](#)’, ‘[align-self](#)’/‘[justify-self](#)’, and ‘[align-items](#)’/‘[justify-items](#)’ properties

6.3 Stacking-axis Content Distribution: the ‘[align-content](#)’/‘[justify-content](#)’ properties

6.4 Stacking-axis Self Alignment: the ‘[align-self](#)’/‘[justify-self](#)’ and ‘[align-items](#)’/‘[justify-items](#)’ properties

6.5 Baseline Alignment

7 Fragmentation

7.1 Fragmentation in the stacking axis

7.2 Fragmentation in the Grid Axis

8 Absolute Positioning

9 Graceful Degradation

10 Acknowledgements

11 Security Considerations

12 Privacy Considerations

Changes

Additions Since Level 2

Recent Changes

Conformance

Document conventions

Conformance classes

Partial implementations

Implementations of Unstable and Proprietary Features

Non-experimental implementations

Index

Terms defined by this specification

Terms defined by reference

References

[Normative References](#)[Non-Normative References](#)[Property Index](#)[Issues Index](#)

§ 1. Introduction

This section is not normative.

Grid Layout is a layout model for CSS that has powerful abilities to control the sizing and positioning of boxes and their contents. Grid Layout is optimized for 2-dimensional layouts: those in which alignment of content is desired in both dimensions.



Figure 1 Representative Grid layout example

Although many layouts can be expressed with regular Grid Layout, restricting items into a grid in both axes also makes it impossible to express some common layouts on the Web.

This module defines a variant of [Grid Layout](#) that removes that restriction so that items can be placed into Grid-like tracks in one axis while stacking one after another in the other. Items not explicitly placed into specific tracks are placed into the column (or row) with the most remaining space based on the layout size of the items placed so far.

§ 1.1. Background and Motivation

§ 1.1.1. Waterfall Layout with Auto-placed Items

[Grid lanes layout](#), sometimes also called “masonry layout” or “waterfall layout”, is a common Web design pattern where a number of items—commonly images or short article summaries—are placed one by one into columns in a way that loosely resembles masonry construction. Unlike [multi-column layout](#) and [multi-line column flex layout](#), where content is placed vertically in the first column until it must spill over to the second column, [automatic placement](#) in grid lanes layout selects

a column for each new item such that it is generally closer to the top of the layout than items placed later.

EXAMPLE 1

The traditional Pinterest search results page exemplifies this layout:

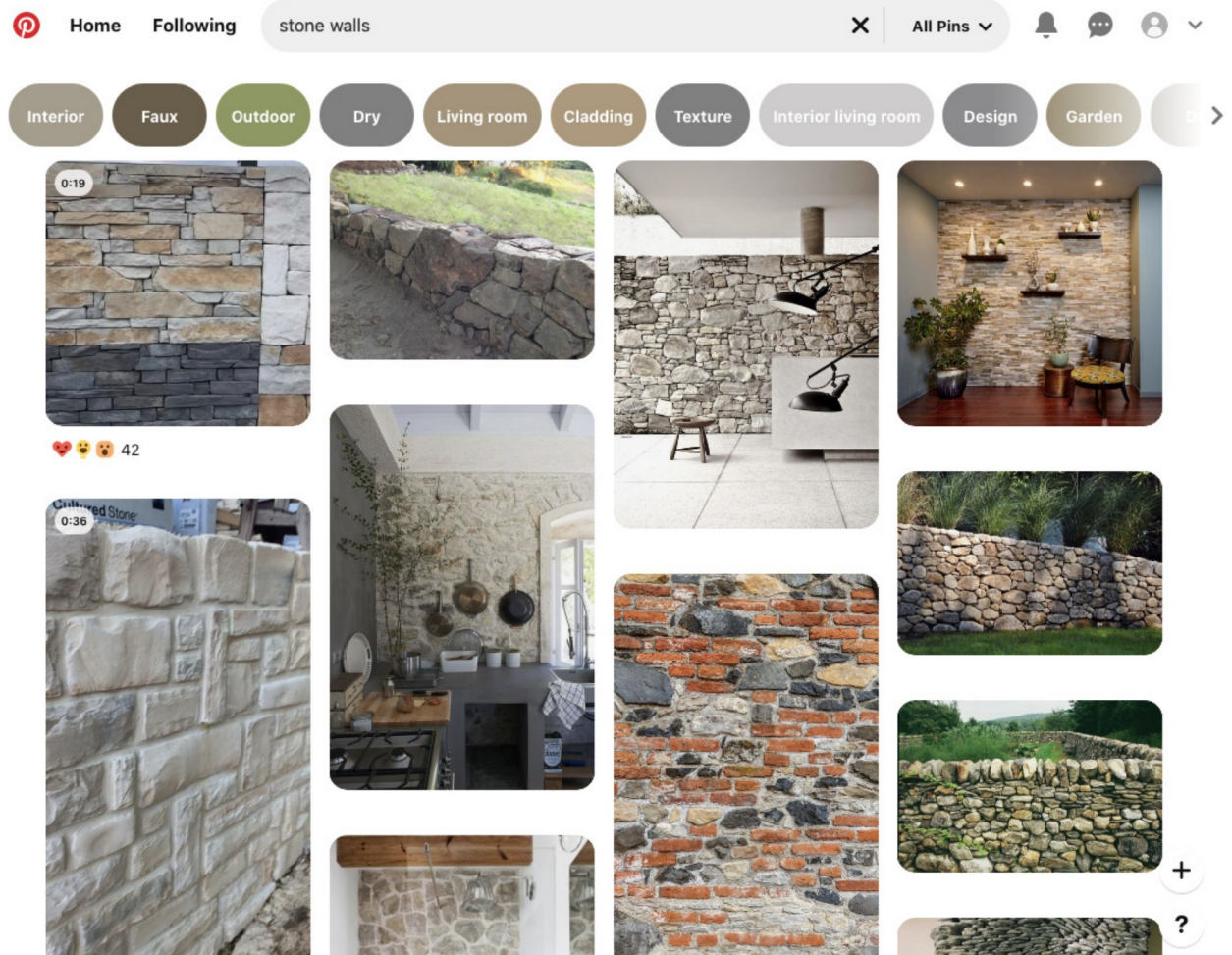


Figure 2 Representative grid lanes layout example

Here, each item has a different height (depending on the content and the width of the column), and inspecting the DOM reveals (as the visual content itself gives no indication of ordering) that each item has been placed into the column with the smallest height so far.

This layout superficially looks similar to multi-column layout; but it has the advantage that scrolling down will naturally lead to “later” items in the layout (such as those less relevant in the search results).

It's not possible to achieve this layout using earlier CSS layout models, unless you know up-front how tall each item will be, or use JavaScript for content measurement or placement.

Using `'display: grid-lanes'` together with `'grid-template-columns'` yields this type of [grid lanes layout](#)

§ 1.1.2. One-dimensional Grid Layout

[Grid layout](#) allows for powerful track sizing and explicit placement in two axes, but sometimes a layout only needs alignment of its items in one dimension.

Using [grid lanes layout](#) together with explicitly-positioned items allows for this type of one-dimensional grid layout.

EXAMPLE 2



This example [by Douglas Graham](#) uses explicit positioning to place each item into its assigned column; but there are no rows. Instead, items in each column stack one after the other. This layout can't be duplicated in [grid layout](#) because the “spanning” relationships among the items in adjacent columns is not fixed: it depends on their relative heights and whether the optional banner or advertisement items are included. It also can't be duplicated in [flex layout](#) because the source order of the items (which is used for reading, sequential navigation, and one-column mobile phone layout) goes back and forth between the two columns.

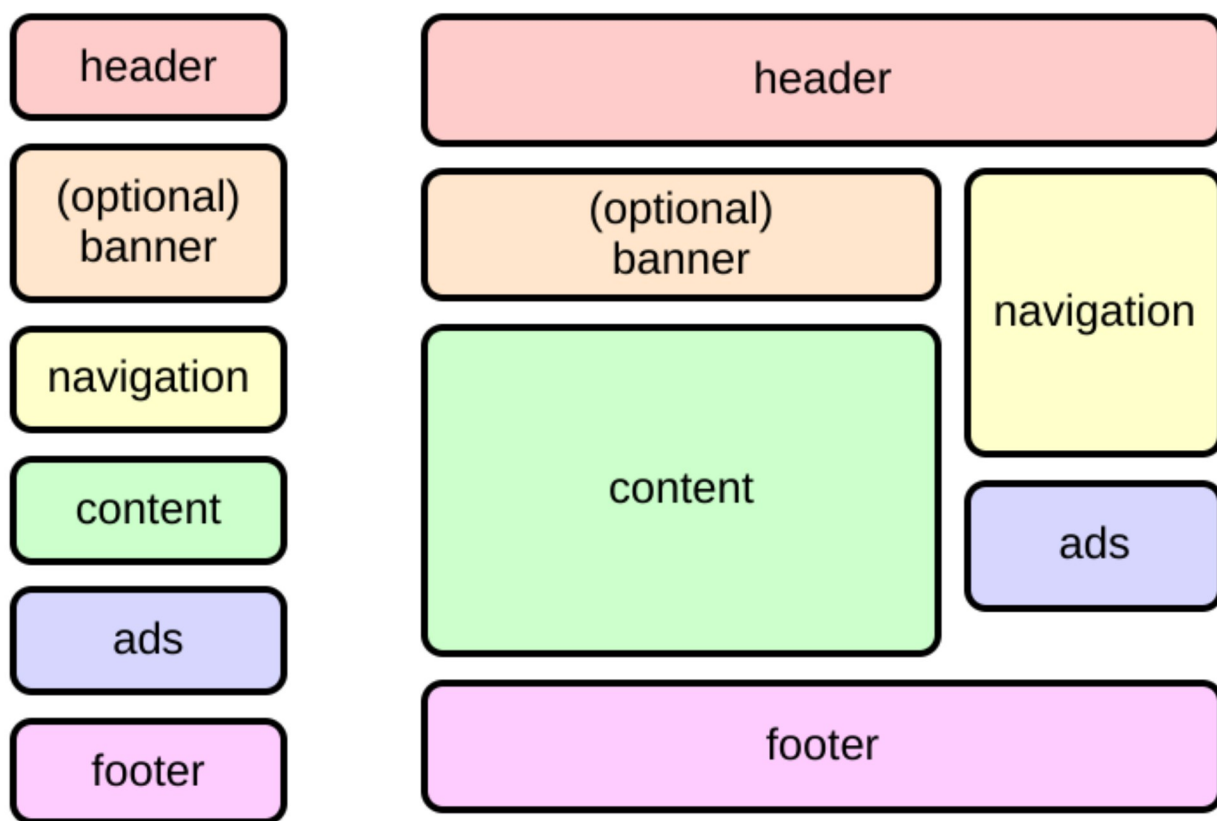


Figure 3 Comparison of one-column and two-column variants of a one-dimensional grid layout.

§ 1.2. Value Definitions

This specification follows the [CSS property definition conventions](#) from [\[CSS2\]](#) using the [value definition syntax](#) from [\[CSS-VALUES-3\]](#). Value types not defined in this specification are defined in CSS Values & Units [\[CSS-VALUES-3\]](#). Combination with other CSS modules may expand the definitions of these value types.

In addition to the property-specific values listed in their definitions, all properties defined in this specification also accept the [CSS-wide keywords](#) as their property value. For readability they have not been repeated explicitly.

§ 2. Grid Lanes Layout Model

Grid lanes layout lays out items into pre-defined tracks similar to [grid layout](#) in one axis (called the **grid axis**), but flows them freely similar to [flex layout](#) in the other (called the **stacking axis**). Similar to grid layout and unlike flex layout, [grid lanes layout](#)'s auto-placement distributes items across the tracks to keep the lengths of those tracks as similar as possible.

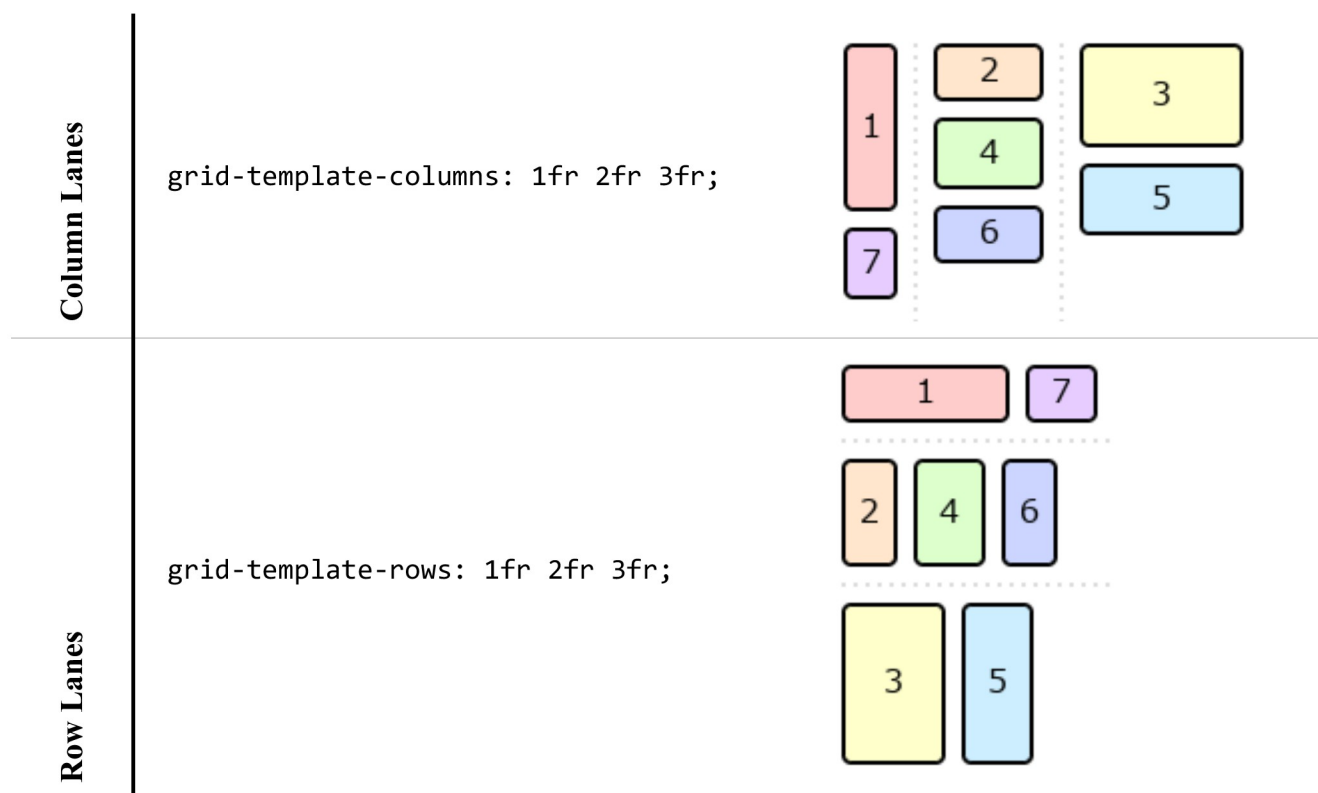
[Grid items](#) are formed and [blockified](#) exactly the same as in a regular [grid container](#).

All CSS properties work the same as in a regular [grid container](#) unless otherwise specified by this specification. For example, [‘order’](#) can be used to specify a different layout order for the items.

NOTE: Subgrid items are supported, but subgridding only occurs in the [grid axis](#); see [§ 3.2 Subgrids](#) for details.

A **grid lanes container** is a [grid container](#) whose contents participate in [grid lanes layout](#). A [grid lanes container](#) creates column [tracks](#) if its [stacking axis](#) is the [block axis](#), or row tracks if its stacking axis is the [inline axis](#).

Comparing Grid Lanes Containers



§ 2.1. Reordering and Accessibility

Although [grid lanes layout](#) generally progresses in a forwards fashion (placing the next item endward of the current item in at least one axis, matching the natural “reading order”), it can switch between endward in the inline or block axis in a seemingly arbitrary manner. In simple cases, the [‘flow-tolerance’](#) property can help reduce the feeling of backtracking due to small sizing differences in the [stacking axis](#) when laying out auto-placed items. But when [auto-placement](#) is mixed with [explicit placement](#) or spanning items, some amount of backtracking may occur.

EXAMPLE 3

For example, in the following markup sample, the fourth item is a spanner that doesn't fit in the remaining empty column on the first line. It ends up positioned into the first column, which is the highest available space into which it will fit. The next few items, which have a span of 1, end up laying out “above” it in the empty column, violating the natural reading order.

```
<section class=masonry>
  <div class=item>1</div>
  <div class=item>2</div>
  <div class="item tall">3</div>
  <div class="item wide">4</div>
  <div class=item>5</div>
  <div class=item>6</div>
  <div class=item>7</div>
</section>
<style>
.masonry {
  display: grid-lanes;
  grid-template-columns: repeat(5, auto);
}
.item { height: 50px; }
.item.wide { grid-column: span 3; }
.item.tall { height: 90px; }
</style>
```

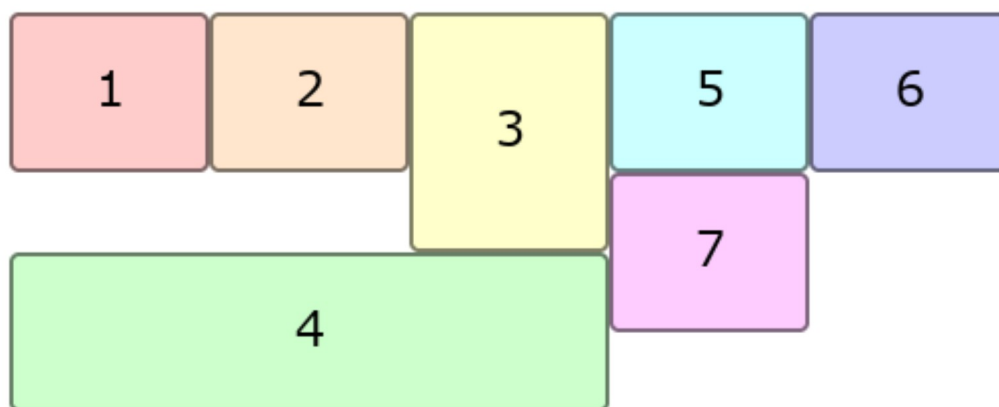


Figure 4 Auto-placed grid lanes layout with mixed-height items and mixed span sizes

Similarly, items explicitly placed into specific tracks can leave gaps behind them, into which subsequent auto-placed items can be placed visually out-of-order.

Authors should be aware of these possibilities and design layouts where such backtracking is minimized so that focus and reading order can be more easily followed. Alternatively, if the items do not have an inherent order, use the [‘reading-flow’](#) property to allow the UA to re-order the items for reading and linear navigation.

ISSUE 1 Or should reordering be the default behavior for auto-placed items here?



Techniques for reducing backtracking include:

- Using appropriate values for [‘flow-tolerance’](#), i.e. values large enough to avoid gratuitous differentiation among similarly-sized tracks, but not so large that meaningful differences get ignored.
- Using explicit placement in ways that help group related items together, rather than ways that disrupt the natural order of items
- Avoiding the combination of mixed span sizes in the [grid axis](#) and disparate item sizes in the [stacking axis](#), which can cause items to get pulled out of order (see example above).

As with [grid layout](#) and [flex layout](#) authors can use the [‘order’](#) property to re-order items; the same caveats apply. See [CSS Grid Layout 2 § 4 Reordering and Accessibility](#) and [CSS Display 4 § 3.1 Reordering and Accessibility](#).

§ 2.2. Establishing Grid Lanes Layout

Name: [‘display’](#)

New grid-lanes | inline-grid-lanes

values:

‘grid-lanes’

This value causes an element to generate a [block-level grid lanes container](#) box.

‘inline-grid-lanes’

This value causes an element to generate an [inline-level grid lanes container](#) box.

A [grid lanes container](#) that is not [subgridded](#) in its [grid axis](#) establishes an [independent formatting context](#) for its contents.

ISSUE 2 ISSUE(12820): Write up how grid/masonry formatting contexts work more formally? 

§ 2.3. Orienting Grid Lanes Layout

The orientation of a [grid lanes container](#), i.e. whether its [grid axis](#) is the [inline axis](#) (establishing columns) or the [block axis](#) (establishing rows) is determined by the **ISSUE 3** [TBD](#) property.

The [initial value](#) of this property is *‘normal’*, which determines the orientation from the *‘grid-template-*’* properties:

- If *‘grid-template-columns’* is *‘none’* and *‘grid-template-rows’* is not *‘none’*, the [grid container](#)’s [block axis](#) is the [grid axis](#) (establishing rows).
- Otherwise (thus by default), the [grid container](#)’s [inline axis](#) is the [grid axis](#) (establishing columns).

EXAMPLE 4

The following code will create a 2-column (“waterfall style”) [grid lanes container](#) that grows downward:

```
‘.container {
  display: grid-lanes;
  grid-template-columns: 100px 200px;
}
```

while the following code will create a 2-row (“brick wall style”) [grid lanes container](#) that grows horizontally:

```
‘.container {
  display: grid-lanes;
  grid-template-rows: 100px 200px;
}
```

ISSUE 4 Figure out whether we are re-using *‘grid-auto-flow’* here (and what it’s values mean)  or defining a new property like *‘grid-lanes-direction’*. [\[Issue #12803\]](#)

§ 3. Grid Lanes Track Specification

In the [grid axis](#), the full power of [grid layout](#) is available for track specification:

- Track sizes, line names, and areas can be specified on the [grid lanes container](#)’s [grid axis](#), just like in [grid layout](#).
- The [explicit grid](#) and [implicit grid](#) are formed in the same way as for a regular [grid container](#).
- Items can be [placed](#) against these grid templates just as in [grid layout](#).

However, auto-placed items contribute sizing to all tracks, not just the track into which they are ultimately placed; see [§ 3.4 Grid Axis Track Sizing](#).

NOTE: This is because auto-placed items must be laid out *as* they are placed, so that each track knows how “full” it is (and therefore which track should receive the next auto-placed item); thus, the tracks themselves must already have a definite size so that the items know their [available space](#) during layout.

§ 3.1. Declaring Grid Lanes Track Templates: the ‘grid-template-*’ properties

The ‘[grid-template-*](#)’ and ‘[grid-auto-rows](#)’/‘[grid-auto-columns](#)’ properties (and their shorthands) apply in the [grid axis](#) of the [grid lanes container](#) and establish tracks just as on regular [grid containers](#). (They are ignored in the [stacking axis](#).)

§ 3.1.1. Intrinsic Tracks and repeat()

Level 3 extends the ‘[repeat\(\)](#)’ notation to allow repetitions where neither the [min track sizing function](#) nor the [max track sizing function](#) is [definite](#); in other words, the syntax for [<auto-repeat>](#) is relaxed to the following:

[<auto-repeat>](#) = repeat([[auto-fill](#) | [auto-fit](#)] , [[<line-names>?](#) [<track-size>](#)]+)

In order to resolve the number of repetitions, a hypothetical size is calculated for such tracks by initializing the track sizes (per [CSS Grid Layout 2 § 12.4 Initialize Track Sizes](#)) and resolving intrinsic track sizes (per [CSS Grid Layout 2 § 12.5 Resolve Intrinsic Track Sizes](#)) in accordance with [§ 3.4 Grid Axis Track Sizing](#) with the following assumptions:

- Ignore explicit item placement. (That is, assume all items have an [automatic position](#).)
- Do not [collapse](#) any tracks.

- Expand the repeated track listing to capture all possible [automatic placements](#) of each item, i.e. repeat the track listing $2 + (\text{largest span} - 2) / (\text{number of tracks in repeat}())$ times, rounded *down* to a whole number.

The hypothetical size of each track in the `'repeat()'` listing is given by the largest track corresponding to that entry (by index) in the listing.

EXAMPLE 5

For example, given a template of `'repeat(auto-fill, 50px auto auto)'` and a largest spanner of 2, you need to repeat the track listing twice, giving `'50px auto auto 50px auto auto'`. After doing the hypothetical layout, the 2nd and 5th tracks are maxed together to provide a hypothetical size for the first `'auto'` and the 3rd and 6th tracks are maxed together to provide a hypothetical size for the second `'auto'`. Those concrete sizes are then used to determine how many repetitions will fill the container.

ISSUE 5 Should this work also in Grid Layout somehow, or fall back to a single repetition? If so, how? [\[Issue #10915\]](#)

NOTE: This simplified layout heuristic is defined to be "good enough", while remaining fast and consistent. Ignoring placement is required just to make the concept coherent; before you know how many repetitions you need, you can't tell what track an item (even one with a definite placement) will end up in.

ISSUE 6 Technically, if your explicit placement does not enter or cross over the repetition, we can know its placement prior to resolving the repetition. Do we want to adjust the algorithm above to account for this? That is, given `'auto repeat(auto-fill, ...) auto'`, if you explicitly place an item against line 1 or -1, we could let it only affect that track (rather than pretending it'll go in every track).

3.2. Subgrids

[Subgridding](#) allows nested [grid lanes containers](#) (and [grid containers](#)) to share track sizes. If the parent's corresponding axis is a [grid axis](#), the subgridded axis is taken from the parent container [as specified for grid containers](#); if the parent's corresponding axis is a [stacking axis](#), the subgridded axis also acts as a stacking axis.

ISSUE 7 What if this conflicts with the lanes orientation, or results in both axes stacking?

In [grid lanes layout](#), auto-placed [subgrids](#) don't inherit any line names from their parent grid, because that would make the placement of the item dependent on layout results; but the subgrid's tracks are still aligned to the parent's tracks as usual.

EXAMPLE 6

Here's a subgrid [example](#):

```
<style>
.grid {
  display: inline-grid-lanes;
  grid-template-rows: auto auto 100px;
  align-content: center;
  height: 300px;
  border: 1px solid;
}

.grid > * {
  margin: 5px;
  background: silver;
}

.grid > :nth-child(2n) {
  background: pink;
}

.grid subgrid {
  display: grid;
  grid: subgrid / subgrid;
  grid-row: 2 / span 2;
  grid-gap: 30px;
}

.grid subgrid > * { background: cyan; }
</style>

<div class="grid">
  <item>1</item>
  <item>2</item>
  <item>3</item>
  <subgrid>
    <item style="height:100px">subgrid.1</item>
    <item>sub.2</item>
    <item>s.3</item>
  </subgrid>
  <item>4</item>
  <item>5</item>
  <item style="width: 80px">6</item>
  <item>7</item>
```

</div>

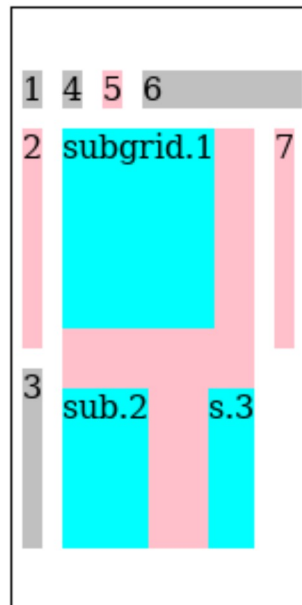


Figure 5 The rendering of the subgrid example above.

Note how the subgrid's first item ("subgrid.1") contributes to the intrinsic size of the 2nd row in the parent grid. This is possible since the subgrid specified a definite position so we know which tracks it will occupy. Note also that trying to subgrid the parent's [stacking axis](#) results in the subgrid converting to a [grid lanes container](#) with its [inline axis](#) as the stacking axis.

A [subgrid](#) that is a [grid lanes container](#) can be referred to as a **grid lanes subgrid**. To distinguish, a regular [grid container](#) subgrid can be referred to as a **regular grid subgrid**.

§ 3.3. Track Repetition: the `'repeat()'` notation

This specification introduces new keywords and grid-lanes-specific behavior for the `'repeat()'` notation.

§ 3.3.1. `repeat(auto-fit)`

In [grid lanes containers](#) (as in regular [grid containers](#)) `'auto-fit'` acts like `'auto-fill'`, but with empty tracks [collapsed](#). However, because placement occurs after track sizing, grid lanes containers use a

heuristic to determine if a track will be occupied:

- All tracks occupied by explicitly placed items are considered occupied.
- With the sum of the spans of all auto-placed items as N , all unoccupied tracks up to the N th such track are considered occupied.

All tracks produced by the ‘[auto-fit](#)’ repetition and considered unoccupied by this heuristic are assumed “empty” and are [collapsed](#). A collapsed grid track cannot accept placement of auto-placed items.

NOTE: It is possible for an auto-placed item to be placed in a track when ‘[auto-fill](#)’ is used that would be collapsed if ‘[auto-fit](#)’ is used if there are auto-placed items with a span greater than 1 mixed with explicitly-placed items that leave gaps too small for the auto-placed items.

§ 3.4. Grid Axis Track Sizing

Track sizing works the same as in [CSS Grid](#), except that when considering which items contribute to intrinsic sizes:

- All items explicitly placed in that track contribute, and
- All items with an [automatic grid position](#) contribute (regardless of whether they are ultimately placed in that track).

EXAMPLE 7

For example, suppose there are two columns in the [grid axis](#) and that

- Items A, B, and C have no explicit position.
- Item D is explicitly placed into the first column.

In this case, items A, B, C, and D all contribute to sizing the first column, while only A, B, and C (and not D) contribute to the second column.

In the case of spanning items with an [automatic grid position](#), they are assumed to be placed at every possible start position, and contribute accordingly.

EXAMPLE 8

For example, suppose there are 5 columns in the [grid axis](#), with the middle having a fixed size of ‘100px’ and the other two being ‘auto’-sized. For the purpose of track sizing, an item that spans 2 tracks and has an intrinsic contribution of 220px is essentially copied and assumed to exist:

- At grid line 1, contributing 110px to each of the first two tracks.
- At grid line 2, contributing 120px to the second track.
- At grid line 3, contributing 120px to the fourth track.
- At grid line 4, contributing 110px to the fourth and fifth tracks.

NOTE: This algorithm ensures that each track is at least big enough to accommodate every item that is ultimately placed in it, and does not create dependency cycles between placement and track sizing. However, depending on the variation in sizes, tracks could be larger than necessary: an exact fit is only guaranteed if all items are explicitly placed in the [grid axis](#) or all items are the same size (or matching multiples of that size, in the case of spanning items).

§ 3.4.1. Subgrid Item Contributions

When sizing the tracks of either a regular [grid container](#) or a [grid lanes container](#), a [grid lanes subgrid](#) has special handling of items that have an [automatic grid position](#):

- Any such item is placed into every possible [parent grid](#) track that could be spanned by the [grid lanes subgrid](#). (If the subgrid has a [definite grid position](#), thus only the spanned tracks; if it has an [automatic grid position](#), then all tracks in the parent grid.)
- Any such item receives the largest margin/border/padding contribution of each edge at which it could hypothetically be placed. If the item spans the entire subgrid, it receives both. (See [CSS Grid Layout §9](#).)

When sizing the tracks of a [grid lanes container](#), a [subgrid](#) with an [automatic grid position](#) (whether a [regular grid subgrid](#) or [grid lanes subgrid](#)) also has special handling of its items:

- *Every* item is placed into every possible [parent grid](#) track that could be spanned by the [subgrid](#), (ignoring any explicit placement of the item).
- Explicitly-placed sub-items only receive the subgrid’s margin/border/padding contribution when their explicit placement is at an edge of the subgrid (as usual).

NOTE: Explicitly placed items inside an explicitly placed [subgrid](#) only ever contribute to their actual [parent grid](#) tracks. This is because in this case, just like in regular [grid layout](#), the item’s placement is known prior to track sizing.

§ 3.4.2. Optimized Track Sizing

Track sizing can be optimized by aggregating items that have the same span size and placement into a single virtual item as follows:

1. Separate all the [grid items](#) into *item groups*, according to the following properties:
 - the span of the item
 - the placement of the item, i.e. which tracks it is allowed to be placed in
 - the item’s [baseline-sharing group](#)

NOTE: For example, an item with span 2 placed in the second track will be in a different group than an item with span 2 that has an [automatic grid position](#).

2. For each [item group](#), synthesize a *virtual grid item* that has the maximum of every intrinsic size contribution among the items in that group.

If the items apply [baseline alignment](#), determine the baselines of the [virtual grid item](#) by placing all of its items into a single hypothetical grid track and finding their shared baseline(s) and shims. Increase the group’s intrinsic size contributions accordingly.

3. Place hypothetical copies of each [virtual grid item](#) into the [grid axis](#) tracks in every position that the item could potentially occupy, and run the [track sizing algorithm](#) with those items. The resulting track sizes are the [grid lanes container’s](#) track sizes.

NOTE: This optimization should give the same results as the track sizing description [above](#); if not this is an error, please [report it to the CSSWG](#).

§ 4. Grid Lanes Item Placement

In the [grid axis](#), items can be *explicitly placed* into tracks and span them using the familiar [grid-placement properties](#)’ syntax. Auto-placement, however, uses the [§ 4.4 Grid Lanes Layout and Placement Algorithm](#), placing each item with an [automatic grid position](#) into the “shortest” track

available.

EXAMPLE 9

Here's a grid lanes layout demonstrating explicitly placed and spanning items:

```
.container {  
  grid-template-columns: repeat(3, auto);  
}  
.container > :nth-child(2) {  
  /* auto-placed, but spanning. */  
  grid-column: span 2;  
}  
.container > :nth-child(3) {  
  /* manually placed */  
  grid-column: 3;  
}  
/* all other children are auto-placed */
```

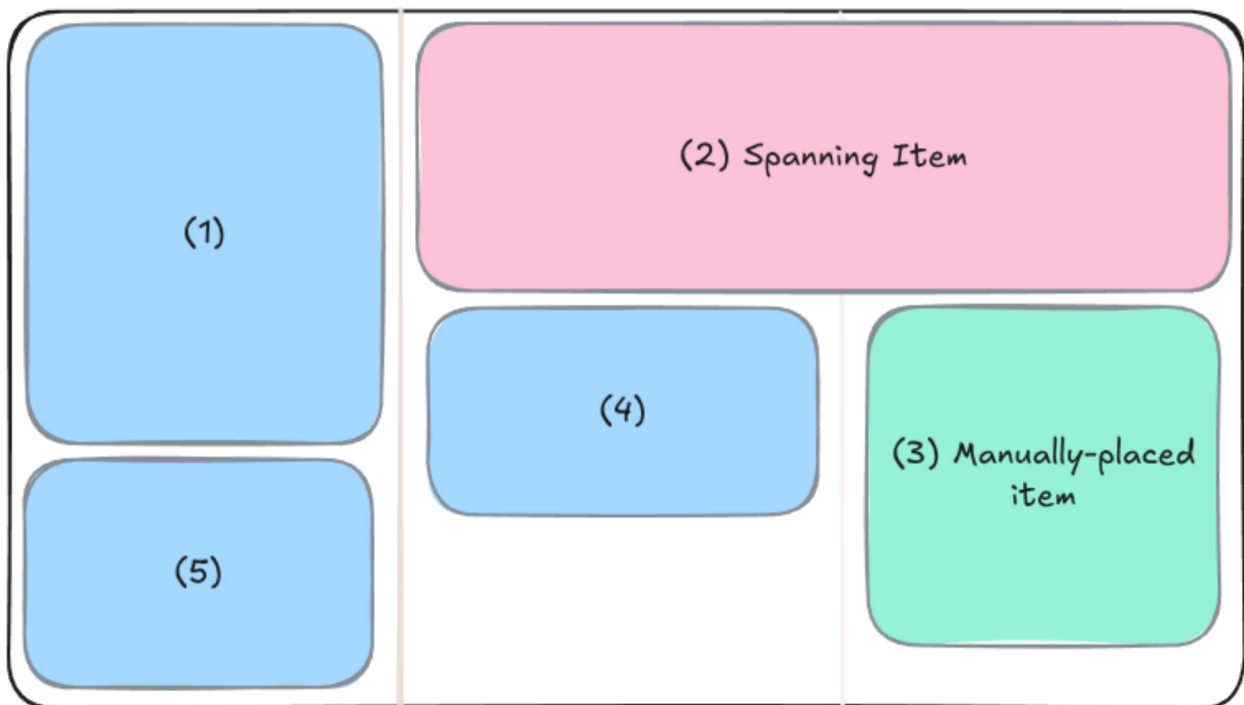


Figure 6 Rendering of the example above.

§ 4.1. Specifying Grid Axis Item Placement: the ‘grid-column-*’ and ‘grid-row-*’ properties

The ‘grid-column-*’ and ‘grid-row-*’ properties (and their shorthands) apply in the [grid axis](#) of the items and establish placement just as in regular [grid layout](#).

§ 4.2. Placement Precision: the ‘flow-tolerance’ property

<i>Name:</i>	‘flow-tolerance’
<i>Value:</i>	normal <length-percentage [0,∞]> infinite
<i>Initial:</i>	normal
<i>Applies to:</i>	grid lanes containers
<i>Inherited:</i>	no
<i>Percentages:</i>	relative to the grid-axis content box size of the grid lanes container
<i>Computed value:</i>	a computed <length-percentage> value
<i>Canonical order:</i>	per grammar
<i>Animation type:</i>	as length

[Grid lanes containers](#) are filled by placing each [grid item](#) in whichever [grid track](#) is currently the least filled. When multiple tracks are tied for least-filled, placing the items in order looks good. But if tracks are only *very slightly* different heights, it can look strange to have them not fill in order, as the height differences aren’t perceived as *meaningfully* different.

The ‘flow-tolerance’ property specifies what the threshold is for considering tracks to be “the same height”, causing them to fill in order.

‘<length-percentage>’

Specifies the *tie threshold* for the [grid lanes container](#). Placement positions are considered to be equally good (“tied”) if they are within the specified distance from the shortest position.

NOTE: The initial value is a “small” distance (`1em`) that is probably appropriate to represent “close enough”.

‘normal’

Resolves to a [used value](#) of `1em` in [grid lanes layout](#) and a used value of `0` in all other layout modes.

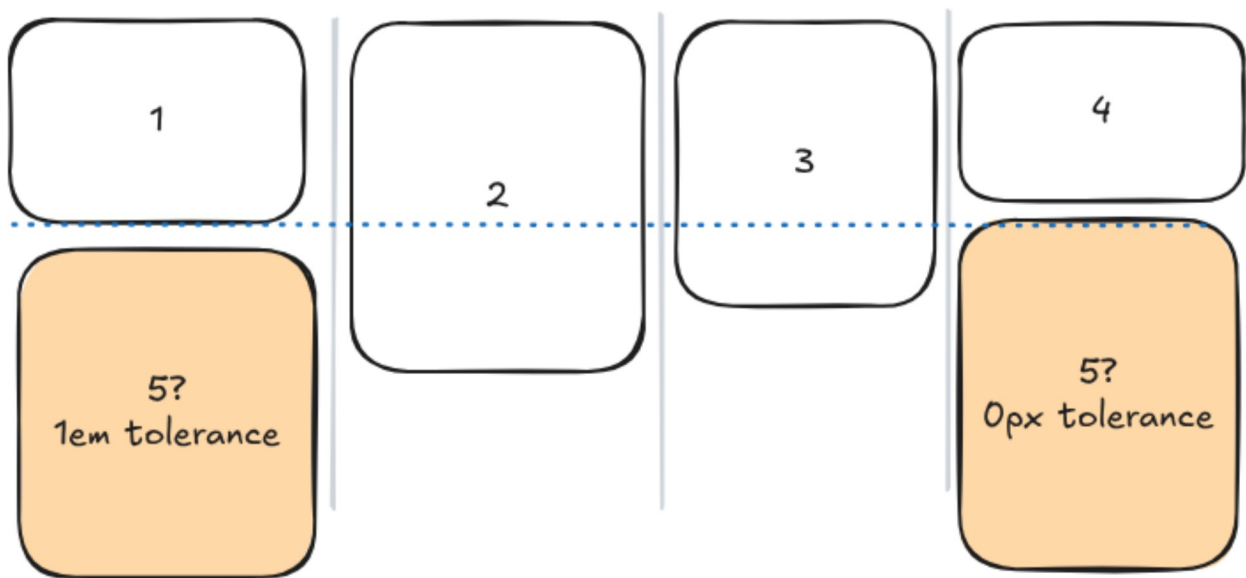
‘infinite’

Specifies an infinite [tie threshold](#). This makes items distribute themselves strictly in order, without considering the length of the tracks at all.

NOTE: This value can result in consecutive items being placed in dramatically different positions in the [stacking axis](#), which can be confusing to readers. If the initial value (`1em`) is too small, consider a larger value (such as `10em` or `50vh`) instead of `infinite`.

EXAMPLE 10

In the following example, when placing the 5th item, the fourth column is the shortest, but the first column is *almost* as short.



With the default tolerance of `1em`, both the first and fourth columns are considered to be “tied”, and so the first is chosen.

If, instead, the tolerance is set to `0px`, then there is no tie; only the fourth column is a possible placement.

NOTE: We expect to apply this property to [flex layout](#) in the future also, see [discussions](#) on how that might work.

§ 4.3. Dense Placement: the ‘dense’ keyword

In [grid layout](#), ‘[grid-auto-flow: dense](#)’ allows backtracking during the [grid item placement algorithm](#). The ‘dense’ keyword similarly allows backtracking during the [grid lanes placement algorithm](#). However, because item placement and sizing are intertwined in [grid lanes layout](#), an item can only backtrack into a compatible empty slot if the total used size of that slot’s tracks matches the used size of the item’s normal-placement tracks.

NOTE: This restriction avoids laying out the item more than once.

§ 4.4. Grid Lanes Layout and Placement Algorithm

For each of the tracks in the [grid axis](#), keep a **running position** initialized to zero. Maintain also a **auto-placement cursor**, initially pointing to the first line.

For each item in [order-modified document order](#):

1. If the item has a [definite grid position](#) in the [grid axis](#), use that placement.

ISSUE 8 Should this also update the placement cursor?



Otherwise, resolve its [grid axis](#) placement using these substeps:

1. Starting at the first [grid axis](#) line in the [implicit grid](#), find the largest [running position](#) of the grid axis tracks that the item would span if it were placed at this line, and call this position *max_pos*.
2. Repeat the previous step for each successive line number until the item would no longer fit inside the grid.
3. Let *possible lines* be the line that resulted in the smallest *max_pos*, and all lines that result in a *max_pos* within the [tie threshold](#) of this *max_pos*.
4. Choose the first line in *possible lines* greater than or equal to the [auto-placement cursor](#) as the item’s position in the [grid axis](#); or if there are none such, choose the first one.
5. Update the [auto-placement cursor](#) to point to item’s last line.

- Place the item in its [grid axis](#) tracks at the maximum of the [running positions](#) of the tracks it spans.
- Calculate the size of the item's [containing block](#) and then layout the item. Set the [running position](#) of the spanned [grid axis](#) tracks to $\text{max_pos} + \text{outer_size} + \text{'grid-gap'}$.

For this purpose, the [outer sizes](#) of the box are floored at zero.

NOTE: That is, if the box has sufficiently large negative margins, it won't somehow count as a negative *size* here. The [running position](#) of a track never decreases, so a negative-sized box won't cause future normally-sized items placed in the track to overlap previous items.

- If the [grid lanes container](#) uses [‘dense’](#) packing, and there exists skipped spaces in the layout (e.g. due to spanning items) into which the item, as it is sized now, could have fit if it were placed earlier, and where the spanned tracks have the same total used size as the tracks into which it is currently placed, then instead place it into the highest such space. If there are multiple valid spaces within the [tie threshold](#) of the highest space, place it in the [start](#)-most of them. Rewind the [auto-placement cursor](#) and the [running position](#) to their values before this item's placement.

NOTE: Items that *visually* intrude into preceding empty spaces (via negative margins, [‘position: relative’](#), transforms, etc.), do not affect the size of those empty spaces. Later items can get placed in those spaces and visually overlap these previous items.

NOTE: Dense packing both ignores the [auto-placement cursor](#) when backfilling, and does not update it after placement. If there aren't any acceptable placement gaps to backfill, though, it places items exactly as when [‘dense’](#) were not specified.

NOTE: This algorithm chooses the track that would result in the item being placed as highly as possible. If there are ties, it chooses the earliest such track, *after* the most recently placed item if possible (ensuring that it always “moves forward” even in the presence of ties).

§ 4.4.1. Containing Block

The [containing block](#) for a [grid item](#) participating in [grid lanes layout](#) is formed by its [grid area](#) in the [grid axis](#) and the [grid container](#)'s [content box](#) in the [stacking axis](#).

§ 4.4.2. Placement and Writing Modes

NOTE: Like all of [grid layout](#), grid lanes layout and placement is sensitive to the [writing mode](#). For example, for [‘direction: rtl’](#), items are placed right-to-left rather than left-to-right, whether the inline axis is a [grid axis](#) or a [stacking axis](#).

EXAMPLE 11

Here’s a simple [example](#) using [‘direction: rtl’](#) in the [grid axis](#):

```
<style>
.grid {
  display: inline-grid-lanes;
  direction: rtl;
  grid-template-columns: repeat(4, 2ch);
  border: 1px solid;
}

.item { background: silver }
.item:nth-child(2n+1) {
  background: pink;
  height: 4em;
}
</style>

<div class="grid">
  <item>1</item>
  <item style="grid-column:span 2">2</item>
  <item>3</item>
  <item>4</item>
</div>
```

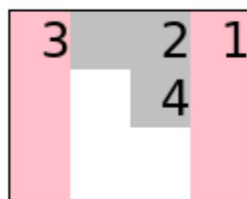


Figure 7 Rendering of the [‘direction: rtl’](#) example above.

EXAMPLE 12



Here's a simple [example](#) using `'direction: rtl'` in the [stacking axis](#):

```
<style>
.grid {
  display: inline-grid-lanes;
  direction: rtl;
  width: 10ch;
  column-gap: 1ch;
  grid-template-rows: repeat(4, 2em);
  border: 1px solid;
}

.item { background: silver; }
.item:nth-child(2n+1) {
  background: pink;
  width: 4ch;
}
</style>

<div class="grid">
  <item>1</item>
  <item style="grid-row:span 2">2</item>
  <item>3</item>
  <item>4</item>
</div>
```

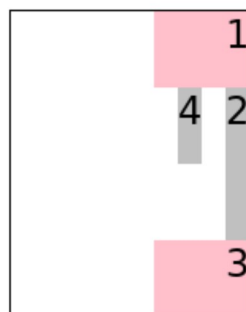


Figure 8 Rendering of the `'direction: rtl'` example above.

§ 5. Sizing Grid Containers

Sizing Grid Containers works the same as for regular grid containers but with the following replacement for the stacking axis: The max-content size (min-content size) of a grid container in the stacking axis is the size of the stacking range when the grid lanes container is sized under a max-content constraint (min-content constraint) in that axis. The ***stacking range*** is the range between the startmost outer edge among the first items of each track and the endmost outer edge among the last items of each track.

EXAMPLE 13

Here's a simple [example](#):

```
<style>
.grid {
  display: inline-grid;
  grid-template-columns: 50px 100px auto;
  grid-gap: 10px;
  border: 1px solid;
}
item { background: silver; margin: 5px; }
</style>

<div class="grid">
  <item style="border:10px solid">1</item>
  <item>2</item>
  <item>3</item>
  <item style="height:50px">4</item>
  <item>5</item>
  <item>6</item>
</div>
```

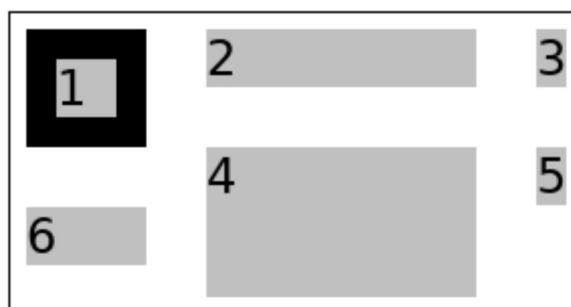


Figure 9 Rendering of the [grid container](#) intrinsic sizing example above.

§ 6. Alignment and Spacing

§ 6.1. Gutters: the ‘row-gap’, ‘column-gap’, and ‘gap’ properties

[Gutters](#) are supported in both axes using the ‘[row-gap](#)’ and ‘[column-gap](#)’ properties (and their ‘[gap](#)’

[shorthand](#)). In the [grid axis](#) they work the same as in a regular [grid container](#); see [CSS Grid Layout 2 § 11. Alignment and Spacing](#). In the [stacking axis](#), the gap is applied between the margin boxes of each pair of adjacent items. Margins do not collapse in either axis.

§ 6.2. Grid-axis Alignment: the ‘[align-content](#)’/‘[justify-content](#)’, ‘[align-self](#)’/‘[justify-self](#)’, and ‘[align-items](#)’/‘[justify-items](#)’ properties

In the [grid axis](#), the [box alignment properties](#) work the same as in a regular [grid container](#). See [CSS Grid Layout 2 § 11. Alignment and Spacing](#) and [CSS Box Alignment Level 3](#).

§ 6.3. Stacking-axis Content Distribution: the ‘[align-content](#)’/‘[justify-content](#)’ properties

In the [stacking axis](#), [content-distribution](#) is applied to the content as a whole, similarly to how it behaves in block containers. More specifically, the [alignment subject](#) is the [stacking range](#).

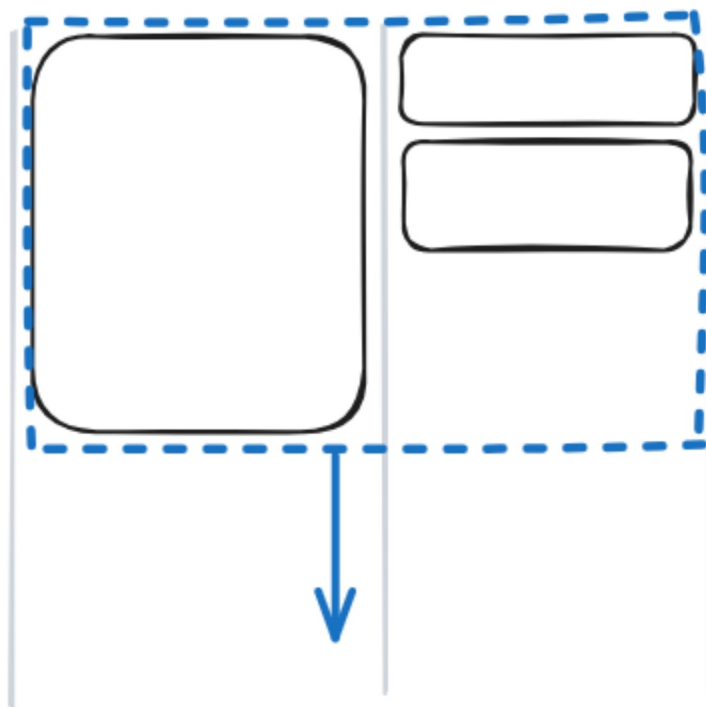


Figure 10 The extent of the alignment subject is indicated by the dashed border, while the alignment container is the whole container (indicated by the height of the column dividers). It defaults to start-aligning, but depending on (in this case) ‘[align-content](#)’, the items can move down, as a block, to center- or end-align vertically.

NOTE: There is only ever one [alignment subject](#) for these properties in the [stacking axis](#), so the unique [‘align-content’](#) / [‘justify-content’](#) values boil down to [‘start’](#), [‘center’](#), [‘end’](#), and [baseline alignment](#). (The behavior of [‘normal’](#) and [‘stretch’](#) is identical to [‘start’](#), and the [distributed alignment](#) values behave as their [fallback alignments](#).) If the [grid items](#) overflow the [grid container](#)’s [content box](#) in the stacking axis, then the [stacking range](#) will be larger than the grid container’s content box.

§ 6.4. Stacking-axis Self Alignment: the [‘align-self’](#)/[‘justify-self’](#) and [‘align-items’](#)/[‘justify-items’](#) properties

In the [stacking axis](#), the [self-alignment properties](#) only apply to items that are adjacent to a “gap” in the layout, i.e. placed in their [grid track\(s\)](#) either immediately before a spanning item or as the last item. The [alignment subject](#) is the item’s [margin box](#), and the [alignment container](#) is that box plus the adjacent “gap”. For the last item in a track, the alignment container extends to the lowest bottom outer edge among all the last items in all tracks.

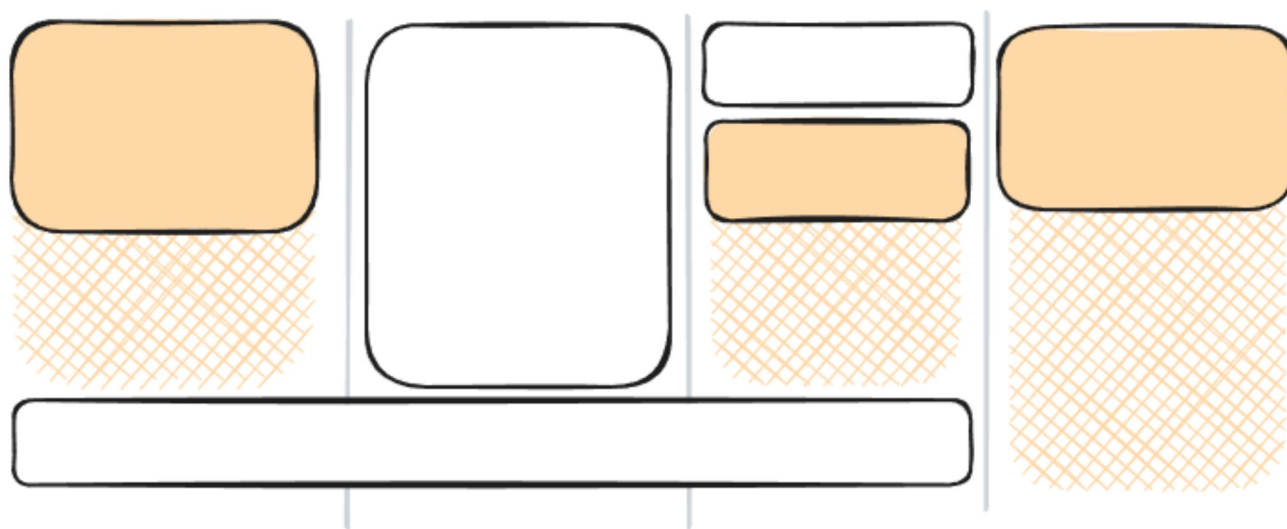


Figure 11 The three highlighted items are the only ones with “gaps” following them, so they’re the only ones that will respond to [‘align-items’](#). Their [alignment containers](#) are indicated by the cross-hatched areas.

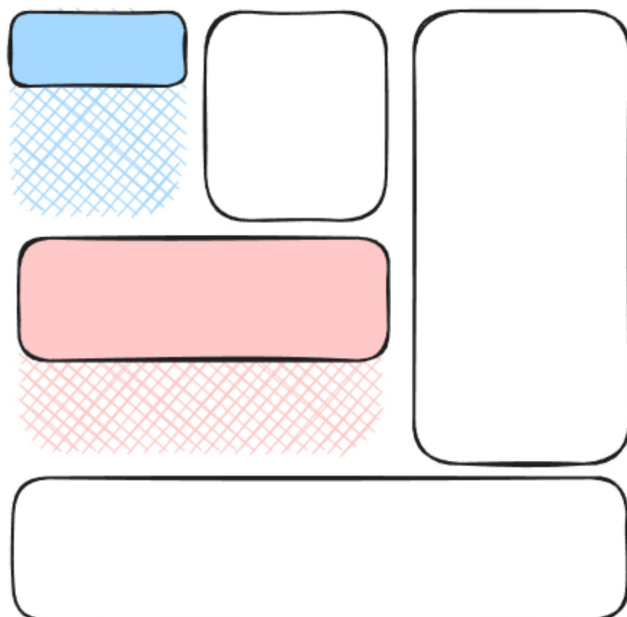
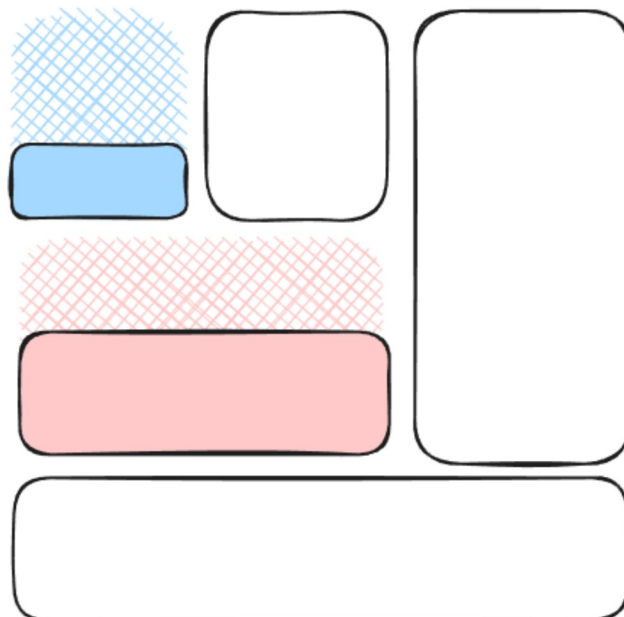
‘align-self: start’‘align-self: end’

Figure 12 Alignment of an item does not affect the size or position of other items' alignment containers.

§ 6.5. Baseline Alignment

Item baseline alignment inside the grid axis tracks works as usual for a regular grid container, and the grid container's baseline is determined the same as for a regular grid container in that axis.

Baseline alignment is not supported in the stacking axis. The baseline alignment values of the self-alignment properties on grid items are treated as their fallback alignment in this axis. The first baseline set of the grid container in this axis is generated from the highest alignment baseline among the grid items placed first in each track, and the last baseline set from the lowest alignment baseline among the grid items placed last in each track.

NOTE: Whether an item is “first” or “last” depends only on placement (including ‘dense’ packing adjustments), and is not affected by its actual position on the page, e.g. negative margins or relative positioning are not considered.

§ 7. Fragmentation

§ 7.1. Fragmentation in the stacking axis

Each grid axis track is fragmented independently in the stacking axis. If a grid item is fragmented, or

has a [forced break](#) before/after it, then the [running position](#) for the tracks that it spans in the grid axis are set to the size of the [fragmentainer](#) so that no further items will be placed in those tracks. An item that is split into multiple fragments retains its placement in the grid axis for all its fragments. A grid item that is pushed, however, is placed again by the next [grid container](#) fragment. Placement continues until all items are placed or pushed to a new fragment.

EXAMPLE 14

Here's an [example](#) illustrating fragmentation of grid lanes layout with [stacking](#) in its [block axis](#). It renders like this:

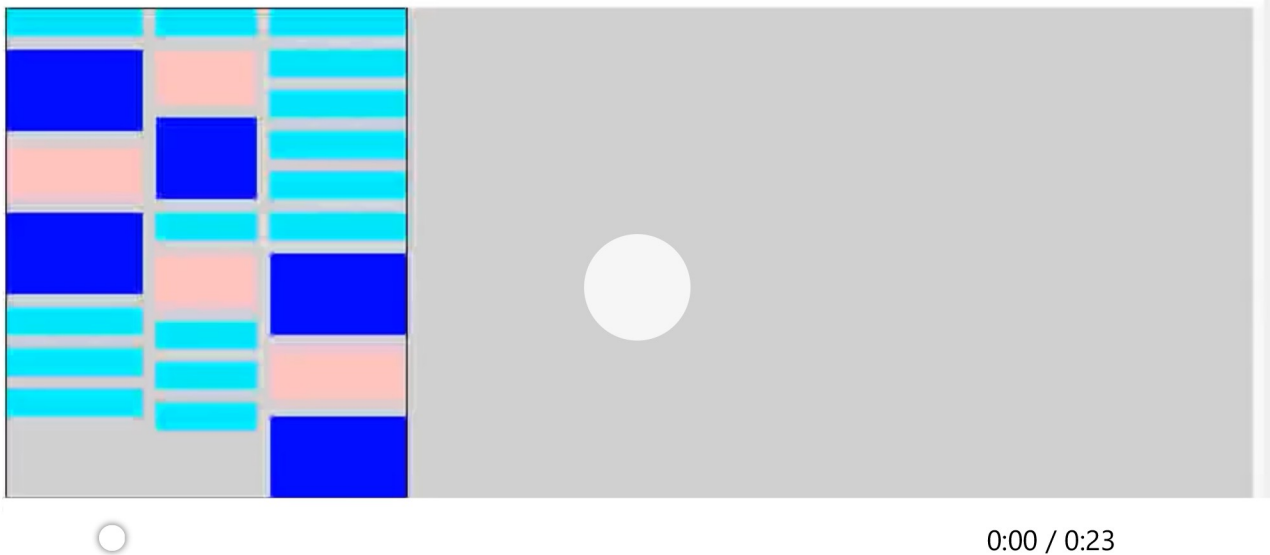


Figure 13 Visualization of fragmentation in a [block-axis grid lanes layout](#).

§ 7.2. Fragmentation in the Grid Axis

Fragmentation in the [grid axis](#) of a [grid lanes container](#) is also supported. In this case the fragmentation behaves more like in a regular [grid container](#); however, there's a separate step to determine which grid-axis track each item is placed into, before fragmentation occurs.

EXAMPLE 15

Here's an [example](#) illustrating fragmentation of grid lanes layout with [stacking](#) in its [inline axis](#). In this case the breaks occurs between the [grid-axis](#) rows. It renders like this:



Figure 14 Visualization of fragmentation in the [block axis](#) with [inline-axis grid lanes layout](#).

§ 8. Absolute Positioning

[Grid-aligned absolute-positioned descendants](#) are supported in [grid lanes containers](#) just as for regular [grid containers](#); however, in the [stacking axis](#) there exist only two lines (in addition to the ‘auto’ lines) for placement:

- line 1 (line -2) corresponds to the start edge of the [stacking range](#)
- line 2 (line -1) corresponds to the end edge of the [stacking range](#)

ISSUE 9 It might be useful to define a static position in the [stacking axis](#). Maybe it could be defined as the max (or min?) current [running position](#) of the [grid-axis](#) tracks at that point? Or the end of the item before it?

§ 9. Graceful Degradation

Typically, a grid lanes design can be expected to degrade quite nicely in a UA that supports [grid layout](#) but not [grid lanes layout](#).

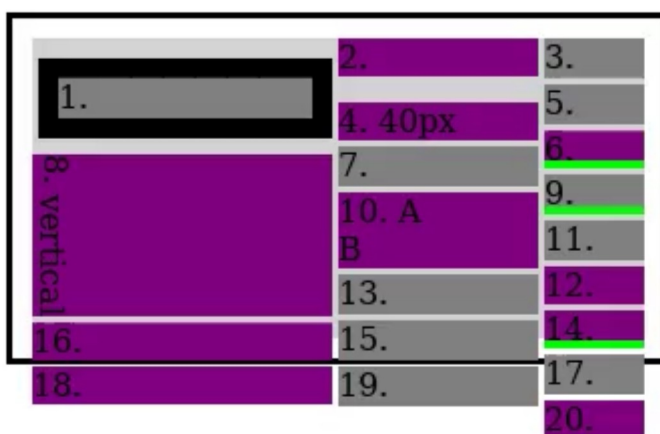
EXAMPLE 16



Here's an [example](#) to illustrate this.

```
display: grid;
display: grid-lanes; /* ignored in UAs that don't support grid lanes layout */
grid-template-columns: 150px 100px 50px;
```

This creates a layout with three columns, but will have "more gaps" in the [block axis](#) if the UA doesn't support [grid lanes layout](#). Here's what it looks like with grid lanes support for comparison:



0:00 / 0:08

Figure 15 Rendering of the example in a UA with grid lanes support.

§ 10. Acknowledgements

Special thanks goes to Cameron McCormack who wrote a masonry layout explainer document (from which was lifted the Background chapter) and presented it to the CSSWG, and to Mats Palmgren who developed the original version of this specification alongside a prototype implementation in Firefox. Thanks also to everyone who provided feedback on the [initial proposal](#) for this feature as well as the [many subsequent issues](#), and to Noam Rosenthal particularly for [naming it](#).

§ 11. Security Considerations

As a layout specification, this spec introduces no new security considerations beyond that exposed by CSS layout in general.

§ 12. Privacy Considerations

As a layout specification, this spec introduces no new privacy considerations beyond that exposed by CSS layout in general.

§ Changes

§ Additions Since Level 2

The following features have been added since [Level 2](#):

- Added [grid lanes layout](#).
- Expanded ‘repeat(auto-fill)’ and ‘repeat(auto-fit)’ to accept indefinite track sizing functions. ([Issue 9321](#), [Issue 12899](#))
- Introduced the ‘[flow-tolerance](#)’ property.

§ Recent Changes

The following changes have been made since the [21 January 2026 Working Draft](#):

- Dropped the ‘item-*’ proposal for unified flow controls across layout modes. ([Issue 11480](#))
- Simplified the sizing calcs for automatic-positioned [subgrids](#) in a [grid lanes container](#). ([Issue](#)

[10926](#))

- Clarify that baseline values for [self-alignment properties](#) in the stacking axis fall back to their fallback value. ([Issue 10275](#))

The following changes have been made since the [23 December 2025 Working Draft](#):

- Renamed ‘[item-tolerance](#)’ to ‘[flow-tolerance](#)’. ([Issue 10884](#))

The following changes have been made since the [16 December 2025 Working Draft](#):

- Defined ‘[normal](#)’ as the initial value for [setting the track orientation](#).
- Defined the [self-alignment properties](#)’ effect in the [stacking axis](#). See [§ 6.4 Stacking-axis Self Alignment: the align-self/justify-self and align-items/justify-items properties](#).

The following changes have been made since the [17 September 2025 Working Draft](#):

- Defined a new inner display type, ‘[grid-lanes](#)’, to establish [grid lanes layout](#), and updated spec vocabulary to match. ([Issue 12022](#))
- Dropped the unnecessary ‘[auto-areas](#)’ value from ‘[repeat\(\)](#)’. ([Issue 10854](#))
- Adjusted heuristic for ‘[repeat\(auto-fill\)](#)’ and ‘[repeat\(auto-fit\)](#)’ with indefinite track sizing functions. ([Issue 12899](#))
- Floored the contribution of items to a track’s [running position](#) at zero. ([Issue 12918](#))
- Renamed ‘[item-tolerance](#)’ to ‘[flow-tolerance](#)’. ([Issue 10884](#))

See also [earlier changes](#).

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [\[RFC2119\]](#)

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

EXAMPLE 17

This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

Note, this is an informative note.

Advisements are normative sections styled to evoke special attention and are set apart from other normative text with `<strong class="advisement">`, like this:

UAs MUST provide an accessible alternative.

▼ TESTS

Tests relating to the content of this specification may be documented in “Tests” blocks like this one. Any such block is non-normative.

§ Conformance classes

Conformance to this specification is defined for three conformance classes:

style sheet

A [CSS style sheet](#).

renderer

A [UA](#) that interprets the semantics of a style sheet and renders documents that use them.

authoring tool

A [UA](#) that writes a style sheet.

A style sheet is conformant to this specification if all of its statements that use syntax defined in this module are valid according to the generic CSS grammar and the individual grammars of each feature defined in this module.

A renderer is conformant to this specification if, in addition to interpreting the style sheet as defined by the appropriate specifications, it supports all the features defined by this specification by parsing them correctly and rendering the document accordingly. However, the inability of a UA to correctly render a document due to limitations of the device does not make the UA non-conformant. (For example, a UA is not required to render color on a monochrome monitor.)

An authoring tool is conformant to this specification if it writes style sheets that are syntactically correct according to the generic CSS grammar and the individual grammars of each feature in this module, and meet all other conformance requirements of style sheets as described in this module.

§ Partial implementations

So that authors can exploit the forward-compatible parsing rules to assign fallback values, CSS renderers **must** treat as invalid (and [ignore as appropriate](#)) any at-rules, properties, property values, keywords, and other syntactic constructs for which they have no usable level of support. In particular, user agents **must not** selectively ignore unsupported component values and honor supported values in a single multi-value property declaration: if any value is considered invalid (as unsupported values must be), CSS requires that the entire declaration be ignored.

§ Implementations of Unstable and Proprietary Features

To avoid clashes with future stable CSS features, the CSSWG recommends [following best practices](#) for the implementation of [unstable](#) features and [proprietary extensions](#) to CSS.

§ Non-experimental implementations

Once a specification reaches the Candidate Recommendation stage, non-experimental implementations are possible, and implementors should release an unprefixed implementation of any CR-level feature they can demonstrate to be correctly implemented according to spec.

To establish and maintain the interoperability of CSS across implementations, the CSS Working Group requests that non-experimental CSS renderers submit an implementation report (and, if necessary, the testcases used for that implementation report) to the W3C before releasing an unprefixed implementation of any CSS features. Testcases submitted to W3C are subject to review and correction by the CSS Working Group.

Further information on submitting testcases and implementation reports can be found from on the CSS Working Group's website at <http://www.w3.org/Style/CSS/Test/>. Questions should be directed to the

public-css-testsuite@w3.org mailing list.

§ Index

§ Terms defined by this specification

[auto-placement cursor](#), in § 4.4

[<auto-repeat>](#), in § 3.1.1

[flow-tolerance](#), in § 4.2

[grid axis](#), in § 2

[grid-axis](#), in § 2

[grid-lanes](#), in § 2.2

[grid lanes container](#), in § 2

[Grid lanes layout](#), in § 2

[grid lanes subgrid](#), in § 3.2

[infinite](#), in § 4.2

[inline-grid-lanes](#), in § 2.2

[item groups](#), in § 3.4.2

[<length-percentage>](#), in § 4.2

[normal](#)

[value for flow-tolerance](#), in § 4.2

[value for grid-auto-flow](#), in § 2.3

[regular grid subgrid](#), in § 3.2

[running position](#), in § 4.4

[stacking axis](#), in § 2

[stacking-axis](#), in § 2

[stacking range](#), in § 5

[tie threshold](#), in § 4.2

[virtual grid item](#), in § 3.4.2

§ Terms defined by reference

[CSS-ALIGN-3] defines the following terms:

- align-content
- align-items
- align-self
- alignment baseline
- alignment container
- alignment subject
- baseline alignment
- baseline-sharing group
- box alignment properties
- center
- distributed alignment
- end
- fallback alignment
- justify-content
- justify-items
- justify-self
- normal
- self-alignment properties
- start
- stretch

[CSS-BOX-4] defines the following terms:

- content box
- margin box
- outer edge

[CSS-BREAK-4] defines the following terms:

- forced break
- fragmentainer

[CSS-CASCADE-5] defines the following terms:

- initial value
- shorthand
- used value

[CSS-DISPLAY-4] defines the following terms:

- block-level
- blockify
- containing block
- display
- independent formatting context
- inline-level
- order
- order-modified document order
- reading-flow

[CSS-FLEXBOX-2] defines the following terms:

- flex layout



[CSS-GAPS-1] defines the following terms:

- column-gap
- gap
- grid-gap
- repeat()
- row-gap

[CSS-GRID-2] defines the following terms:

- <line-names>
- <track-size>
- auto-fill
- auto-fit
- automatic grid position
- automatic placement
- automatic position
- collapsed grid track
- definite grid position
- definite position
- dense
- explicit grid
- grid area
- grid container
- grid item
- grid layout
- grid track
- grid-auto-columns
- grid-auto-flow
- grid-auto-rows
- grid-placement property
- grid-template-columns
- grid-template-rows
- implicit grid
- max track sizing function
- min track sizing function
- none
- parent grid
- subgrid

[CSS-POSITION-3] defines the following terms:

position

[CSS-SIZING-3] defines the following terms:

- available space
- definite
- max-content constraint
- max-content size
- min-content constraint
- min-content size
- outer size

[CSS-VALUES-4] defines the following terms:

- +
- ,
- <length-percentage>
- ?
- CSS-wide keywords
- |

[CSS-WRITING-MODES-3] defines the following terms:

direction

[CSS-WRITING-MODES-4] defines the following terms:

- block axis
- block-axis
- inline axis
- inline-axis
- start
- writing mode

[CSS2] defines the following terms:

relative positioning

§ References

§ Normative References

[CSS-ALIGN-3]

Elika Etemad; Tab Atkins Jr.. *CSS Box Alignment Module Level 3*. URL: <https://drafts.csswg.org/css-align/>

[CSS-BOX-4]

Elika Etemad. *CSS Box Model Module Level 4*. URL: <https://drafts.csswg.org/css-box-4/>

[CSS-BREAK-4]

Rossen Atanassov; Elika Etemad. *CSS Fragmentation Module Level 4*. URL: <https://drafts.csswg.org/css-break-4/>

[CSS-CASCADE-5]

Elika Etemad; Miriam Suzanne; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 5*. URL: <https://drafts.csswg.org/css-cascade-5/>

[CSS-DISPLAY-4]

Elika Etemad; Tab Atkins Jr.. *CSS Display Module Level 4*. URL: <https://drafts.csswg.org/css-display-4/>

[CSS-FLEXBOX-2]

CSS Flexible Box Layout Module Level 2. Editor's Draft. URL: <https://drafts.csswg.org/css-flexbox-2/>

[CSS-GAPS-1]

Kevin Babbitt. *CSS Gaps Module Level 1*. URL: <https://drafts.csswg.org/css-gaps-1/>

[CSS-GRID-1]

Tab Atkins Jr.; et al. *CSS Grid Layout Module Level 1*. URL: <https://drafts.csswg.org/css-grid-1/>

[CSS-GRID-2]

Tab Atkins Jr.; et al. *CSS Grid Layout Module Level 2*. URL: <https://drafts.csswg.org/css-grid-2/>

[CSS-SIZING-3]

Tab Atkins Jr.; Elika Etemad. *CSS Box Sizing Module Level 3*. URL: <https://drafts.csswg.org/css-sizing-3/>

[CSS-VALUES-3]

Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 3*. URL: <https://drafts.csswg.org/css-values-3/>

[CSS-VALUES-4]

Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 4*. URL: <https://drafts.csswg.org/css-values-4/>

[CSS-WRITING-MODES-4]

Elika Etemad; Koji Ishii. *CSS Writing Modes Level 4*. URL: <https://drafts.csswg.org/css-writing-modes-4/>

[CSS2]

Bert Bos; et al. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. URL: <https://drafts.csswg.org/css2/>

[RFC2119]

S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: <https://datatracker.ietf.org/doc/html/rfc2119>

§ Non-Normative References

[CSS-FLEXBOX-1]

Elika Etemad; Tab Atkins Jr.; Rossen Atanassov. *CSS Flexible Box Layout Module Level 1*. URL: <https://drafts.csswg.org/css-flexbox/>

[CSS-MULTICOL-1]

Florian Rivoal; Rachel Andrew. *CSS Multi-column Layout Module Level 1*. URL: <https://drafts.csswg.org/css-multicol/>

[CSS-POSITION-3]

Elika Etemad; Tab Atkins Jr.. *CSS Positioned Layout Module Level 3*. URL: <https://drafts.csswg.org/css-position-3/>

[CSS-WRITING-MODES-3]

Elika Etemad; Koji Ishii. *CSS Writing Modes Level 3*. URL: <https://drafts.csswg.org/css-writing-modes-3/>

§ Property Index

Name	Value	Initial	Applies to	Inh.	%ages	Anim- ation type	Canonical order	Computed value
<u>‘flow-tolerance’</u>	normal <length- percentage [0,∞]> infinite	normal	grid lanes containers	no	relative to the grid-axis content box size of the grid lanes container	as length	per grammar	a computed <length- percentage> value

§ Issues Index

ISSUE 1 Or should reordering be the default behavior for auto-placed items here?



ISSUE 2 ISSUE(12820): Write up how grid/masonry formatting contexts work more formally?



ISSUE 3 TBD



ISSUE 4 Figure out whether we are re-using ‘[grid-auto-flow](#)’ here (and what its values mean) or defining a new property like ‘[grid-lanes-direction](#)’. [[Issue #12803](#)]



ISSUE 5 Should this work also in Grid Layout somehow, or fall back to a single repetition? If so, how? [[Issue #10915](#)]



ISSUE 6 Technically, if your explicit placement does not enter or cross over the repetition, we can know its placement prior to resolving the repetition. Do we want to adjust the algorithm above to account for this? That is, given ‘[auto repeat\(auto-fill, ...\) auto](#)’, if you explicitly place an item against line 1 or -1, we could let it only affect that track (rather than pretending it’ll go in every track).



ISSUE 7 What if this conflicts with the lanes orientation, or results in both axes stacking?



ISSUE 8 Should this also update the placement cursor?



ISSUE 9 It might be useful to define a static position in the [stacking axis](#). Maybe it could be defined as the max (or min?) current [running position](#) of the [grid-axis](#) tracks at that point? Or the end of the item before it?

