CSS Grid Layout Module Level 1

Editor's Draft, 17 October 2017

▶ **Specification Metadata**

Abstract

This CSS module defines a two-dimensional grid-based layout system, optimized for user interface design. In the grid layout model, the children of a grid container can be positioned into arbitrary slots in a predefined flexible or fixed-size layout grid.

CSS is a language for describing the rendering of structured documents (such as HTML and XML) on screen, on paper, in speech, 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.

GitHub Issues are preferred for discussion of this specification. When filing an issue, please put the text "css-grid" in the title, preferably like this: "[css-grid] *…summary of comment…*". All issues and comments are archived, and there is also a historical archive.

This document was produced by the CSS Working Group (part of the Style Activity).

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 March 2017 W3C Process Document.

The following features are at-risk, and may be dropped during the CR period:

- application of grid placement to absolutely-positioned boxes

"At-risk" is a W3C Process term-of-art, and does not necessarily imply that the feature is in danger of being dropped or delayed. It means that the WG believes the feature may have difficulty being interoperably implemented in a timely manner, and marking it as such allows the WG to drop the feature if necessary when

transitioning to the Proposed Rec stage, without having to publish a new Candidate Rec without the feature first.

Table of Contents

ISSUE 1 If you notice any inconsistencies between this Grid Layout Module and the Flexible Box Layout Module, please report them to the CSSWG, as this is likely an error.

§ 1. Introduction

This section is not normative.

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

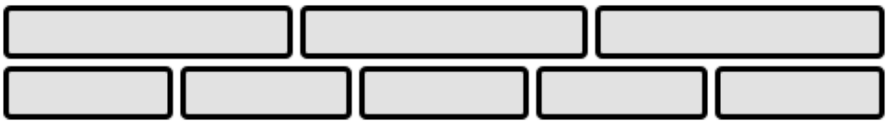


Figure 1 *Exemplary Flex Layout Example*

Figure 2 *Exemplary Grid Layout Example*

In addition, due to its ability to explicitly position items in the grid, Grid Layout allows dramatic transformations in visual layout structure without requiring corresponding markup changes. By combining media queries with the CSS properties that control layout of the grid container and its children, authors can adapt their layout to changes in device form factors, orientation, and available space, while preserving a more ideal semantic structuring of their content across presentations.

Although many layouts can be expressed with either Grid or Flexbox, they each have their specialties. Grid enforces 2-dimensional alignment, uses a top-down approach to layout, allows explicit overlapping of items, and has more powerful spanning capabilities. Flexbox focuses on space distribution within an axis, uses a simpler bottom-up approach to layout, can use a content-size–based line-wrapping system to control its secondary axis, and relies on the underlying markup hierarchy to build more complex layouts. It is expected that both will be valuable and complementary tools for CSS authors.

§ 1.1. Background and Motivation

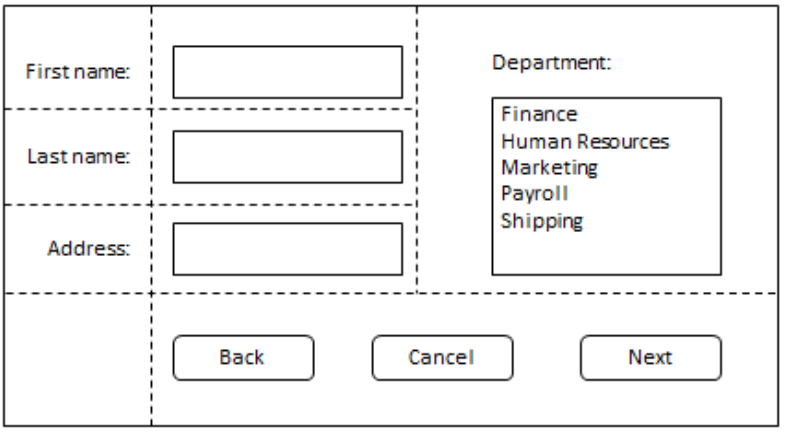


Figure 3 *Application layout example requiring horizontal and vertical alignment.*

As websites evolved from simple documents into complex, interactive applications, techniques for document layout, e.g. floats, were not necessarily well suited for application layout. By using a combination of tables, JavaScript, or careful measurements on floated elements, authors discovered workarounds to achieve desired layouts. Layouts that adapted to the available space were often brittle and resulted in counter-intuitive

behavior as space became constrained. As an alternative, authors of many web applications opted for a fixed layout that cannot take advantage of changes in the available rendering space on a screen.

The capabilities of grid layout address these problems. It provides a mechanism for authors to divide available space for layout into columns and rows using a set of predictable sizing behaviors. Authors can then precisely position and size the building block elements of their application into the grid areas defined by the intersections of these columns and rows. The following examples illustrate the adaptive capabilities of grid layout, and how it allows a cleaner separation of content and style.

§ 1.1.1. Adapting Layouts to Available Space

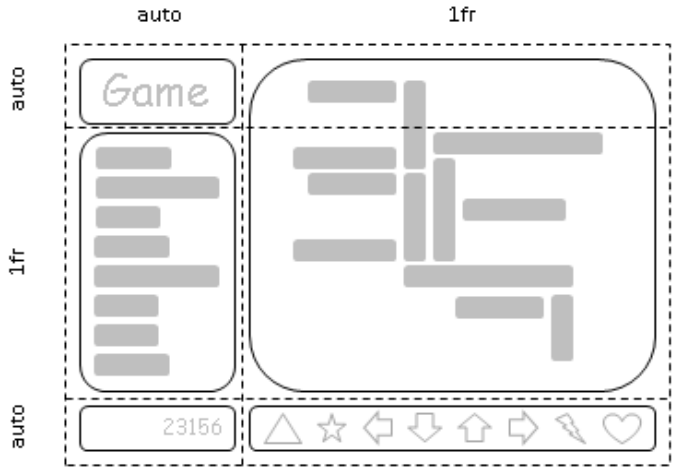


Figure 4 *Five grid items arranged according to content size and available space.*

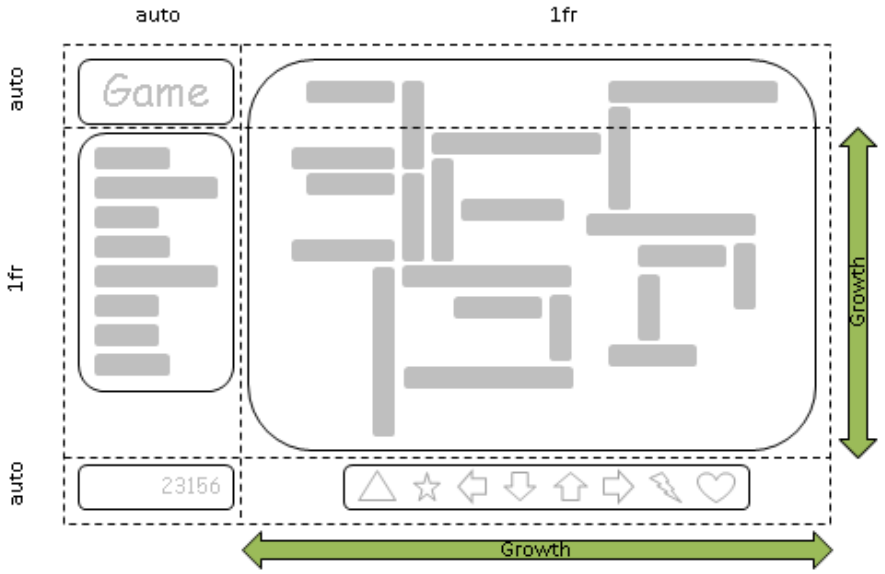


Figure 5 *Growth in the grid due to an increase in available space.*

Grid layout can be used to intelligently resize elements within a webpage. The adjacent figures represent a game with five major components in the layout: the game title, stats area, game board, score area, and control area. The author's intent is to divide the space for the game such that:

- The stats area always appears immediately under the game title.
- The game board appears to the right of the stats and title.
- The top of the game title and the game board should always align.
- The bottom of the game board and bottom of the stats area align when the game has reached its minimum height. In all other cases the game board will stretch to take advantage of all the space available to it.
- The controls are centered under the game board.
- The top of the score area is aligned to the top of the controls area.
- The score area is beneath the stats area.
- The score area is aligned to the controls beneath the stats area.

The following grid layout example shows how an author might achieve all the sizing, placement, and alignment rules declaratively.

EXAMPLE 1

```
/**
 * Define the space for each grid item by declaring the grid
 * on the grid container.
 */
#grid {
  /**
   * Two columns:
   *  1. the first sized to content,
   *  2. the second receives the remaining space
   *     (but is never smaller than the minimum size of the board
   *     or the game controls, which occupy this column [Figure 4])
   *
   * Three rows:
   *  3. the first sized to content,
   *  4. the middle row receives the remaining space
   *     (but is never smaller than the minimum height
   *      of the board or stats areas)
   *  5. the last sized to content.
   */
  display: grid;
  grid-template-columns:
    /* 1 */ auto
    /* 2 */ 1fr;
  grid-template-rows:
    /* 3 */ auto
    /* 4 */ 1fr
    /* 5 */ auto;
}

/* Specify the position of each grid item using coordinates on
 * the 'grid-row' and 'grid-column' properties of each grid item.
 */
#title    { grid-column: 1; grid-row: 1; }
#score    { grid-column: 1; grid-row: 3; }
#stats    { grid-column: 1; grid-row: 2; align-self: start; }
#board    { grid-column: 2; grid-row: 1 / span 2; }
#controls { grid-column: 2; grid-row: 3; justify-self: center; }


<div id="grid">
  <div id="title">Game Title</div>
  <div id="score">Score</div>
  <div id="stats">Stats</div>
  <div id="board">Board</div>
  <div id="controls">Controls</div>
</div>
```

Note: There are multiple ways to specify the structure of the grid and to position and size grid items, each optimized for different scenarios.

§ 1.1.2. Source-Order Independence

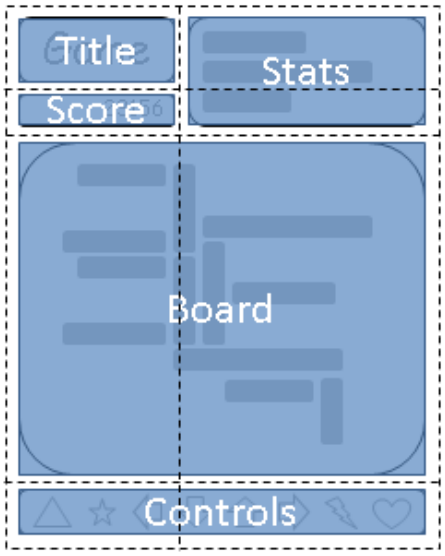


Figure 6 *An arrangement suitable for "portrait" orientation.*

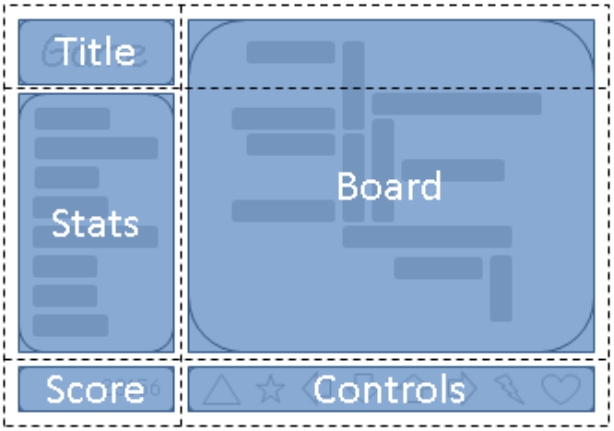


Figure 7 *An arrangement suitable for "landscape" orientation.*

Continuing the prior example, the author also wants the game to adapt to different devices. Also, the game should optimize the placement of the components when viewed either in portrait or landscape orientation (Figures 6 and 7). By combining grid layout with media queries, the author is able to use the same semantic markup, but rearrange the layout of elements independent of their source order, to achieve the desired layout in both orientations.

The following example uses grid layout's ability to name the space which will be occupied by a grid item. This allows the author to avoid rewriting rules for grid items as the grid's definition changes.

EXAMPLE 2

```
@media (orientation: portrait) {
  #grid {
    display: grid;

    /* The rows, columns and areas of the grid are defined visually
     * using the grid-template-areas property.  Each string is a row,
     * and each word an area.  The number of words in a string
     * determines the number of columns. Note the number of words
     * in each string must be identical. */
    grid-template-areas: "title stats"
                         "score stats"
                         "board board"
                         "ctrls ctrls";

    /* The way to size columns and rows can be assigned with the
     * grid-template-columns and grid-template-rows properties. */
    grid-template-columns: auto 1fr;
    grid-template-rows: auto auto 1fr auto;
  }
}

@media (orientation: landscape) {
  #grid {
    display: grid;

    /* Again the template property defines areas of the same name,
     * but this time positioned differently to better suit a
     * landscape orientation. */
    grid-template-areas: "title board"
                         "stats board"
                         "score ctrls";

    grid-template-columns: auto 1fr;
    grid-template-rows: auto 1fr auto;
  }
}

/* The grid-area property places a grid item into a named
 * area of the grid. */
#title    { grid-area: title }
#score    { grid-area: score }
#stats    { grid-area: stats }
#board    { grid-area: board }
#controls { grid-area: ctrls }
```

```
<div id="grid">
  <div id="title">Game Title</div>
  <div id="score">Score</div>
  <div id="stats">Stats</div>
  <div id="board">Board</div>
  <div id="controls">Controls</div>
</div>
```

Note: The reordering capabilities of grid layout intentionally affect *only the visual rendering*, leaving speech order and navigation based on the source order. This allows authors to manipulate the visual presentation while leaving the source order intact and optimized for non-CSS UAs and for linear models such as speech and sequential navigation.

Grid item placement and reordering must not be used as a substitute for correct source ordering, as that can ruin the accessibility of the document.

§ 2. Overview

This section is not normative.

Grid Layout controls the layout of its content through the use of a grid: an intersecting set of horizontal and vertical lines which create a sizing and positioning coordinate system for the grid container's contents. Grid Layout features

- fixed, flexible, and content-based track sizing functions
- explicit item placement via forwards (positive) and backwards (negative) numerical grid coordinates, named grid lines, and named grid areas; automatic item placement into empty areas, including reordering with 'order'
- space-sensitive track repetition and automatic addition of rows or columns to accommodate additional content
- control over alignment and spacing with margins, gutters, and the alignment properties
- the ability to overlap content and control layering with 'z-index'

Grid containers can be nested or mixed with flex containers as necessary to create more complex layouts.

§ 2.1. Declaring the Grid

The tracks (rows and columns) of the grid are declared and sized either explicitly through the explicit grid properties or are implicitly created when items are placed outside the explicit grid. The 'grid' shorthand and its sub-properties define the parameters of the grid. §7 Defining the Grid

EXAMPLE 3

Below are some examples of grid declarations:

- The following declares a grid with four named areas: `H`, `A`, `B`, and `F`. The first column is sized to fit its contents (‘auto’), and the second column takes up the remaining space (‘1fr’). Rows default to ‘auto’ (content-based) sizing; the last row is given a fixed size of ‘30px’.

```
main {
  grid: "H    H "
        "A    B "
        "F    F " 30px
  /     auto 1fr;
}
```

- The following declares a grid with as many rows of at least ‘5em’ as will fit in the height of the grid container (‘100vh’). The grid has no explicit columns; instead columns are added as content is added, the resulting column widths are equalized (‘1fr’). Since content overflowing to the right won’t print, an alternate layout for printing adds rows instead.

```
main {
  grid: repeat(auto-fill, 5em) / auto-flow 1fr;
  height: 100vh;
}
@media print {
  main {
    grid: auto-flow 1fr / repeat(auto-fill, 5em);
  }
}
```

- The following declares a grid with 5 evenly-sized columns and three rows, with the middle row taking up all remaining space (and at least enough to fit its contents).

```
main {
  grid: auto 1fr auto / repeat(5, 1fr);
  min-height: 100vh;
}
```

§ 2.2. Placing Items

The contents of the grid container are organized into individual grid items (analogous to flex items), which are then assigned to predefined areas in the grid. They can be explicitly placed using coordinates through the grid-placement properties or implicitly placed into empty areas using auto-placement. §8 Placing Grid Items

EXAMPLE 4

Below are some examples of grid placement declarations using the ‘grid-area’ shorthand:

```
grid-area: a;          /* Place into named grid area “a”     */
grid-area: auto;       /* Auto-place into next empty area    */
grid-area: 2 / 4;      /* Place into row 2, column 4         */
grid-area: 1 / 3 / -1; /* Place into column 3, span all rows */
grid-area: header-start / sidebar-start / footer-end / sidebar-start;
                       /* Place using named lines            */
```

These are equivalent to the following ‘grid-row’ + ‘grid-column’ declarations:

```
grid-row: a;                          grid-column: a;
grid-row: auto;                       grid-column: auto;
grid-row: 2;                          grid-column: 4;
grid-row: 1 / -1;                     grid-column: 3;
grid-row: header-start / footer-end; grid-column: sidebar-start / footer-end;
```

They can further be decomposed into the ‘grid-row-start’/‘grid-row-end’/‘grid-column-start’/‘grid-column-end’ longhands, e.g.

```
grid-area: a;
/* Equivalent to grid-row-start: a; grid-column-start: a; grid-row-end: a; grid-columr

grid-area: 1 / 3 / -1;
/* Equivalent to grid-row-start: 1; grid-column-start: 3; grid-row-end: -1; grid-colur
```

§ 2.3. Sizing the Grid

Once the grid items have been placed, the sizes of the grid tracks (rows and columns) are calculated, accounting for the sizes of their contents and/or available space as specified in the grid definition.

The resulting sized grid is aligned within the grid container according to the grid container’s ‘align-content’ and ‘justify-content’ properties. §10 Alignment and Spacing

EXAMPLE 5

The following example justifies all columns by distributing any extra space among them, and centers the grid in the grid container when it is smaller than 100vh.

```
main {
  grid: auto-flow 1fr / repeat(auto-fill, 5em);
  min-height: 100vh;
  justify-content: space-between;
  align-content: safe center;
}
```

Finally each grid item is sized and aligned within its assigned grid area, as specified by its own sizing [CSS21] and alignment properties [CSS-ALIGN-3].

§ 3. Grid Layout Concepts and Terminology

In ***grid layout***, the content of a grid container is laid out by positioning and aligning it into a grid. The ***grid*** is an intersecting set of horizontal and vertical grid lines that divides the grid container's space into grid areas, into which grid items (representing the grid container's content) can be placed. There are two sets of grid lines: one set defining ***columns*** that run along the block axis (the ***column axis***), and an orthogonal set defining ***rows*** along the inline axis (the ***row axis***). [CSS3-WRITING-MODES]

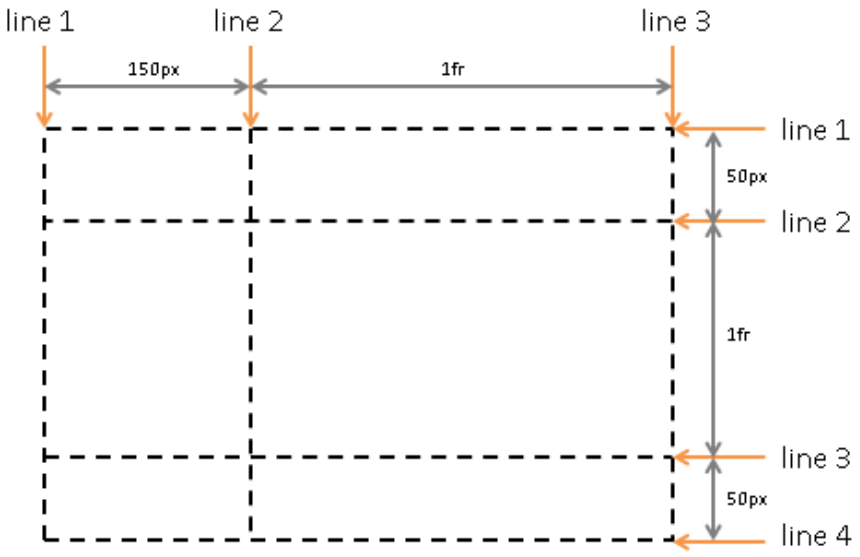


Figure 8 *Grid lines: Three in the block axis and four in the inline axis.*

> Note: The "column axis" term as used here was chosen to match the block axis because this matches the visual orientation of the column boxes themselves and matches the direction that a column grows as more items are added to it. As a result, it does not match the axis in which new columns are added or in which their track size is measured, which is the inline or row axis The opposite logic, of course, applies to the "row axis".

§ 3.1. Grid Lines

Grid lines are the horizontal and vertical dividing lines of the grid. A grid line exists on either side of a column or row. They can be referred to by numerical index, or by an author-specified name. A grid item references the grid lines to determine its position within the grid using the grid-placement properties.

EXAMPLE 6

The following two examples both create three column grid lines and four row grid lines.

This first example demonstrates how an author would position a grid item using grid line numbers:

```
#grid {
  display: grid;
  grid-template-columns: 150px 1fr;
  grid-template-rows: 50px 1fr 50px;
}

#item1 { grid-column: 2;
         grid-row-start: 1; grid-row-end: 4; }
```

This second example uses explicitly named grid lines:

```
/* equivalent layout to the prior example, but using named lines */
#grid {
  display: grid;
  grid-template-columns: 150px [item1-start] 1fr [item1-end];
  grid-template-rows: [item1-start] 50px 1fr 50px [item1-end];
}

#item1 {
  grid-column: item1-start / item1-end;
  grid-row: item1-start / item1-end;
}
```

§ 3.2. Grid Tracks and Cells

Grid track is a generic term for a grid column or grid row—in other words, it is the space between two adjacent grid lines. Each grid track is assigned a sizing function, which controls how wide or tall the column or row may grow, and thus how far apart its bounding grid lines are. Adjacent grid tracks can be separated by gutters but are otherwise packed tightly.

A ***grid cell*** is the intersection of a grid row and a grid column. It is the smallest unit of the grid that can be referenced when positioning grid items.

EXAMPLE 7

In the following example there are two columns and three rows. The first column is fixed at 150px. The second column uses flexible sizing, which is a function of the unassigned space in the grid, and thus will vary as the width of the grid container changes. If the used width of the grid container is 200px, then the second column is 50px wide. If the used width of the grid container is 100px, then the second column is 0px and any content positioned in the column will overflow the grid container.

```
#grid {
  display: grid;
  grid-template-columns: 150px 1fr;  /* two columns */
  grid-template-rows: 50px 1fr 50px; /* three rows  */
}
```

§ 3.3. Grid Areas

A ***grid area*** is the logical space used to lay out one or more grid items. A grid area consists of one or more adjacent grid cells. It is bound by four grid lines, one on each side of the grid area, and participates in the sizing of the grid tracks it intersects. A grid area can be named explicitly using the 'grid-template-areas' property of the grid container, or referenced implicitly by its bounding grid lines. A grid item is assigned to a grid area using the grid-placement properties.

EXAMPLE 8

```
/* using the template syntax */
#grid  {
display: grid;
grid-template-areas: ". a"
                     "b a"
                     ". a";
grid-template-columns: 150px 1fr;
grid-template-rows: 50px 1fr 50px;
}

#item1 { grid-area: a }
#item2 { grid-area: b }
#item3 { grid-area: b }

/* Align items 2 and 3 at different points in the grid area "b".  */
/* By default, grid items are stretched to fit their grid area    */
/* and these items would layer one over the other. */
#item2 { align-self: start; }
#item3 { justify-self: end; align-self: end; }
```

A grid item's grid area forms the containing block into which it is laid out. Grid items placed into the same grid area do not directly affect each other's layout. Indirectly, however, a grid item occupying a grid track

with an intrinsic sizing function can affect the size of that track (and thus the positions of its bounding grid lines), which in turn can affect the position or size of another grid item.

§ 4. Reordering and Accessibility

Grid layout gives authors great powers of rearrangement over the document. However, these are not a substitute for correct ordering of the document source. The 'order' property and grid placement *do not* affect ordering in non-visual media (such as speech). Likewise, rearranging grid items visually does not affect the default traversal order of sequential navigation modes (such as cycling through links, see e.g. `tabindex` [HTML5]).

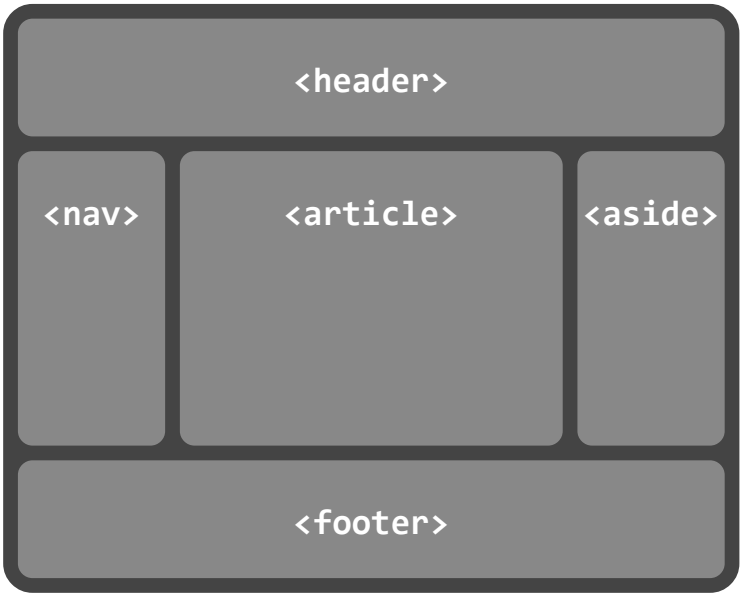
Authors *must* use 'order' and the grid-placement properties only for visual, not logical, reordering of content. Style sheets that use these features to perform logical reordering are non-conforming.

Note: This is so that non-visual media and non-CSS UAs, which typically present content linearly, can rely on a logical source order, while grid layout's placement and ordering features are used to tailor the visual arrangement. (Since visual perception is two-dimensional and non-linear, the desired visual order is not always equivalent to the desired reading order.)

EXAMPLE 9

Many web pages have a similar shape in the markup, with a header on top, a footer on bottom, and then a content area and one or two additional columns in the middle. Generally, it's desirable that the content come first in the page's source code, before the additional columns. However, this makes many common designs, such as simply having the additional columns on the left and the content area on the right, difficult to achieve. This has been addressed in many ways over the years, often going by the name "Holy Grail Layout" when there are two additional columns. Grid Layout makes this example trivial. For example, take the following sketch of a page's code and desired layout:

```
<!DOCTYPE html>
<header>...</header>
<article>...</article>
<nav>...</nav>
<aside>...</aside>
<footer>...</footer>
```



This layout can be easily achieved with grid layout:

```
main { display: grid;
       grid: "h h h"
             "a b c"
             "f f f";
       grid-template-columns: auto 1fr 20%; }
article { grid-area: b; min-width: 12em;     }
nav     { grid-area: a; /* auto min-width */ }
aside   { grid-area: c; min-width: 12em;     }
```

As an added bonus, the columns will all be 'equal-height' by default, and the main content will be as wide as necessary to fill the screen. Additionally, this can then be combined with media queries to switch to an all-vertical layout on narrow screens:

```
@media all and (max-width: 60em) {
  /* Too narrow to support three columns */
  main { display: block; }
}
```

In order to preserve the author's intended ordering in all presentation modes, authoring tools—including WYSIWYG editors as well as Web-based authoring aids—must reorder the underlying document source and not use 'order' or grid-placement properties to perform reordering unless the author has explicitly indicated that the underlying document order (which determines speech and navigation order) should be *out-of-sync* with the visual order.

EXAMPLE 10

For example, a tool might offer both drag-and-drop arrangement of grid items as well as handling of media queries for alternate layouts per screen size range.

Since most of the time, reordering should affect all screen ranges as well as navigation and speech order, the tool would match the resulting drag-and-drop visual arrangement by simultaneously reordering the DOM layer. In some cases, however, the author may want different visual arrangements per screen size. The tool could offer this functionality by using the grid-placement properties together with media queries, but also tie the smallest screen size's arrangement to the underlying DOM order (since this is most likely to be a logical linear presentation order) while using grid-placement properties to rearrange the visual presentation in other size ranges.

This tool would be conformant, whereas a tool that only ever used the grid-placement properties to handle drag-and-drop grid rearrangement (however convenient it might be to implement it that way) would be non-conformant.

§ 5. Grid Containers

§ 5.1. Establishing Grid Containers: the 'grid' and 'inline-grid' 'display' values

Name:	'display'
New values:	grid | inline-grid

'grid'
: This value causes an element to generate a block-level grid container box.

'inline-grid'
: This value causes an element to generate an inline-level grid container box.

A ***grid container*** establishes a new ***grid formatting context*** for its contents. This is the same as establishing a block formatting context, except that grid layout is used instead of block layout: floats do not intrude into the grid container, and the grid container's margins do not collapse with the margins of its contents. The

contents of a grid container are laid out into a grid, with grid lines forming the boundaries of each grid items containing block. The 'overflow' property applies to grid containers.

Grid containers are not block containers, and so some properties that were designed with the assumption of block layout don't apply in the context of grid layout. In particular:

- 'float' and 'clear' have no effect on a grid item. However, the 'float' property still affects the computed value of 'display' on children of a grid container, as this occurs *before* grid items are determined.
- 'vertical-align' has no effect on a grid item.
- the '::first-line' and '::first-letter' pseudo-elements do not apply to grid containers, and grid containers do not contribute a first formatted line or first letter to their ancestors.

If an element's specified 'display' is 'inline-grid' and the element is floated or absolutely positioned, the computed value of 'display' is 'grid'. The table in CSS 2.1 Chapter 9.7 is thus amended to contain an additional row, with 'inline-grid' in the "Specified Value" column and 'grid' in the "Computed Value" column.

§ 5.2. Sizing Grid Containers

Note see [CSS3-SIZING] for a definition of the terms in this section.

A grid container is sized using the rules of the formatting context in which it participates:

- As a block-level box in a block formatting context, it is sized like a block box that establishes a formatting context, with an 'auto' inline size calculated as for non-replaced block boxes.
- As an inline-level box in an inline formatting context, it is sized as an atomic inline-level box (such as an inline-block).

In both inline and block formatting contexts, the grid container's 'auto' block size is its max-content size.

The block layout spec should probably define this, but it isn't written yet.

The max-content size (min-content size) of a grid container is the sum of the grid container's track sizes (including gutters) in the appropriate axis, when the grid is sized under a max-content constraint (min-content constraint).

§ 5.3. Clamping Overly Large Grids

Since memory is limited, UAs may clamp the possible size of the grid to be within a UA-defined limit, dropping all lines outside that limit. If a grid item is placed outside this limit, its grid area must be clamped to within this limited grid.

To ***clamp a grid area***:

- If the grid area would span outside the limited grid, its span is clamped to the last line of the limited grid.

- If the grid area would be placed completely outside the limited grid, its span must be truncated to 1 and the area repositioned into the last grid track on that side of the grid.

EXAMPLE 11

For example, if a UA only supported grids with at most 1000 tracks in each dimension, the following placement properties:

```
.grid-item {
  grid-row: 500 / 1500;
  grid-column: 2000 / 3000;
}
```

Would end up being equivalent to:

```
.grid-item {
  grid-row: 500 / 1001;
  grid-column: 1000 / 1001;
}
```

§ 6. Grid Items

Loosely speaking, the ***grid items*** of a grid container are boxes representing its in-flow contents.

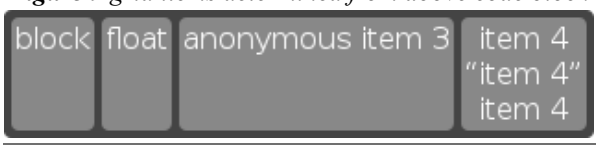
Each in-flow child of a grid container becomes a grid item, and each contiguous sequence of child text runs is wrapped in an anonymous block container grid item. However, if the entire sequence of child text runs contains only white space (i.e. characters that can be affected by the 'white-space' property) it is instead not rendered (just as if its text nodes were 'display:none').

EXAMPLE 12

Examples of grid items:

```
<div style="display: grid">

    <!-- grid item: block child -->
    <div id="item1">block</div>

    <!-- grid item: floated element; floating is ignored -->
    <div id="item2" style="float: left;">float</div>

    <!-- grid item: anonymous block box around inline content -->
    anonymous item 3

    <!-- grid item: inline child -->
    <span>
        item 4
        <!-- grid items do not split around blocks -->
        <q style="display: block" id=not-an-item>item 4</q>
        item 4
    </span>
</div>
```

Figure 9 *grid items determined from above code block*



Note: inter-element white space disappears: it does not become its own grid item, even though inter-element text *does* get wrapped in an anonymous grid item.

Note: The box of a anonymous item is unstyleable, since there is no element to assign style rules to. Its contents will however inherit styles (such as font settings) from the grid container.

§ 6.1. Grid Item Display

A grid item establishes a new formatting context for its contents. The type of this formatting context is determined by its ‘display’ value, as usual. However, grid items are ***grid-level*** boxes, not block-level boxes: they participate in their container’s grid formatting context, not in a block formatting context.

The ‘display’ value of a grid item is blockified: if the specified ‘display’ of an in-flow child of an element generating a grid container is an inline-level value, it computes to its block-level equivalent. (See

CSS2.1§9.7 [CSS21] and CSS Display [CSS3-DISPLAY] for details on this type of 'display' value conversion.)

> Note: Some values of 'display' normally trigger the creation of anonymous boxes around the original box. If such a box is a grid item, it is blockified first, and so anonymous box creation will not happen. For example, two contiguous grid items with 'display: table-cell' will become two separate 'display: block' grid items, instead of being wrapped into a single anonymous table.

§ 6.2. Grid Item Sizing

A grid item is sized within the containing block defined by its grid area.

Grid item calculations for 'auto' widths and heights vary by their self-alignment values:

'normal'
: If the grid item is either non-replaced—or is replaced but has no intrinsic aspect ratio and no intrinsic size in the relevant dimension—use the width calculation rules for non-replaced boxes as defined in CSS2.1 § 10.3.3.

  Otherwise, if the grid item has an intrinsic ratio or an intrinsic size in the relevant dimension, the grid item is sized as for 'align-self: start' (consistent with the width calculation rules for block-level replaced elements in CSS2.1 § 10.3.4).

'stretch'
: Use the width calculation rules for non-replaced boxes, as defined in CSS2.1 § 10.3.3.

  > Note: This may distort the aspect ratio of the item, if it has one.

all other values
: Size the item as 'fit-content'.

The following informative table summarizes the automatic sizing of grid items:

Summary of automatic sizing behavior of grid items

Alignment	Non-replaced Element Size	Replaced Element Size
'normal'	Fill grid area	Use intrinsic size
'stretch'	Fill grid area	Fill grid area
'start'/'center'/etc.	'fit-content' sizing (like floats)	Use intrinsic size

> Note: The 'auto' value of 'min-width' and 'min-height' affects track sizing in the relevant axis similar to how it affects the main size of a flex item. See §6.6 Automatic Minimum Size of Grid Items.

§ 6.3. Reordered Grid Items: the 'order' property

The 'order' property also applies to grid items. It affects their auto-placement and painting order.

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. See CSS Flexbox 1 §5.4.1 Reordering and Accessibility in [CSS-FLEXBOX-1].

§ 6.4. Grid Item Margins and Paddings

As adjacent grid items are independently contained within the containing block formed by their grid areas, the margins of adjacent grid items do not collapse.

Percentage margins and paddings on grid items can be resolved against either:

- their own axis (left/right percentages resolve against width, top/bottom resolve against height), or,
- the inline axis (left/right/top/bottom percentages all resolve against width)

A User Agent must choose one of these two behaviors.

Note: This variance sucks, but it accurately captures the current state of the world (no consensus among implementations, and no consensus within the CSSWG). It is the CSSWG's intention that browsers will converge on one of the behaviors, at which time the spec will be amended to require that.

Authors should avoid using percentages in paddings or margins on grid items entirely, as they will get different behavior in different browsers.

Auto margins expand to absorb extra space in the corresponding dimension, and can therefore be used for alignment.

See §10.2 Aligning with auto margins

§ 6.5. Z-axis Ordering: the 'z-index' property

Grid items can overlap when they are positioned into intersecting grid areas, or even when positioned in non-intersecting areas because of negative margins or positioning. The painting order of grid items is exactly the same as inline blocks [CSS21], except that order-modified document order is used in place of raw document order, and 'z-index' values other than 'auto' create a stacking context even if 'position' is 'static' (behaving exactly as if 'position' were 'relative'). Thus the 'z-index' property can easily be used to control the z-axis order of grid items.

Note: Descendants that are positioned outside a grid item still participate in any stacking context established by the grid item.

EXAMPLE 13

The following diagram shows several overlapping grid items, with a combination of implicit source order and explicit 'z-index' used to control their stacking order.

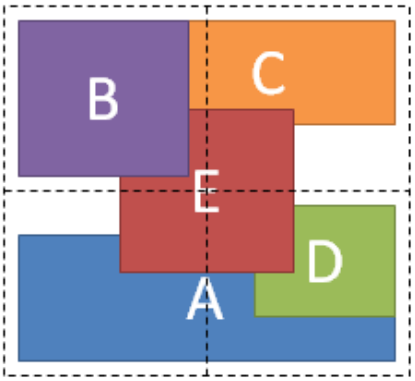


Figure 10 *Drawing order controlled by z-index and source order.*

```
<style type="text/css">
#grid {
  display: grid;
  grid-template-columns: 1fr 1fr;
  grid-template-rows: 1fr 1fr
}
#A { grid-column: 1 / span 2; grid-row: 2; align-self: end; }
#B { grid-column: 1; grid-row: 1; z-index: 10; }
#C { grid-column: 2; grid-row: 1; align-self: start; margin-left: -20px; }
#D { grid-column: 2; grid-row: 2; justify-self: end; align-self: start; }
#E { grid-column: 1 / span 2; grid-row: 1 / span 2;
     z-index: 5; justify-self: center; align-self: center; }
</style>

<div id="grid">
  <div id="A">A</div>
  <div id="B">B</div>
  <div id="C">C</div>
  <div id="D">D</div>
  <div id="E">E</div>
</div>
```

§ 6.6. Automatic Minimum Size of Grid Items

To provide a more reasonable default minimum size for grid items, this specification defines that the 'auto' value of 'min-width'/'min-height' also applies an automatic minimum size in the specified axis to grid items whose 'overflow' is 'visible' and which span at least one track whose min track sizing function is 'auto'.

The automatic minimum size for a grid item in a given dimension is its specified size if it exists, otherwise its transferred size if that exists, else its content size, each as defined in [CSS-FLEXBOX-1]. However, if the

grid item spans only grid tracks that have a fixed max track sizing function, its specified size and content size in that dimension (and the input to the transferred size in the other dimension) are further clamped to less than or equal to the stretch fit the grid area's size (so as to prevent the automatic minimum size from forcing overflow of its fixed-size grid area).

> Note that while a content-based minimum size is often appropriate, and helps prevent content from overlapping or spilling outside its container, in some cases it is not:
>
> In particular, if grid layout is being used for a major content area of a document, it is better to set an explicit font-relative minimum width such as 'min-width: 12em'. A content-based minimum width could result in a large table or large image stretching the size of the entire content area, potentially into an overflow zone, and thereby making lines of text needlessly long and hard to read.
>
> Note also, when content-based sizing is used on an item with large amounts of content, the layout engine must traverse all of this content before finding its minimum size, whereas if the author sets an explicit minimum, this is not necessary. (For items with small amounts of content, however, this traversal is trivial and therefore not a performance concern.)

§ 7. Defining the Grid

§ 7.1. The Explicit Grid

The three properties 'grid-template-rows', 'grid-template-columns', and 'grid-template-areas' together define the ***explicit grid*** of a grid container. The final grid may end up larger due to grid items placed outside the explicit grid; in this case implicit tracks will be created, these implicit tracks will be sized by the 'grid-auto-rows' and 'grid-auto-columns' properties.

The size of the explicit grid is determined by the larger of the number of rows/columns defined by 'grid-template-areas' and the number of rows/columns sized by 'grid-template-rows'/'grid-template-columns'. Any rows/columns defined by 'grid-template-areas' but not sized by 'grid-template-rows'/'grid-template-columns' take their size from the 'grid-auto-rows'/'grid-auto-columns' properties. If these properties don't define *any* explicit tracks the explicit grid still contains one grid line in each axis.

Numeric indexes in the grid-placement properties count from the edges of the explicit grid. Positive indexes count from the start side (starting from 1 for the start-most explicit line), while negative indexes count from the end side (starting from -1 for the end-most explicit line).

The 'grid' and 'grid-template' properties are a shorthands that can be used to set all three ***explicit grid properties*** ('grid-template-rows', 'grid-template-columns', and 'grid-template-areas') at the same time. The 'grid' shorthand also resets properties controlling the implicit grid, whereas the 'grid-template' property leaves them unchanged.

§ 7.2. Explicit Track Sizing: the 'grid-template-rows' and 'grid-template-columns' properties

Name:	***'grid-template-columns'***, ***'grid-template-rows'***
Value:	none | <track-list> | <auto-track-list>
Initial:	none
Applies to:	grid containers
Inherited:	no
Percentages:	refer to corresponding dimension of the content area
Media:	visual
Computed value:	As specified, with lengths made absolute
Canonical order:	per grammar
Animatable:	as a simple list of length, percentage, or calc, provided the only differences are the values of the length, percentage, or calc components in the list

These properties specify, as a space-separated ***track list***, the line names and track sizing functions of the grid. The 'grid-template-columns' property specifies the track list for the grid's columns, while 'grid-template-rows' specifies the track list for the grid's rows.

Values have the following meanings:

'none'
: Indicates that no explicit grid tracks are created by this property (though explicit grid tracks could still be created by 'grid-template-areas').

> Note: In the absence of an explicit grid any rows/columns will be implicitly generated, and their size will be determined by the 'grid-auto-rows' and 'grid-auto-columns' properties.

'<track-list> | <auto-track-list>'
: Specifies the track list as a series of track sizing functions and line names. Each ***track sizing function*** can be specified as a length, a percentage of the grid container's size, a measurement of the contents occupying the column or row, or a fraction of the free space in the grid. It can also be specified as a range using the 'minmax()' notation, which can combine any of the previously mentioned mechanisms to specify separate min and max track sizing functions for the column or row.

The syntax of a track list is:

```
<track-list>          = [ <line-names>? [ <track-size> | <track-repeat> ] ]+ <line-name
<auto-track-list>     = [ <line-names>? [ <fixed-size> | <fixed-repeat> ] ]* <line-names
                        [ <line-names>? [ <fixed-size> | <fixed-repeat> ] ]* <line-names
<explicit-track-list> = [ <line-names>? <track-size> ]+ <line-names>?

<track-size>          = <track-breadth> | minmax( <inflexible-breadth> , <track-breadth>
<fixed-size>          = <fixed-breadth> | minmax( <fixed-breadth> , <track-breadth> ) | m
<track-breadth>       = <length-percentage> | <flex> | min-content | max-content | auto
<inflexible-breadth>  = <length-percentage> | min-content | max-content | auto
<fixed-breadth>       = <length-percentage>
<line-names>          = '[' <custom-ident>* ']'
```

Where:

'<length-percentage>'

A non-negative length or percentage, as defined by CSS3 Values. [CSS3VAL]

<percentage> values are relative to the inline size of the grid container in column grid tracks, and the block size of the grid container in row grid tracks. If the size of the grid container depends on the size of its tracks, then the <percentage> must be treated as 'auto', for the purpose of calculating the intrinsic sizes of the grid container and then resolve against that resulting grid container size for the purpose of laying out the grid and its items.

'<flex>'

A non-negative dimension with the unit 'fr' specifying the track's ***flex factor***. Each <flex>-sized track takes a share of the remaining space in proportion to its flex factor. See Flexible Lengths for more details.

When appearing outside a 'minmax()' notation, implies an automatic minimum (i.e. "minmax(auto, <flex>)").

'max-content'

Represents the largest max-content contribution of the grid items occupying the grid track.

'min-content'

Represents the largest min-content contribution of the grid items occupying the grid track.

'minmax(*min, max*)'

Defines a size range greater than or equal to *min* and less than or equal to *max*. If *max* < *min*, then *max* is ignored and 'minmax(min,max)' is treated as *min*. As a maximum, a <flex> value sets the track's flex factor; it is invalid as a minimum.

> Note: A future level of this spec may allow <flex> minimums, and will update the track sizing algorithm to account for this correctly

'auto'

As a maximum, identical to 'max-content'. As a minimum, represents the largest minimum size (as specified by 'min-width'/'min-height') of the grid items occupying the grid track.

Note: ‘auto’ track sizes (and only ‘auto’ track sizes) can be stretched by the ‘align-content’ and ‘justify-content’ properties.

‘fit-content(<length-percentage>)’

Represents the formula `min(‘max-content’, max(‘auto’, argument))`, which is calculated like ‘minmax(auto, max-content)’, except that the track size is clamped at *argument* if it is greater than the ‘auto’ minimum.

EXAMPLE 14

Given the following ‘grid-template-columns’ declaration:

```
grid-template-columns: 100px 1fr max-content minmax(min-content, 1fr);
```

Five grid lines are created:

1. At the start edge of the grid container.
2. 100px from the start edge of the grid container.
3. A distance from the previous line equal to half the free space (the width of the grid container, minus the width of the non-flexible grid tracks).
4. A distance from the previous line equal to the maximum size of any grid items belonging to the column between these two lines.
5. A distance from the previous line at least as large as the largest minimum size of any grid items belonging to the column between these two lines, but no larger than the other half of the free space.

If the non-flexible sizes (‘100px’, ‘max-content’, and ‘min-content’) sum to larger than the grid container’s width, the final grid line will be a distance equal to their sum away from the start edge of the grid container (the ‘1fr’ sizes both resolve to ‘0’). If the sum is less than the grid container’s width, the final grid line will be exactly at the end edge of the grid container. This is true in general whenever there’s at least one <flex> value among the grid track sizes.

EXAMPLE 15

Additional examples of valid grid track definitions:

```
/* examples of valid track definitions */
grid-template-rows: 1fr minmax(min-content, 1fr);
grid-template-rows: 10px repeat(2, 1fr auto minmax(30%, 1fr));
grid-template-rows: calc(4em - 5px);
```

Note: The size of the grid is not purely the sum of the track sizes, as ‘row-gap’, ‘column-gap’ and ‘justify-content’, ‘align-content’ can add additional space between tracks.

§ 7.2.1. Named Grid Lines: the ‘[<custom-ident>*]’ syntax

While grid lines can always be referred to by their numerical index, ***named lines*** can make the grid-placement properties easier to understand and maintain. Lines can be explicitly named in the 'grid-template-rows' and 'grid-template-columns' properties, or implicitly named by creating named grid areas with the 'grid-template-areas' property.

EXAMPLE 16

For example, the following code gives meaningful names to all of the lines in the grid. Note that some of the lines have multiple names.

```
#grid {
  display: grid;
  grid-template-columns: [first nav-start] 150px [main-start] 1fr [last];
  grid-template-rows: [first header-start] 50px [main-start] 1fr [footer-start] 50px |
}
```

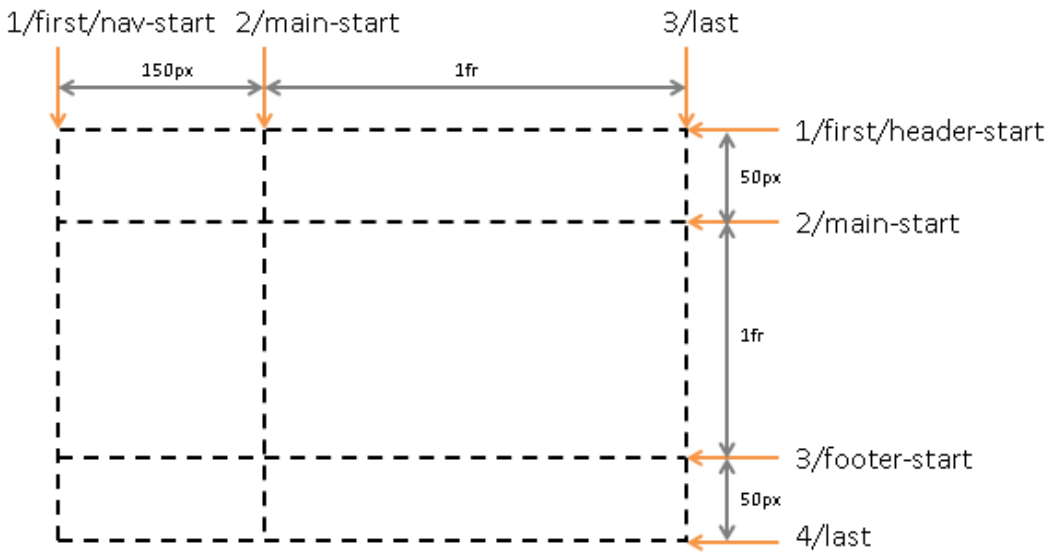


Figure 11 *Named Grid Lines.*

A line name cannot be 'span', i.e. the <custom-ident> in the <line-names> production excludes the keyword 'span'.

§ 7.2.2. Repeating Rows and Columns: the 'repeat()' notation

The ***'repeat()'*** notation represents a repeated fragment of the track list, allowing a large number of columns or rows that exhibit a recurring pattern to be written in a more compact form.

EXAMPLE 17

This example shows two equivalent ways of writing the same grid definition. Both declarations produce four “main” columns, each 250px wide, surrounded by 10px “gutter” columns.

```
grid-template-columns: 10px [col-start] 250px [col-end]
                       10px [col-start] 250px [col-end]
                       10px [col-start] 250px [col-end]
                       10px [col-start] 250px [col-end] 10px;
/* same as above, except easier to write */
grid-template-columns: repeat(4, 10px [col-start] 250px [col-end]) 10px;
```

§ 7.2.2.1. Syntax of ‘repeat()’

The generic form of the ‘repeat()’ syntax is, approximately,

```
repeat( [ <positive-integer> | auto-fill | auto-fit ] , <track-list> )
```

The first argument specifies the number of repetitions. The second argument is a track list, which is repeated that number of times. However, there are some restrictions:

- The ‘repeat()’ notation can’t be nested.
- Automatic repetitions (‘auto-fill’ or ‘auto-fit’) cannot be combined with intrinsic or flexible sizes.

Thus the precise syntax of the ‘repeat()’ notation has several forms:

```
<track-repeat> = repeat( [ <positive-integer> ] , [ <line-names>? <track-size> ]+ <line-r
<auto-repeat>  = repeat( [ auto-fill | auto-fit ] , [ <line-names>? <fixed-size> ]+ <line
<fixed-repeat> = repeat( [ <positive-integer> ] , [ <line-names>? <fixed-size> ]+ <line-r
```

- The <track-repeat> variant can represent the repetition of any <track-size>, but is limited to a fixed number of repetitions.
- The <auto-repeat> variant can repeat automatically to fill a space, but requires definite track sizes so that the number of repetitions can be calculated. It can only appear once in the track list, but the same track list can also contain <fixed-repeat>s.

If the ‘repeat()’ function ends up placing two <line-names> adjacent to each other, the name lists are merged. For example, ‘repeat(2, [a] 1fr [b])’ is equivalent to ‘[a] 1fr [b a] 1fr [b]’.

§ 7.2.2.2. Repeat-to-fill: ‘auto-fill’ and ‘auto-fit’ repetitions

When ***‘auto-fill’*** is given as the repetition number, if the grid container has a definite size or max size in the relevant axis, then the number of repetitions is the largest possible positive integer that does not cause the grid to overflow its grid container (treating each track as its max track sizing function if that is definite or as its minimum track sizing function otherwise, and taking ‘gap’ into account); if any number of repetitions would overflow, then 1 repetition. Otherwise, if the grid container has a definite min size in the relevant axis,

the number of repetitions is the smallest possible positive integer that fulfills that minimum requirement. Otherwise, the specified track list repeats only once.

> EXAMPLE 18
> For example, the following code will create as many 25-character columns as will fit into the window width. If there is any remaining space, it will be distributed among the 25-character columns.
>
> ```
> body {
>   display: grid;
>   grid-template-columns: repeat(auto-fill, minmax(25ch, 1fr));
> }
> ```

The ***'auto-fit'*** keyword behaves the same as 'auto-fill', except that after grid item placement any empty repeated tracks are collapsed. An empty track is one with no in-flow grid items placed into or spanning across it. (This can result in *all* tracks being collapsed, if they're all empty.)

A ***collapsed track*** is treated as having a fixed track sizing function of '0px', and the gutters on either side of it—including any space allotted through distributed alignment—collapse.

For the purpose of finding the number of auto-repeated tracks, the UA must floor the track size to a UA-specified value to avoid division by zero. It is suggested that this floor be '1px'.

§ 7.2.3. Flexible Lengths: the 'fr' unit

A ***flexible length*** or ***'<flex>'*** is a dimension with the ***'fr'*** unit, which represents a fraction of the leftover space in the grid container. Tracks sized with 'fr' units are called ***flexible tracks*** as they flex in response to leftover space similar to how flex items fill space in a flex container.

> Note: <flex> values are not <length>s (nor are they compatible with <length>s, like some <percentage> values), so they cannot be represented in or combined with other unit types in 'calc()' expressions.

The distribution of leftover space occurs after all non-flexible track sizing functions have reached their maximum. The total size of such rows or columns is subtracted from the available space, yielding the leftover space, which is then divided among the flex-sized rows and columns in proportion to their flex factor.

> Note: Flexible lengths in a track list work similarly to flexible lengths with a zero base size in [CSS-FLEXBOX-1].

Each column or row's share of the leftover space can be computed as the column or row's `<flex> * <leftover space> / <sum of all flex factors>`.

> Note: If the sum of the flex factors is less than 1, they'll take up only a corresponding fraction of the leftover space, rather than expanding to fill the entire thing. This is similar to how Flexbox [CSS-FLEXBOX-1] acts when the sum of the 'flex' values is less than 1.

When the available space is infinite (which happens when the grid container's width or height is indefinite), flex-sized grid tracks are sized to their contents while retaining their respective proportions. The used size of each flex-sized grid track is computed by determining the 'max-content' size of each flex-sized grid track and dividing that size by the respective flex factor to determine a "hypothetical '1fr' size". The maximum of those is used as the resolved '1fr' length (the ***flex fraction***), which is then multiplied by each grid track's flex factor to determine its final size.

§ 7.2.4. Resolved Values

The 'grid-template-rows' and 'grid-template-columns' properties are resolved value special case properties. [CSSOM]

When an element's 'display' is 'grid' or 'inline-grid' and it generates a box, the resolved value of the 'grid-template-rows' and 'grid-template-columns' properties is the used value, serialized as follows:

- Every track listed, whether implicitly or explicitly created.
- Every track size given as a length in pixels, regardless of sizing function.
- A contiguous run of two or more tracks that have the same size and associated line names may be serialized with the 'repeat()' notation.

Otherwise, (e.g. when the element has 'display: none' or is not a grid) the resolved value is simply the computed value.

EXAMPLE 19

```
<style>
#grid {
  width: 500px;
  grid-template-columns:
    [a]     auto
    [b]     minmax(min-content, 1fr)
    [b c d] repeat(2, [e] 40px)
            repeat(5, auto);
}
</style>
<div id="grid">
  <div style="grid-column-start: 1; width: 50px"></div>
  <div style="grid-column-start: 9; width: 50px"></div>
</div>
<script>
  var gridElement = document.getElementById("grid");
  getComputedStyle(gridElement).gridTemplateColumns;
  // [a] 50px [b] 320px [b c d] repeat(2, [e] 40px) repeat(4, 0px) 50px
</script>
```

Note: In general, resolved values are the computed values, except for a small list of legacy 2.1 properties. However, compatibility with early implementations of this module requires us to define 'grid-template-rows' and 'grid-template-columns' as returning used values.

§ 7.3. Named Areas: the 'grid-template-areas' property

Name:	***'grid-template-areas'***
Value:	none | <string>+
Initial:	none
Applies to:	grid containers
Inherited:	no
Percentages:	n/a
Media:	visual
Computed value:	specified value
Canonical order:	per grammar
Animation type:	discrete

This property specifies ***named grid areas***, which are not associated with any particular grid item, but can be referenced from the grid-placement properties. The syntax of the 'grid-template-areas' property also provides a visualization of the structure of the grid, making the overall layout of the grid container easier to understand.

Values have the following meanings:

'none'

Indicates that no named grid areas, and likewise no explicit grid tracks, are defined by this property (though explicit grid tracks could still be created by 'grid-template-columns' or 'grid-template-rows').

> Note: In the absence of an explicit grid any rows/columns will be implicitly generated, and their size will be determined by the 'grid-auto-rows' and 'grid-auto-columns' properties.

'<string>+'

A row is created for every separate string listed for the 'grid-template-areas' property, and a column is created for each cell in the string, when parsed as follows:

Tokenize the string into a list of the following tokens, using longest-match semantics:

- A sequence of name code points, representing a ***named cell token*** with a name consisting of its code points.
- A sequence of one or more "." (U+002E FULL STOP), representing a ***null cell token***.
- A sequence of whitespace, representing nothing (do not produce a token).
- A sequence of any other characters, representing a ***trash token***.

> Note: These rules can produce cell names that do not match the <ident> syntax, such as "1st 2nd 3rd", which requires escaping when referencing those areas by name in other properties, like 'grid-row: \31st;' to reference the area named '1st'.

- A null cell token represents an unnamed area in the grid container.
- A named cell token creates a named grid area with the same name. Multiple named cell tokens within and between rows create a single named grid area that spans the corresponding grid cells.
- A trash token is a syntax error, and makes the declaration invalid.

All strings must have the same number of columns, or else the declaration is invalid. If a named grid area spans multiple grid cells, but those cells do not form a single filled-in rectangle, the declaration is invalid.

> Note: Non-rectangular or disconnected regions may be permitted in a future version of this module.

EXAMPLE 20

In this example, the 'grid-template-areas' property is used to create a page layout where areas are defined for header content (`head`), navigational content (`nav`), footer content (`foot`), and main content (`main`). Accordingly, the template creates three rows and two columns, with four named grid areas. The `head` area spans both columns and the first row of the grid.

```
#grid {
  display: grid;
  grid-template-areas: "head head"
                       "nav  main"
                       "foot ...."
}
#grid > header { grid-area: head; }
#grid > nav    { grid-area: nav; }
#grid > main   { grid-area: main; }
#grid > footer { grid-area: foot; }
```

§ 7.3.1. Implicit Named Lines

The 'grid-template-areas' property creates ***implicit named lines*** from the named grid areas in the template. For each named grid area *foo*, four implicit named lines are created: two named '*foo*-start', naming the row-start and column-start lines of the named grid area, and two named '*foo*-end', naming the row-end and column-end lines of the named grid area.

These named lines behave just like any other named line, except that they do not appear in the value of ‘grid-template-rows’/‘grid-template-columns’. Even if an explicit line of the same name is defined, the implicit named lines are just more lines with the same name.

§ 7.3.2. Implicit Named Areas

Since a named grid area is referenced by the implicit named lines it produces, explicitly adding named lines of the same form (‘foo-start’/‘foo-end’) effectively creates a named grid area. Such ***implicit named areas*** do not appear in the value of ‘grid-template-areas’, but can still be referenced by the grid-placement properties.

§ 7.4. Explicit Grid Shorthand: the ‘grid-template’ property

Name:	***‘grid-template’***
Value:	none | [<‘grid-template-rows’> / <‘grid-template-columns’>] | [<line-names>? <string> <track-size>? <line-names>?]+ [/ <explicit-track-list>]?
Initial:	see individual properties
Applies to:	grid containers
Inherited:	see individual properties
Percentages:	see individual properties
Media:	visual
Computed value:	see individual properties
Canonical order:	per grammar
Animatable:	see individual properties

The ‘grid-template’ property is a shorthand for setting ‘grid-template-columns’, ‘grid-template-rows’, and ‘grid-template-areas’ in a single declaration. It has several distinct syntax forms:

‘none’
: Sets all three properties to their initial values (‘none’).

‘<‘grid-template-rows’> / <‘grid-template-columns’>’
: Sets ‘grid-template-rows’ and ‘grid-template-columns’ to the specified values, respectively, and sets ‘grid-template-areas’ to ‘none’.

EXAMPLE 21

```
grid-template: auto 1fr / auto 1fr auto;
```

is equivalent to

```
grid-template-rows: auto 1fr;
grid-template-columns: auto 1fr auto;
grid-template-areas: none;
```

'[<line-names>? <string> <track-size>? <line-names>?]+ [/ <explicit-track-list>]?'

- Sets 'grid-template-areas' to the strings listed.
- Sets 'grid-template-rows' to the <track-size>s following each string (filling in 'auto' for any missing sizes), and splicing in the named lines defined before/after each size.
- Sets 'grid-template-columns' to the track listing specified after the slash (or 'none', if not specified).

This syntax allows the author to align track names and sizes inline with their respective grid areas.

EXAMPLE 22

```
grid-template: [header-top] "a   a   a"     [header-bottom]
               [main-top] "b   b   b" 1fr [main-bottom]
                        / auto 1fr auto;
```

is equivalent to

```
grid-template-areas: "a a a"
                     "b b b";
grid-template-rows: [header-top] auto [header-bottom main-top] 1fr [main-bottom];
grid-template-columns: auto 1fr auto;
```

and creates the following grid:

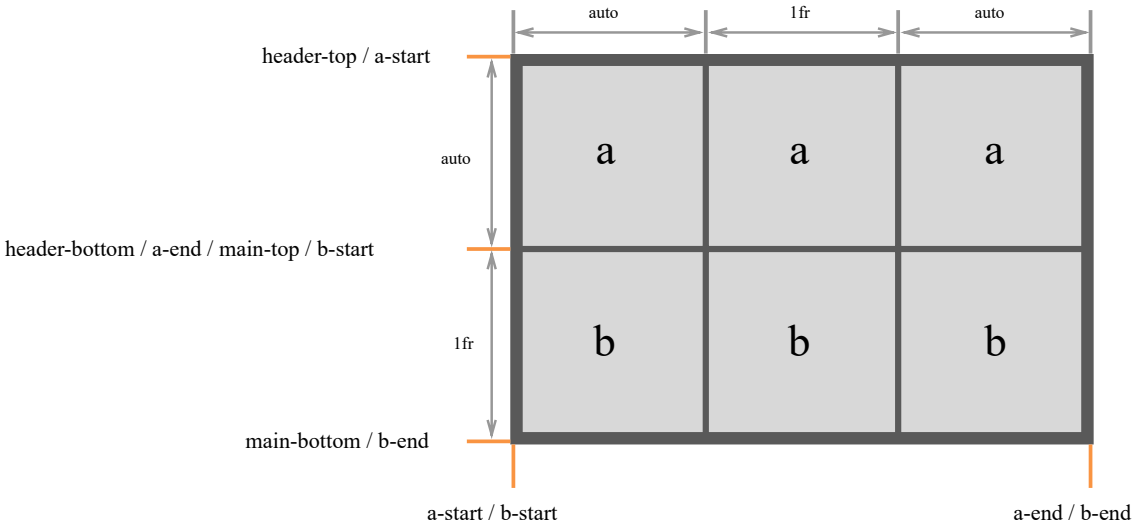


Figure 12 *The grid created by the declarations above. (The "a/b-start/end" names are created implicitly by the named grid areas.)*

Note: Note that the 'repeat()' function isn't allowed in these track listings, as the tracks are intended to visually line up one-to-one with the rows/columns in the "ASCII art".

Note: The 'grid' shorthand accepts the same syntax, but also resets the implicit grid properties to their initial values. Unless authors want those to cascade in separately, it is therefore recommended to use 'grid' instead of 'grid-template'.

§ 7.5. The Implicit Grid

The 'grid-template-rows', 'grid-template-columns', and 'grid-template-areas' properties define a fixed number of tracks that form the explicit grid. When grid items are positioned outside of these bounds, the grid container generates ***implicit grid tracks*** by adding ***implicit grid lines*** to the grid. These lines together with

the explicit grid form the ***implicit grid***. The ‘grid-auto-rows’ and ‘grid-auto-columns’ properties size these implicit grid tracks.

The ‘grid-auto-flow’ property controls auto-placement of grid items without an explicit position. Once the explicit grid is filled (or if there is no explicit grid) auto-placement will also cause the generation of implicit grid tracks.

The ‘grid’ shorthand property can set the ***implicit grid properties*** (‘grid-auto-flow’, ‘grid-auto-rows’, and ‘grid-auto-columns’) together with the explicit grid properties in a single declaration.

§ 7.6. Implicit Track Sizing: the ‘grid-auto-rows’ and ‘grid-auto-columns’ properties

Name:	***‘grid-auto-columns’***, ***‘grid-auto-rows’***
Value:	<track-size>+
Initial:	auto
Applies to:	grid containers
Inherited:	no
Percentages:	see Track Sizing
Media:	visual
Computed value:	see Track Sizing
Canonical order:	per grammar
Animation type:	discrete

If a grid item is positioned into a row or column that is not explicitly sized by ‘grid-template-rows’ or ‘grid-template-columns’, implicit grid tracks are created to hold it. This can happen either by explicitly positioning into a row or column that is out of range, or by the auto-placement algorithm creating additional rows or columns. The ‘grid-auto-columns’ and ‘grid-auto-rows’ properties specify the size of such implicitly-created tracks.

If multiple track sizes are given, the pattern is repeated as necessary to find the size of the implicit tracks. The first implicit grid track after the explicit grid receives the first specified size, and so on forwards; and the last implicit grid track before the explicit grid receives the last specified size, and so on backwards.

EXAMPLE 23

```
<style>
  #grid {
    display: grid;
    grid-template-columns: 20px;
    grid-auto-columns: 40px;
    grid-template-rows: 20px;
    grid-auto-rows: 40px;
  }
  #A { grid-column: 1; grid-row: 1; }
  #B { grid-column: 2; grid-row: 1; }
  #C { grid-column: 1; grid-row: 2; }
  #D { grid-column: 2; grid-row: 2; }
</style>

<div id="grid">
  <div id="A">A</div>
  <div id="B">B</div>
  <div id="C">C</div>
  <div id="D">D</div>
</div>
```

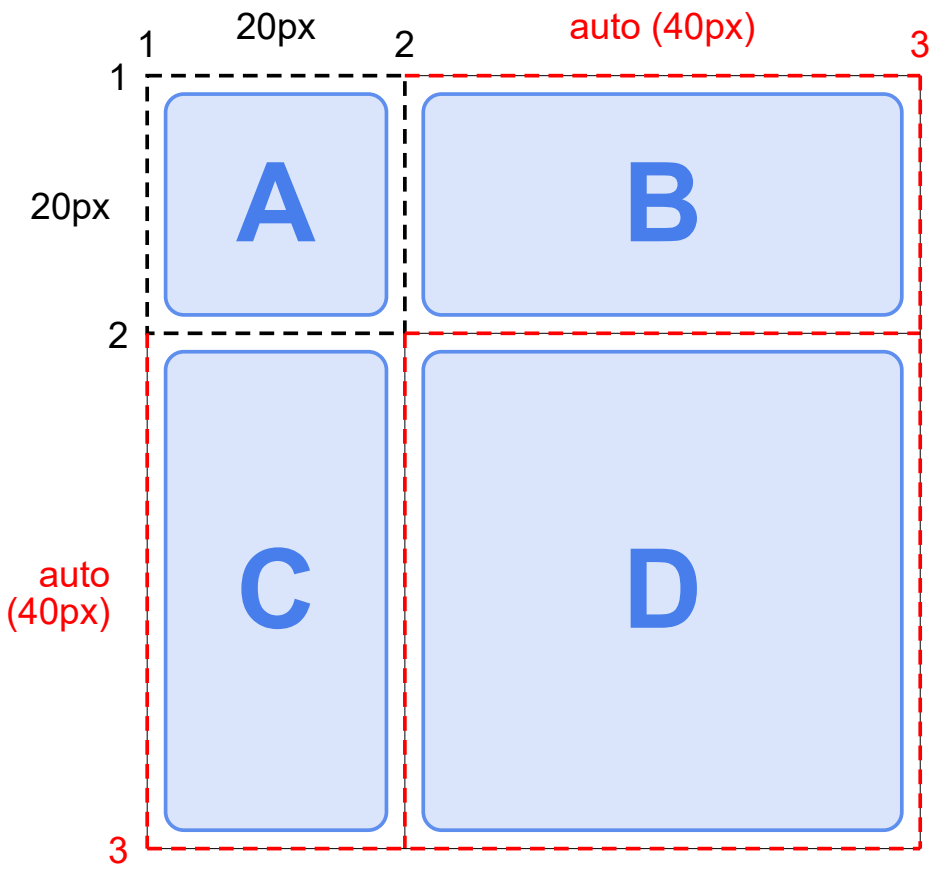


Figure 13 *A 2×2 grid with one explicit 20px×20px grid cell in the first row+column and three additional cells resulting from the implicit 40px column and row generated to hold the additional grid items.*

§ 7.7. ***Automatic Placement***: the ‘grid-auto-flow’ property

Name:	***'grid-auto-flow'***
Value:	[row | column] || dense
Initial:	row
Applies to:	grid containers
Inherited:	no
Percentages:	n/a
Media:	visual
Computed value:	specified value
Canonical order:	per grammar
Animation type:	discrete

Grid items that aren't explicitly placed are automatically placed into an unoccupied space in the grid container by the auto-placement algorithm. 'grid-auto-flow' controls how the auto-placement algorithm works, specifying exactly how auto-placed items get flowed into the grid. See §8.5 Grid Item Placement Algorithm for details on precisely how the auto-placement algorithm works.

'row'
: The auto-placement algorithm places items by filling each row in turn, adding new rows as necessary. If neither 'row' nor 'column' is provided, 'row' is assumed.

'column'
: The auto-placement algorithm places items by filling each column in turn, adding new columns as necessary.

'dense'
: If specified, the auto-placement algorithm uses a "dense" packing algorithm, which attempts to fill in holes earlier in the grid if smaller items come up later. This may cause items to appear out-of-order, when doing so would fill in holes left by larger items.

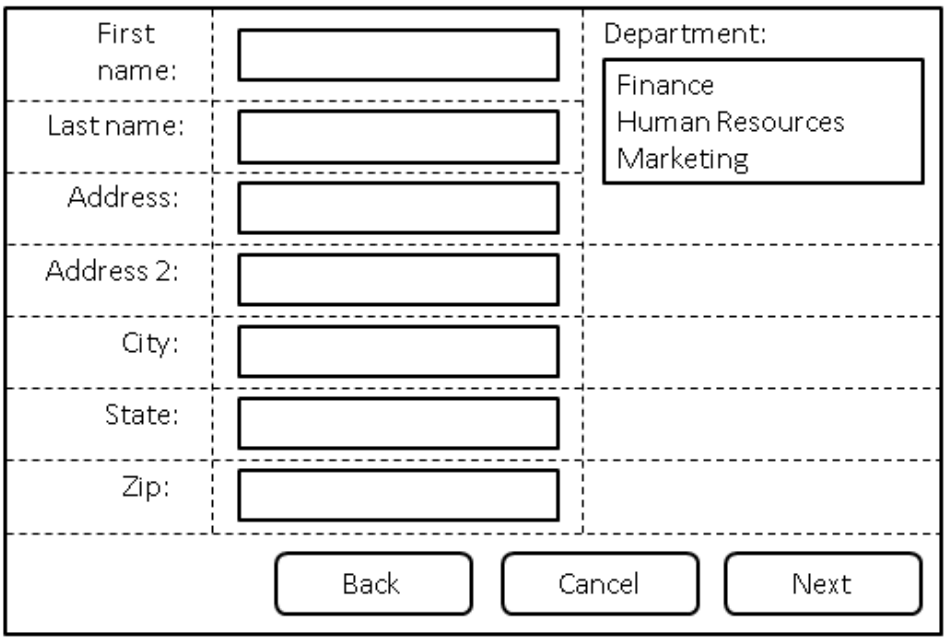
  If omitted, a "sparse" algorithm is used, where the placement algorithm only ever moves "forward" in the grid when placing items, never backtracking to fill holes. This ensures that all of the auto-placed items appear "in order", even if this leaves holes that could have been filled by later items.

Note: A future level of this module is expected to add a value that flows auto-positioned items together into a single "default" cell.

Auto-placement takes grid items in order-modified document order.

EXAMPLE 24

In the following example, there are three columns, each auto-sized to their contents. No rows are explicitly defined. The 'grid-auto-flow' property is 'row' which instructs the grid to search across its three columns starting with the first row, then the next, adding rows as needed until sufficient space is located to accommodate the position of any auto-placed grid item.



A form arranged using automatic placement.

```
<style type="text/css">
form {
  display: grid;
  /* Define three columns, all content-sized,
     and name the corresponding lines. */
  grid-template-columns: [labels] auto [controls] auto [oversized] auto;
  grid-auto-flow: row dense;
}
form > label {
  /* Place all labels in the "labels" column and
     automatically find the next available row. */
  grid-column: labels;
  grid-row: auto;
}
form > input, form > select {
  /* Place all controls in the "controls" column and
     automatically find the next available row. */
  grid-column: controls;
  grid-row: auto;
}

#department-block {
  /* Auto place this item in the "oversized" column
     in the first row where an area that spans three rows
     won’t overlap other explicitly placed items or areas
     or any items automatically placed prior to this area. */
  grid-column: oversized;
  grid-row: span 3;
}

/* Place all the buttons of the form
   in the explicitly defined grid area. */
#buttons {
  grid-row: auto;

  /* Ensure the button area spans the entire grid element
     in the row axis. */
  grid-column: 1 / -1;
  text-align: end;
}
</style>
<form>
  <label for="firstname">First name:</label>
  <input type="text" id="firstname" name="firstname" />
  <label for="lastname">Last name:</label>
  <input type="text" id="lastname" name="lastname" />
  <label for="address">Address:</label>
  <input type="text" id="address" name="address" />
  <label for="address2">Address 2:</label>
  <input type="text" id="address2" name="address2" />
  <label for="city">City:</label>
  <input type="text" id="city" name="city" />
```

```
  <label for="state">State:</label>
  <select type="text" id="state" name="state">
    <option value="WA">Washington</option>
  </select>
  <label for="zip">Zip:</label>
  <input type="text" id="zip" name="zip" />

  <div id="department-block">
    <label for="department">Department:</label>
    <select id="department" name="department" multiple>
      <option value="finance">Finance</option>
      <option value="humanresources">Human Resources</option>
      <option value="marketing">Marketing</option>
    </select>
  </div>

  <div id="buttons">
    <button id="cancel">Cancel</button>
    <button id="back">Back</button>
    <button id="next">Next</button>
  </div>
</form>
```

§ 7.8. Grid Definition Shorthand: the ‘grid’ property

Name:	***‘grid’***
Value:	<‘grid-template’> | <‘grid-template-rows’> / [auto-flow && dense?] <‘grid-auto-columns’>? | [auto-flow && dense?] <‘grid-auto-rows’>? / <‘grid-template-columns’>
Initial:	see individual properties
Applies to:	grid containers
Inherited:	see individual properties
Percentages:	see individual properties
Media:	visual
Computed value:	see individual properties
Canonical order:	per grammar
Animatable:	see individual properties

The ‘grid’ property is a shorthand that sets all of the explicit grid properties (‘grid-template-rows’, ‘grid-template-columns’, and ‘grid-template-areas’), and all the implicit grid properties (‘grid-auto-rows’, ‘grid-auto-columns’, and ‘grid-auto-flow’), in a single declaration. (It does not reset the gutter properties.) Its syntax matches ‘grid-template’, plus an additional syntax form for defining auto-flow grids:

‘<‘grid-template-rows’> / [auto-flow && dense?] <‘grid-auto-columns’>?’
‘[auto-flow && dense?] <‘grid-auto-rows’>? / <‘grid-template-columns’>’

Sets up auto-flow, by setting the tracks in one axis explicitly (setting either ‘grid-template-rows’ or ‘grid-template-columns’ as specified, and setting the other to ‘none’), and specifying how to auto-repeat the tracks in the other axis (setting either ‘grid-auto-rows’ or ‘grid-auto-columns’ as specified, and setting the other to ‘auto’). ‘grid-auto-flow’ is also set to either ‘row’ or ‘column’ accordingly, with ‘dense’ if it’s specified.

All other ‘grid’ sub-properties are reset to their initial values.

Note: Note that you can only specify the explicit *or* the implicit grid properties in a single ‘grid’ declaration. The sub-properties you don’t specify are set to their initial value, as normal for shorthands.

EXAMPLE 25
In addition to accepting the ‘grid-template’ shorthand syntax for setting up the explicit grid, the ‘grid’ shorthand can also easily set up parameters for an auto-formatted grid. For example, ‘grid: auto-flow 1fr / 100px;’ is equivalent to

```
grid-template: none / 100px;
grid-auto-flow: row;
grid-auto-rows: 1fr;
grid-auto-columns: auto;
```

Similarly, ‘grid: none / auto-flow 1fr’ is equivalent to

```
grid-template: none;
grid-auto-flow: column;
grid-auto-rows: auto;
grid-auto-columns: 1fr;
```

§ 8. Placing Grid Items

Every grid item is associated with a grid area, a rectangular set of adjacent grid cells that the grid item occupies. This grid area defines the containing block for the grid item within which the self-alignment properties (‘justify-self’ and ‘align-self’) determine their actual position. The cells that a grid item occupies also influence the sizing of the grid’s rows and columns, defined in §11 Grid Sizing.

The location of a grid item’s grid area within the grid is defined by its ***placement***, which consists of a grid position and a grid span:

grid position

The grid item's location in the grid. A grid position can be either ***definite*** (explicitly specified) or ***automatic*** (determined by auto-placement).

grid span
: How many grid tracks the grid item occupies. A grid item's grid span is always ***definite***, defaulting to 1 if it can't be otherwise determined.

The ***grid-placement properties***—the longhands 'grid-row-start', 'grid-row-end', 'grid-column-start', 'grid-column-end', and their shorthands 'grid-row', 'grid-column', and 'grid-area'—allow the author to specify a grid item's placement by providing any (or none) of the following six pieces of information:

	Row	Column
Start	row-start line	column-start line
End	row-end line	column-end line
Span	row span	column span

A definite value for any two of *Start*, *End*, and *Span* in a given dimension implies a definite value for the third.

The following table summarizes the conditions under which a grid position or span is *definite* or *automatic*:

	Position	Span
Definite	At least one specified line	Explicit, implicit, or defaulted span.
Automatic	No lines explicitly specified	N/A

§ 8.1. Common Patterns for Grid Placement

This section is informative.

The grid-placement property longhands are organized into three shorthands:

<table>
<tr><td colspan="4">'grid-area'</td></tr>
<tr><td colspan="2">'grid-column'</td><td colspan="2">'grid-row'</td></tr>
<tr><td>'grid-column-start'</td><td>'grid-column-end'</td><td>'grid-row-start'</td><td>'grid-row-end'</td></tr>
</table>

§ 8.1.1. Named Areas

An item can be placed into a named grid area (such as those produced by the template in 'grid-template-areas') by specifying the area's name in 'grid-area':

EXAMPLE 26

```
article {
  grid-area: main;
  /* Places item into the named area "main". */
}
```

An item can also be *partially* aligned with a named grid area, with other edges aligned to some other line:

EXAMPLE 27

```
.one {
  grid-row-start: main;
  /* Align the row-start edge to the start edge of the "main" named area. */
}
```

§ 8.1.2. Numeric Indexes and Spans

Grid items can be positioned and sized by number, which is particularly helpful for script-driven layouts:

EXAMPLE 28

```
.two {
  grid-row: 2;    /* Place item in the second row. */
  grid-column: 3; /* Place item in the third column. */
  /* Equivalent to grid-area: 2 / 3; */
}
```

By default, a grid item has a span of 1. Different spans can be given explicitly:

EXAMPLE 29

```
.three {
  grid-row: 2 / span 5;
  /* Starts in the 2nd row,
     spans 5 rows down (ending in the 7th row). */
}

.four {
  grid-row: span 5 / 7;
  /* Ends in the 7th row,
     spans 5 rows up (starting in the 2nd row). */
}
```

Note: Note that grid indexes are writing mode relative. For example, in a right-to-left language like Arabic, the first column is the rightmost column.

§ 8.1.3. Named Lines and Spans

Instead of counting lines by number, named lines can be referenced by their name:

EXAMPLE 30

```
.five {
  grid-column: first / middle;
  /* Span from line "first" to line "middle". */
}
```

Note: Note that if a named grid area and a named line have the same name, the placement algorithm will prefer to use named grid area's lines instead.

If there are multiple lines of the same name, they effectively establish a named set of grid lines, which can be exclusively indexed by filtering the placement by name:

EXAMPLE 31

```
.six {
  grid-row: text 5 / text 7;
  /* Span between the 5th and 7th lines named "text". */
  grid-row: text 5 / span text 2;
  /* Same as above - start at the 5th line named "text",
     then span across two more "text" lines, to the 7th. */
}
```

§ 8.1.4. Auto Placement

A grid item can be automatically placed into the next available empty grid cell, growing the grid if there's no space left.

EXAMPLE 32

```
.eight {
  grid-area: auto; /* Initial value */
}
```

This can be used, for example, to list a number of sale items on a catalog site in a grid pattern.

Auto-placement can be combined with an explicit span, if the item should take up more than one cell:

EXAMPLE 33

```
.nine {
  grid-area: span 2 / span 3;
  /* Auto-placed item, covering two rows and three columns. */
}
```

Whether the auto-placement algorithm searchs across and adds rows, or searches across and adds columns, is controlled by the 'grid-auto-flow' property.

Note: By default, the auto-placement algorithm looks linearly through the grid without backtracking; if it has to skip some empty spaces to place a larger item, it will not return to fill those spaces. To change this behavior, specify the 'dense' keyword in 'grid-auto-flow'.

§ 8.2. Grid Item Placement vs. Source Order

> "With great power comes great responsibility."

The abilities of the grid-placement properties allow content to be freely arranged and reordered within the grid, such that the visual presentation can be largely disjoint from the underlying document source order. These abilities allow the author great freedom in tailoring the rendering to different devices and modes of presentation e.g. using media queries. However **they are not a substitute for correct source ordering**.

Correct source order is important for speech, for sequential navigation (such as keyboard navigation), and non-CSS UAs such as search engines, tactile browsers, etc. Grid placement *only* affects the visual presentation! This allows authors to optimize the document source for non-CSS/non-visual interaction modes, and use grid placement techniques to further manipulate the visual presentation so as to leave that source order intact.

§ 8.3. Line-based Placement: the 'grid-row-start', 'grid-column-start', 'grid-row-end', and 'grid-column-end' properties

Name:	***'grid-row-start'**, **'grid-column-start'**, **'grid-row-end'**, **'grid-column-end'***
Value:	<grid-line>
Initial:	auto
Applies to:	grid items and absolutely-positioned boxes whose containing block is a grid container
Inherited:	no
Percentages:	n/a
Media:	visual
Computed value:	specified value
Canonical order:	per grammar
Animation type:	discrete

```
<grid-line> =
  auto |
  <custom-ident> |
  [ <integer> && <custom-ident>? ] |
  [ span && [ <integer> || <custom-ident> ] ]
```

The 'grid-row-start', 'grid-column-start', 'grid-row-end', and 'grid-column-end' properties determine a grid item's size and location within the grid by contributing a line, a span, or nothing (automatic) to its grid placement, thereby specifying the inline-start, block-start, inline-end, and block-end edges of its grid area.

Values have the following meanings:

'<custom-ident>'

First attempt to match the grid area's edge to a named grid area: if there is a named line with the name "<custom-ident>-start' (for 'grid-*-start') / '<custom-ident>-end" (for 'grid-*-end'), contributes the first such line to the grid item's placement.

> Note: Named grid areas automatically generate implicit named lines of this form, so specifying 'grid-row-start: foo' will choose the start edge of that named grid area (unless another line named 'foo-start' was explicitly specified before it).

Otherwise, treat this as if the integer '1' had been specified along with the <custom-ident>.

'<integer> && <custom-ident>?'

Contributes the *N*th grid line to the grid item's placement. If a negative integer is given, it instead counts in reverse, starting from the end edge of the explicit grid.

If a name is given as a <custom-ident>, only lines with that name are counted. If not enough lines with that name exist, all implicit grid lines are assumed to have that name for the purpose of finding this position.

An <integer> value of zero makes the declaration invalid.

'span && [<integer> || <custom-ident>]'

Contributes a grid span to the grid item's placement such that the corresponding edge of the grid item's grid area is *N* lines from its opposite edge in the corresponding direction. For example, 'grid-column-end: span 2' indicates the second grid line in the endward direction from the 'grid-column-start' line.

If a name is given as a <custom-ident>, only lines with that name are counted. If not enough lines with that name exist, all implicit grid lines on the side of the explicit grid corresponding to the search direction are assumed to have that name for the purpose of counting this span.

EXAMPLE 34

For example, given the following declarations:

```
.grid { grid-template-columns: 100px; }
.griditem { grid-column: span foo / 4; }
```

The grid container has an explicit grid with two grid lines, numbered 1 and 2. The grid item's column-end edge is specified to be at line 4, so two lines are generated in the endward side of the implicit grid.

Its column-start edge must be the first "foo" line it can find startward of that. There is no "foo" line in the grid, though, so the only possibility is a line in the implicit grid. Line 3 is not a candidate, because it's on the endward side of the explicit grid, while the 'grid-column-start' span forces it to search startward. So, the only option is for the implicit grid to generate a line on the startward side of the explicit grid.

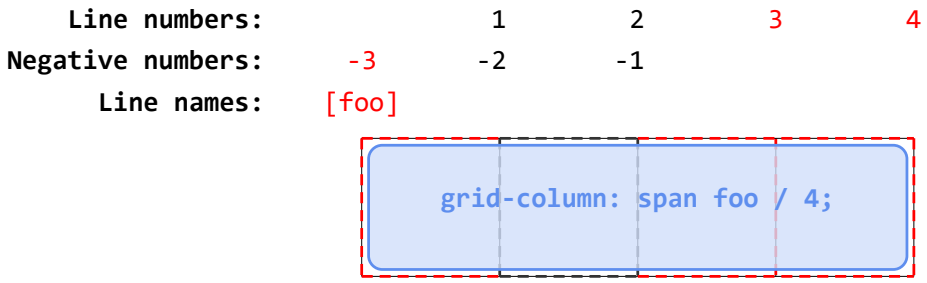


Figure 15 *An illustration of the result.*

If the <integer> is omitted, it defaults to '1'. Negative integers or zero are invalid.

'auto'

The property contributes nothing to the grid item's placement, indicating auto-placement or a default span of one. (See §8 Placing Grid Items, above.)

In all the above productions, the <custom-ident> additionally excludes the keyword 'span'.

EXAMPLE 35

Given a single-row, 8-column grid and the following 9 named lines:

```
1  2  3  4  5  6  7  8  9
+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |
A  B  C  A  B  C  A  B  C
|  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+
```

The following declarations place the grid item between the lines indicated by index:

```
grid-column-start: 4; grid-column-end: auto;
/* Line 4 to line 5 */

grid-column-start: auto; grid-column-end: 6;
/* Line 5 to line 6 */

grid-column-start: C; grid-column-end: C -1;
/* Line 3 to line 9 */

grid-column-start: C; grid-column-end: span C;
/* Line 3 to line 6 */

grid-column-start: span C; grid-column-end: C -1;
/* Line 6 to line 9 */

grid-column-start: span C; grid-column-end: span C;
/* Error: The end span is ignored, and an auto-placed
   item can't span to a named line.
   Equivalent to 'grid-column: span 1;'. */

grid-column-start: 5; grid-column-end: C -1;
/* Line 5 to line 9 */

grid-column-start: 5; grid-column-end: span C;
/* Line 5 to line 6 */

grid-column-start: 8; grid-column-end: 8;
/* Error: line 8 to line 9 */

grid-column-start: B 2; grid-column-end: span 1;
/* Line 5 to line 6 */
```

§ 8.3.1. Grid Placement Conflict Handling

If the placement for a grid item contains two lines, and the start line is further end-ward than the end line, swap the two lines. If the start line is *equal* to the end line, remove the end line.

If the placement contains two spans, remove the one contributed by the end grid-placement property.

If the placement contains only a span for a named line, replace it with a span of 1.

§ 8.4. Placement Shorthands: the 'grid-column', 'grid-row', and 'grid-area' properties

Name:	***'grid-row'***, ***'grid-column'***
Value:	<grid-line> [/ <grid-line>]?
Initial:	see individual properties
Applies to:	grid items and absolutely-positioned boxes whose containing block is a grid container
Inherited:	see individual properties
Percentages:	see individual properties
Media:	visual
Computed value:	see individual properties
Canonical order:	per grammar
Animation type:	discrete

The 'grid-row' and 'grid-column' properties are shorthands for 'grid-row-start'/'grid-row-end' and 'grid-column-start'/'grid-column-end', respectively.

If two <grid-line> values are specified, the 'grid-row-start'/'grid-column-start' longhand is set to the value before the slash, and the 'grid-row-end'/'grid-column-end' longhand is set to the value after the slash.

When the second value is omitted, if the first value is a <custom-ident>, the 'grid-row-end'/'grid-column-end' longhand is also set to that <custom-ident>; otherwise, it is set to 'auto'.

Name:	***'grid-area'***
Value:	<grid-line> [/ <grid-line>]{0,3}
Initial:	see individual properties
Applies to:	grid items and absolutely-positioned boxes whose containing block is a grid container
Inherited:	see individual properties
Percentages:	see individual properties
Media:	visual
Computed value:	see individual properties
Canonical order:	per grammar
Animation type:	discrete

If four <grid-line> values are specified, 'grid-row-start' is set to the first value, 'grid-column-start' is set to the second value, 'grid-row-end' is set to the third value, and 'grid-column-end' is set to the fourth value.

When 'grid-column-end' is omitted, if 'grid-column-start' is a <custom-ident>, 'grid-column-end' is set to that <custom-ident>; otherwise, it is set to 'auto'.

When 'grid-row-end' is omitted, if 'grid-row-start' is a <custom-ident>, 'grid-row-end' is set to that <custom-ident>; otherwise, it is set to 'auto'.

When 'grid-column-start' is omitted, if 'grid-row-start' is a <custom-ident>, all four longhands are set to that value. Otherwise, it is set to 'auto'.

Note: The resolution order for this shorthand is row-start/column-start/row-end/column-end, which goes CCW for LTR pages, the opposite direction of the related 4-edge properties using physical directions, like 'margin'.

§ 8.5. Grid Item Placement Algorithm

The following ***grid item placement algorithm*** resolves automatic positions of grid items into definite positions, ensuring that every grid item has a well-defined grid area to lay out into. (Grid spans need no special resolution; if they're not explicitly specified, they default to 1.)

Note: This algorithm can result in the creation of new rows or columns in the implicit grid, if there is no room in the explicit grid to place an auto-positioned grid item.

Every grid cell (in both the explicit and implicit grids) can be ***occupied*** or ***unoccupied***. A cell is occupied if it's covered by the grid area of a grid item with a definite grid position; otherwise, the cell is unoccupied. A cell's occupied/unoccupied status can change during this algorithm.

To aid in clarity, this algorithm is written with the assumption that 'grid-auto-flow' has 'row' specified. If it is instead set to 'column', swap all mentions of rows and columns, inline and block, etc. in this algorithm.

> Note: The auto-placement algorithm works with the grid items in order-modified document order, not their original document order.

0. **Generate anonymous grid items** as described in §6 Grid Items. (Anonymous grid items are always auto-placed, since their boxes can't have any grid-placement properties specified.)
1. **Position anything that's not auto-positioned.**
2. **Process the items locked to a given row.**

   For each grid item with a definite row position (that is, the 'grid-row-start' and 'grid-row-end' properties define a definite grid position), in order-modified document order:

   "sparse" packing (default behavior)
   Set the column-start line of its placement to the earliest (smallest positive index) line index that ensures this item's grid area will not overlap any occupied grid cells and that is past any grid items previously placed in this row by this step.

   "dense" packing ('dense' specified)
   Set the column-start line of its placement to the earliest (smallest positive index) line index that ensures this item's grid area will not overlap any occupied grid cells.
3. **Determine the columns in the implicit grid.**

   Create columns in the implicit grid:

   1. Start with the columns from the explicit grid.
   2. Among all the items with a definite column position (explicitly positioned items, items positioned in the previous step, and items not yet positioned but with a definite column) add columns to the beginning and end of the implicit grid as necessary to accomodate those items.
   3. If the largest column span among all the items *without* a definite column position is larger than the width of the implicit grid, add columns to the end of the implicit grid to accomodate that column span.

EXAMPLE 36

For example, in the following style fragment:

```
#grid {
  display: grid;
  grid-template-columns: repeat(5, 100px);
  grid-auto-flow: row;
}
#grid-item {
  grid-column: 4 / span 3;
}
```

The number of columns needed is 6. The explicit grid provides its 5 columns (from 'grid-template-columns') with lines number 1 through 6, but `#grid-item`'s column position means it ends on line 7, which requires an additional column added to the end of the implicit grid.

4. **Position the remaining grid items.**

   The ***auto-placement cursor*** defines the current "insertion point" in the grid, specified as a pair of row and column grid lines. Initially the auto-placement cursor is set to the start-most row and column lines in the implicit grid.

   The 'grid-auto-flow' value in use determines how to position the items:

   "sparse" packing (default behavior)

   For each grid item that hasn't been positioned by the previous steps, in order-modified document order:

   If the item has a definite column position:

   1. Set the column position of the cursor to the grid item's column-start line. If this is less than the previous column position of the cursor, increment the row position by 1.
   2. Increment the cursor's row position until a value is found where the grid item does not overlap any occupied grid cells (creating new rows in the implicit grid as necessary).
   3. Set the item's row-start line to the cursor's row position, and set the item's row-end line according to its span from that position.

   If the item has an automatic grid position in both axes:

   1. Increment the column position of the auto-placement cursor until either this item's grid area does not overlap any occupied grid cells, or the cursor's column position, plus the item's column span, overflow the number of columns in the implicit grid, as determined earlier in this algorithm.
   2. If a non-overlapping position was found in the previous step, set the item's row-start and column-start lines to the cursor's position. Otherwise, increment the auto-placement cursor's row position (creating new rows in the implicit grid as necessary), set its column position to the start-most column line in the implicit grid, and return to the previous step.

   "dense" packing ('dense' specified)

For each grid item that hasn't been positioned by the previous steps, in order-modified document order:

If the item has a definite column position:

1. Set the row position of the cursor to the start-most row line in the implicit grid. Set the column position of the cursor to the grid item's column-start line.
2. Increment the auto-placement cursor's row position until a value is found where the grid item does not overlap any occupied grid cells (creating new rows in the implicit grid as necessary).
3. Set the item's row-start line index to the cursor's row position. (Implicitly setting the item's row-end line according to its span, as well.)

If the item has an automatic grid position in both axes:

1. Set the cursor's row and column positions to start-most row and column lines in the implicit grid.
2. Increment the column position of the auto-placement cursor until either this item's grid area does not overlap any occupied grid cells, or the cursor's column position, plus the item's column span, overflow the number of columns in the implicit grid, as determined earlier in this algorithm.
3. If a non-overlapping position was found in the previous step, set the item's row-start and column-start lines to the cursor's position. Otherwise, increment the auto-placement cursor's row position (creating new rows in the implicit grid as necessary), reset its column position to the start-most column line in the implicit grid, and return to the previous step.

§ 9. Absolute Positioning

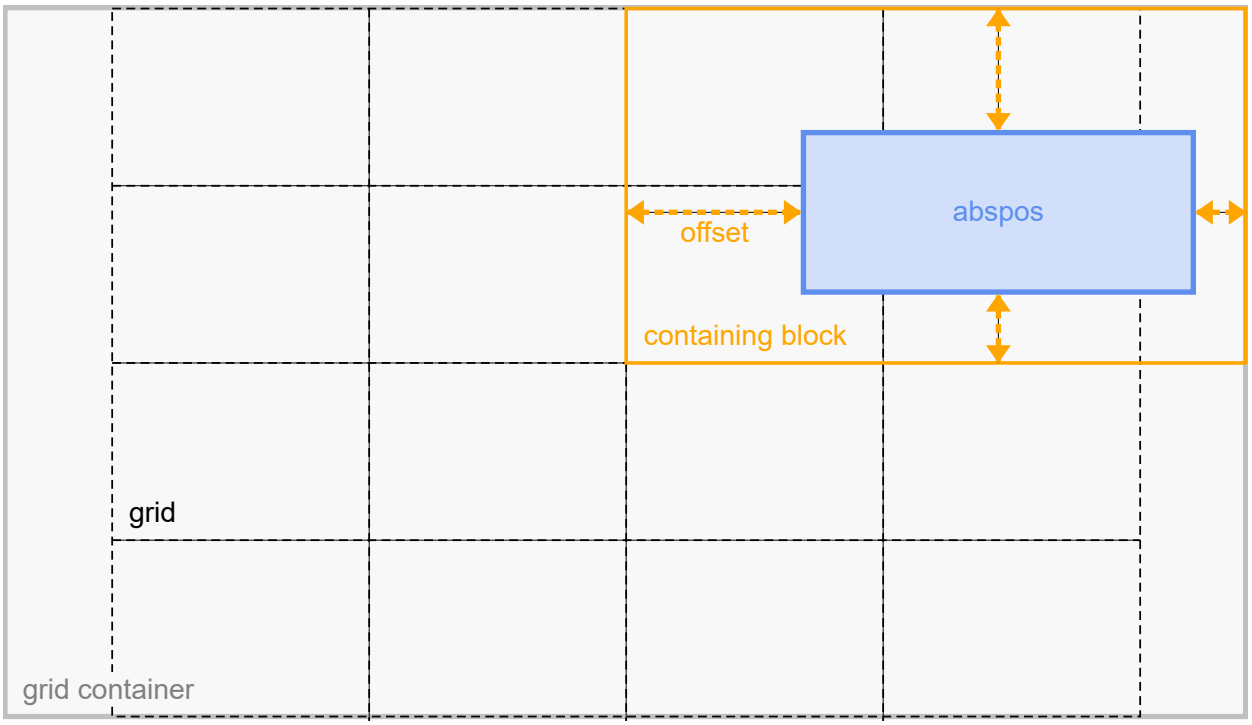
§ 9.1. With a Grid Container as Containing Block

If an absolutely positioned element's containing block is generated by a grid container, the containing block corresponds to the grid area determined by its grid-placement properties. The offset properties ('top'/'right'/'bottom'/'left') then indicate offsets inwards from the corresponding edges of this containing block, as normal.

> Note: While absolutely-positioning an element to a grid container does allow it to align to that container's grid lines, such elements do not take up space or otherwise participate in the layout of the grid.

EXAMPLE 37

```
.grid {
  grid: 1fr 1fr 1fr 1fr / 10rem 10rem 10rem 10rem;
  /* 4 equal-height rows filling the grid container,
     4 columns of '10rem' each */
  justify-content: center;
  /* center the grid horizontally within the grid container */
  position: relative;
  /* Establish abspos containing block */
}

.abspos {
  grid-row-start: 1;     /* 1st grid row line = top of grid container */
  grid-row-end: span 2;  /* 3rd grid row line */
  grid-column-start: 3;  /* 3rd grid col line */
  grid-column-end: auto; /* right padding edge */
  /* Containing block covers the top right quadrant of the grid container */

  position: absolute;
  top: 70px;
  bottom: 40px;
  left: 100px;
  right: 30px;
}
```



> Note: Grids and the grid-placement properties are flow-relative, while the offset properties (‘left’, ‘right’, ‘top’, and ‘bottom’) are physical, so if the ‘direction’ or ‘writing-mode’ properties change, the grid will transform to match, but the offsets won’t.

Instead of auto-placement, an ‘auto’ value for a grid-placement property contributes a special line to the placement whose position is that of the corresponding padding edge of the grid container (the padding edge of the scrollable area, if the grid container overflows). These lines become the first and last lines (0th and -0th) of the ***augmented grid*** used for positioning absolutely-positioned items.

> Note: Thus, by default, the absolutely-positioned box’s containing block will correspond to the padding edges of the grid container, as it does for block containers.

Absolute positioning occurs after layout of the grid and its in-flow contents, and does not contribute to the sizing of any grid tracks or affect the size/configuration of the grid in any way. If a grid-placement property refers to a non-existent line either by explicitly specifying such a line or by spanning outside of the existing implicit grid, it is instead treated as specifying ‘auto’ (instead of creating new implicit grid lines).

If the placement only contains a grid span, replace it with the two ‘auto’ lines in that axis. (This happens when both grid-placement properties in an axis contributed a span originally, and §8.3.1 Grid Placement Conflict Handling caused the second span to be ignored.)

§ 9.2. With a Grid Container as Parent

An absolutely-positioned child of a grid container is out-of-flow and not a grid item, and so does not affect the placement of other items or the sizing of the grid.

The static position [CSS21] of an absolutely-positioned child of a grid container is determined as if it were the sole grid item in a grid area whose edges coincide with the padding edges of the grid container. However, if the grid container parent is also the generator of the absolutely positioned element’s containing block, instead use the grid area determined in §9.1 With a Grid Container as Containing Block.

> Note: Note that this position is affected by the values of ‘justify-self’ and ‘align-self’ on the child, and that, as in most other layout models, the absolutely-positioned child has no effect on the size of the containing block or layout of its contents.

§ 10. Alignment and Spacing

After a grid container’s grid tracks have been sized, and the dimensions of all grid items are finalized, grid items can be aligned within their grid areas.

The ‘margin’ properties can be used to align items in a manner similar to what margins can do in block layout. Grid items also respect the box alignment properties from the CSS Box Alignment Module [CSS-ALIGN-3], which allow easy keyword-based alignment of items in both the row axis and column axis.

By default, grid items stretch to fill their grid area. However, if 'justify-self' or 'align-self' compute to a value other than 'stretch' or margins are 'auto', grid items will auto-size to fit their contents.

§ 10.1. Gutters: the 'row-gap', 'column-gap', and 'gap' properties

The 'row-gap' and 'column-gap' properties (and their 'gap' shorthand), when specified on a grid container, define the gutters between grid rows and grid columns. Their syntax is defined in CSS Box Alignment 3 §8 Gaps Between Boxes.

The effect of these properties is as though the affected grid lines acquired thickness: the grid track between two grid lines is the space between the gutters that represent them. For the purpose of track sizing, each gutter is treated as an extra, empty track of the specified size.

> Note: Additional spacing may be added between tracks due to 'justify-content'/'align-content'. See §11.1 Grid Sizing Algorithm. This space effectively increases the size of the gutters.

If a grid is fragmented between tracks, the gutter spacing between those tracks must be suppressed. Note that gutters are suppressed even after forced breaks, unlike margins.

Gutters only appear *between* tracks of the implicit grid; there is no gutter before the first track or after the last track. (In particular, there is no gutter between the first/last track of the implicit grid and the "auto" lines in the augmented grid.)

When a collapsed track's gutters ***collapse***, they coincide exactly—the two gutters overlap so that their start and end edges coincide. If one side of a collapsed track does not have a gutter (e.g. if it is the first or last track of the implicit grid), then collapsing its gutters results in no gutter on either "side" of the collapsed track.

§ 10.2. Aligning with 'auto' margins

This section is non-normative. The normative definition of how margins affect grid items is in §11 Grid Sizing.

Auto margins on grid items have an effect very similar to auto margins in block flow:

- During calculations of grid track sizes, auto margins are treated as '0'.
- 'auto' margins absorb positive free space prior to alignment via the box alignment properties.
- Overflowing elements ignore their 'auto' margins and overflow as specified by their box alignment properties.

§ 10.3. Row-axis Alignment: the 'justify-self' and 'justify-items' properties

Grid items can be aligned in the inline dimension by using the 'justify-self' property on the grid item or 'justify-items' property on the grid container, as defined in [CSS-ALIGN-3].

EXAMPLE 38

For example, for an English document, the inline axis is horizontal, and so the 'justify-*' properties align the grid items horizontally.

§ 10.4. Column-axis Alignment: the 'align-self' and 'align-items' properties

Grid items can also be aligned in the block dimension (perpendicular to the inline dimension) by using the 'align-self' property on the grid item or 'align-items' property on the grid container, as defined in [CSS-ALIGN-3].

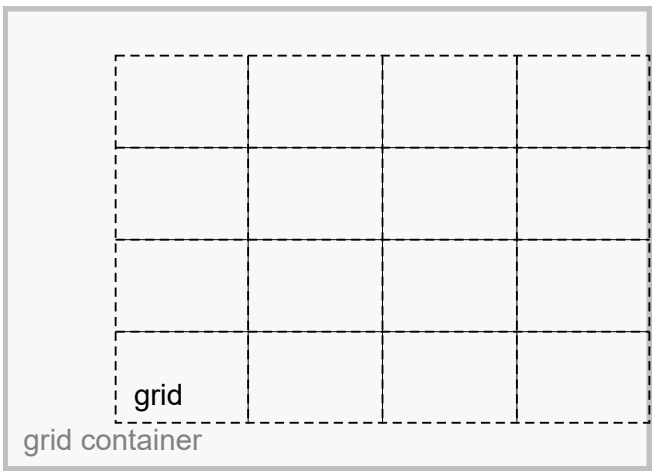
§ 10.5. Aligning the Grid: the 'justify-content' and 'align-content' properties

If the grid's outer edges do not correspond to the grid container's content edges (for example, if no columns are flex-sized), the grid tracks are aligned within the content box according to the 'justify-content' and 'align-content' properties on the grid container.

EXAMPLE 39

For example, the following grid is centered vertically, and aligned to the right edge of its grid container:

```
.grid {
  display: grid;
  grid: 12rem 12rem 12rem 12rem / 10rem 10rem 10rem 10rem;
  justify-content: end;
  align-content: center;
}
```



If there are no grid tracks (the explicit grid is empty, and no tracks were created in the implicit grid), the sole grid line in each axis is aligned with the start edge of the grid container.

Note that certain values of ‘justify-content’ and ‘align-content’ can cause the tracks to be spaced apart (‘space-around’, ‘space-between’, ‘space-evenly’) or to be resized (‘stretch’). If the grid is fragmented between tracks, any such additional spacing between those tracks must be suppressed.

EXAMPLE 40

For example, in the following grid, the spanning item's grid area is increased to accommodate the extra space assigned to the gutters due to alignment:

```
.wrapper {
  display: grid;
  /* 3-row / 4-column grid container */
  grid: repeat(3, auto) / repeat(4, auto);
  gap: 10px;
  align-content: space-around;
  justify-content: space-between;
}

.item1 { grid-column: 1 / 5; }
.item2 { grid-column: 1 / 3; grid-row: 2 / 4; }
.item3 { grid-column: 3 / 5; }
/* last two items auto-place into the last two grid cells */
```

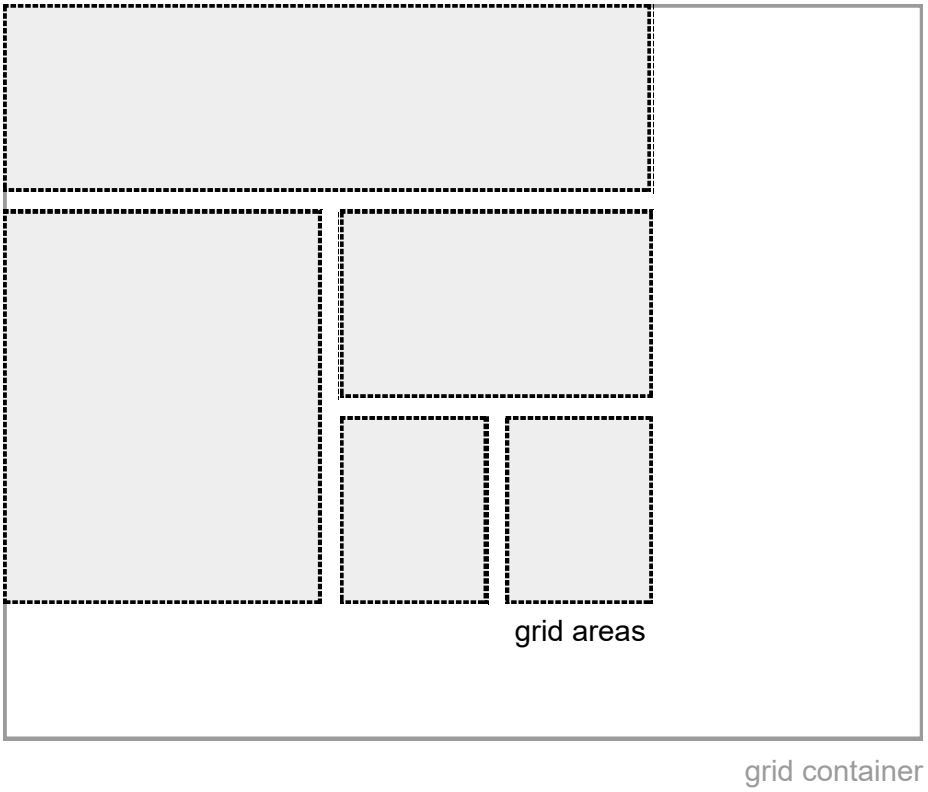


Figure 16 *Grid before alignment*

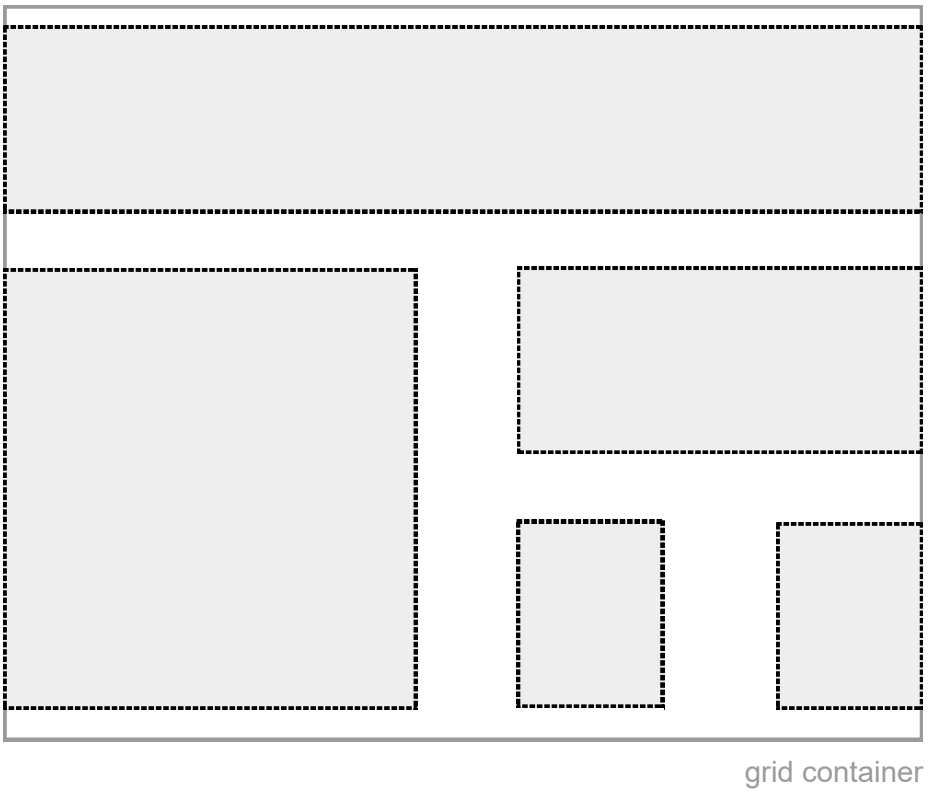


Figure 17 *Grid after alignment*

Note that alignment (unlike 'gap' spacing) happens after the grid tracks are sized, so if the track sizes are determined by the contents of the spanned item, it will gain excess space in the alignment stage to accommodate the alignment spacing.

§ 10.6. Grid Container Baselines

The first (last) baselines of a grid container are determined as follows:

1. If any of the grid items whose areas intersect the grid container's first (last) row participate in baseline alignment, the grid container's baseline set is generated from the shared alignment baseline of those grid items.
2. Otherwise, if the grid container has at least one grid item whose area intersects the first (last) row, the grid container's first (last) baseline set is generated from the alignment baseline of the first (last) such grid item (in row-major grid order). If the item has no alignment baseline in the grid's inline axis, then one is first synthesized from its border edges.
3. Otherwise, the grid container has no first (last) baseline set, and one is synthesized if needed according to the rules of its alignment context.

'grid-modified document order (grid order) is the order in which grid items are encountered when traversing the grid's grid cells. If two items are encountered at the same time, they are taken in order-modified document order.

When calculating the baseline according to the above rules, if the box contributing a baseline has an 'overflow' value that allows scrolling, the box must be treated as being in its initial scroll position for the purpose of determining its baseline.

When determining the baseline of a table cell, a grid container provides a baseline just as a line box or table-row does. [CSS21]

See CSS Writing Modes 3 §4.1 Introduction to Baselines and CSS Box Alignment 3 §9 Baseline Alignment Details for more information on baselines.

§ 11. Grid Sizing

This section defines the ***grid sizing algorithm***, which determines the size of all grid tracks and, by extension, the entire grid.

Each track has specified minimum and maximum sizing functions (which may be the same). Each sizing function is either:

- A ***fixed sizing function*** (<length> or resolveable <percentage>).
- An ***intrinsic sizing function*** ('min-content', 'max-content', 'auto', 'fit-content()').
- A ***flexible sizing function*** (<flex>).

The grid sizing algorithm defines how to resolve these sizing constraints into used track sizes.

§ 11.1. Grid Sizing Algorithm

1. First, the track sizing algorithm is used to resolve the sizes of the grid columns.

   If calculating the layout of a grid item in this step depends on the available space in the block axis, assume the available space that it would have if any row with a definite max track sizing function had that size and all other rows were infinite.

ISSUE 2 Would it help to have heuristics that attempt a more accurate initial estimate? E.g. assuming the available space that it would have as the maximum of:

- the sum of all definite track sizes that it spans (using the maximum of a track's min and max sizing functions, if both are definite, the argument to 'fit-content()' if that is definite).
- the item's 'min-content' size, if any track that it spans has a 'min-content' or 'fit-content()' sizing function.
- the item's automatic minimum size, if any track that it spans has an 'auto' min sizing function.
- infinity, if any track that it spans has a 'max-content' min sizing function or a 'max-content', 'auto', or <flex> max sizing function.

This is may reduce the amount of re-layout passes that are necessary, but will it produce a different or better result in any cases? Should we adopt it into the spec?

2. Next, the track sizing algorithm resolves the sizes of the grid rows, using the grid column sizes calculated in the previous step.
3. Then, if the min-content contribution of any grid items have changed based on the row sizes calculated in step 2, steps 1 and 2 are repeated with the new min-content contribution and max-content contribution (once only).

   This cycle is necessary for cases where the inline size of a grid item depends on the block size of its grid area. Examples include wrapped column flex containers ('flex-flow: column wrap'), orthogonal flows ('writing-mode'), and multi-column elements.

4. Finally, the grid container is sized using the resulting size of the grid as its content size, and the tracks are aligned within the grid container according to the 'align-content' and 'justify-content' properties.

   Note: This can introduce extra space between tracks, potentially enlarging the grid area of any grid items spanning the gaps beyond the space allotted to during track sizing.

Once the size of each grid area is thus established, the grid items are laid out into their respective containing blocks. The grid area's width and height are considered definite for this purpose.

Note: Since formulas calculated using only definite sizes, such as the stretch fit formula, are also definite, the size of a grid item which is stretched is also considered definite.

§ 11.2. Track Sizing Terminology

min track sizing function

If the track was sized with a 'minmax()' function, this is the first argument to that function. If the track was sized with a <flex> value or 'fit-content()' function, 'auto'. Otherwise, the track's sizing function.

max track sizing function

If the track was sized with a 'minmax()' function, this is the second argument to that function. Otherwise, the track's sizing function. In all cases, treat 'auto' and 'fit-content()' as 'max-content', except where specified otherwise for 'fit-content()'.

available grid space

Independently in each dimension, the available grid space is:

- If the grid container's size is definite, then use the size of the resulting content box.
- If the grid container is being sized under a min-content constraint or max-content constraint , then the available grid space is that constraint (and is indefinite).

> Note: 'auto' sizes that indicate content-based sizing (e.g. the height of a block-level box in horizontal writing modes) are equivalent to 'max-content'.

In all cases, clamp the available grid space by the grid container's min/max-width/height properties, if they are definite.

free space

Equal to the available grid space minus the sum of the base sizes of all the grid tracks (including gutters), floored at zero. If available grid space is indefinite, the free space is indefinite as well.

span count

The number of grid tracks crossed by a grid item in the applicable dimension.

§ 11.3. Track Sizing Algorithm

The remainder of this section is the ***track sizing algorithm***, which calculates from the min and max track sizing functions the used track size. Each track has a ***base size***, a <length> which grows throughout the algorithm and which will eventually be the track's final size, and a ***growth limit***, a <length> which provides a desired maximum size for the base size. There are 5 steps:

1. Initialize Track Sizes
2. Resolve Intrinsic Track Sizes
3. Maximize Tracks
4. Expand Flexible Tracks
5. Expand Stretched 'auto' Tracks

§ 11.4. Initialize Track Sizes

Initialize each track's base size and growth limit. For each track, if the track's min track sizing function is:

↪ **A fixed sizing function**

Resolve to an absolute length and use that size as the track's initial base size.

> Note: Indefinite lengths cannot occur, as they're treated as 'auto'.

↳ **An intrinsic sizing function**
↳ **A flexible sizing function**
Use an initial base size of zero.

For each track, if the track's max track sizing function is:

↳ **A fixed sizing function**
Resolve to an absolute length and use that size as the track's initial growth limit.

↳ **An intrinsic sizing function**
Use an initial growth limit of infinity.

↳ **A flexible sizing function**
Use the track's initial base size as its initial growth limit.

In all cases, if the growth limit is less than the base size, increase the growth limit to match the base size.

> Note: Gutters are treated as empty fixed-size tracks for the purpose of the track sizing algorithm.

§ 11.5. Resolve Intrinsic Track Sizes

This step resolves intrinsic track sizing functions to absolute lengths. First it resolves those sizes based on items that are contained wholly within a single track. Then it gradually adds in the space requirements of items that span multiple tracks, evenly distributing the extra space across those tracks insofar as possible.

> Note: When this step is complete, all intrinsic base sizes and growth limits will have been resolved to absolute lengths.

1. **Shim baseline-aligned items so their intrinsic size contributions reflect their baseline alignment.** For the items in each baseline-sharing group, add a "shim" (effectively, additional margin) on the start/end side (for first/last-baseline alignment) of each item so that, when start/end-aligned together their baselines align as specified.

   Consider these "shims" as part of the items' intrinsic size contribution for the purpose of track sizing, below. If an item uses multiple intrinsic size contributions, it can have different shims for each one.

   > ISSUE 3 Add example of multiple contributions/shims.

   > Note: Note that both baseline self-aligned and baseline content-aligned items are considered in this step, but they live in separate baseline-sharing groups. [CSS-ALIGN-3]

2. **Size tracks to fit non-spanning items:** For each track with an intrinsic track sizing function, consider the items in it with a span of 1:

   ↳ **For min-content minimums:**
   If the track has a 'min-content' min track sizing function, set its base size to the maximum of the items' min-content contributions.

↪ **For max-content minimums:**
If the track has a ‘max-content’ min track sizing function, set its base size to the maximum of the items’ max-content contributions.

↪ **For auto minimums:**
If the track has an ‘auto’ min track sizing function and the grid container is being sized under a min/max-content constraint, set the track’s base size to the maximum of its items’ min/max-content contributions, respectively.

Otherwise, set its base size to the maximum of its items’ min-size contributions. The ***min-size contribution*** of an item is the outer size that would result from assuming the item’s ‘min-width’ or ‘min-height’ value (whichever matches the relevant axis) as its specified size if its specified size (‘width’ or ‘height’, whichever matches the relevant axis) is ‘auto’, or else the item’s min-content contribution.

> Note: For items with a specified minimum size of ‘auto’ (the initial value), this is usually equivalent to a min-content minimum—but can differ in some cases, see §6.6 Automatic Minimum Size of Grid Items.

↪ **For min-content maximums:**
If the track has a ‘min-content’ max track sizing function, set its growth limit to the maximum of the items’ min-content contributions.

↪ **For max-content maximums:**
If the track has a ‘max-content’ max track sizing function, set its growth limit to the maximum of the items’ max-content contributions. For ‘fit-content()’ maximums, furthermore clamp this growth limit by the ‘fit-content()’ argument.

In all cases, if a track’s growth limit is now less than its base size, increase the growth limit to match the base size.

3. **Increase sizes to accommodate spanning items:** Next, consider the items with a span of 2 that do not span a track with a flexible sizing function, treating a min track sizing function of ‘auto’ as ‘min-content’/‘max-content’ when the grid container is being sized under a min/max-content constraint (respectively):
    1. **For intrinsic minimums:** First increase the base size of tracks with an intrinsic min track sizing function by distributing extra space as needed to accommodate these items’ min-size contributions.
    2. **For content-based minimums:** Next continue to increase the base size of tracks with a min track sizing function of ‘min-content’ or ‘max-content’ by distributing extra space as needed to account for these items' min-content contributions.
    3. **For max-content minimums:** Third continue to increase the base size of tracks with a min track sizing function of ‘max-content’ by distributing extra space as needed to account for these items' max-content contributions.
    4. If at this point any track’s growth limit is now less than its base size, increase its growth limit to match its base size.

5. **For intrinsic maximums:** Next increase the growth limit of tracks with an intrinsic max track sizing function by distributing extra space as needed to account for these items' min-size contributions. Mark any tracks whose growth limit changed from infinite to finite in this step as ***infinitely growable*** for the next step.

   ▶ **Why does the infinitely growable flag exist?**

6. **For max-content maximums:** Lastly continue to increase the growth limit of tracks with a max track sizing function of 'max-content' by distributing extra space as needed to account for these items' max-content contributions. However, limit the growth of any 'fit-content()' tracks by their 'fit-content()' argument.

Repeat incrementally for items with greater spans until all items have been considered.

4. If any track still has an infinite growth limit (because, for example, it had no items placed in it), set its growth limit to its base size.

Note: There is no single way to satisfy intrinsic sizing constraints when items span across multiple tracks. This algorithm embodies a number of heuristics which have been seen to deliver good results on real-world use-cases, such as the "game" examples earlier in this specification. This algorithm may be updated in the future to take into account more advanced heuristics as they are identified.

§ 11.5.1. Distributing Extra Space Across Spanned Tracks

To ***distribute extra space*** by increasing the affected sizes of a set of tracks as required by a set of intrinsic size contributions,

1. Maintain separately for each affected base size or growth limit a *planned increase*, initially set to 0. (This prevents the size increases from becoming order-dependent.)
2. For each considered item,
   1. **Find the space to distribute:** Subtract the corresponding size (base size or growth limit) of *every* spanned track from the item's size contribution to find the item's remaining size contribution. (For infinite growth limits, substitute the track's base size.) This is the space to distribute. Floor it at zero.

      ```
      extra-space = max(0, size-contribution - ∑track-sizes)
      ```

   2. **Distribute space to base sizes up to growth limits:** Find the *item-incurred increase* for each spanned track with an affected size by distributing the space equally among them, freezing tracks as their size reaches their growth limit (and continuing to grow the unfrozen tracks as needed).

      If a track was marked as infinitely growable for this phase, treat its growth limit as infinite for this calculation (and then unmark it).

      Note: If the affected size was a growth limit, this step has no effect.

3. **Distribute space beyond growth limits:** If space remains after all tracks are frozen, unfreeze and continue to distribute space to the *item-incurred increase* of…
   - when handling ‘min-content’ or ‘auto’ base sizes: any affected track that happens to also have an intrinsic max track sizing function; if there are no such tracks, then all affected tracks.
   - when handling ‘max-content’ base sizes: any affected track that happens to also have a ‘max-content’ max track sizing function; if there are no such tracks, then all affected tracks.
   - when handling any intrinsic growth limit: all affected tracks.

   For this purpose, ‘fit-content()’ tracks are treated as ‘max-content’ until they reach the limit specified as the ‘fit-content()’ argument, after which they are treated as having a fixed sizing function of that argument.

4. For each affected track, if the track’s *item-incurred increase* is larger than the track’s *planned increase* set the track’s *planned increase* to that value.

3. **Update the tracks' affected sizes** by adding in the *planned increase* so that the next round of space distribution will account for the increase. (If the affected size is an infinite growth limit, set it to the track’s base size plus the *planned increase*.)

§ 11.6. Maximize Tracks

If the free space is positive, distribute it equally to the base sizes of all tracks, freezing tracks as they reach their growth limits (and continuing to grow the unfrozen tracks as needed).

For the purpose of this step: if sizing the grid container under a max-content constraint, the free space is infinite; if sizing under a min-content constraint, the free space is zero.

If this would cause the grid to be larger than the grid container’s ‘max-width/height’, then redo this step, treating the available grid space as equal to the grid container’s content box size when it’s sized to its ‘max-width/height’.

§ 11.7. Expand Flexible Tracks

This step sizes flexible tracks using the largest value it can assign to an ‘fr’ without exceeding the available space.

First, find the used flex fraction:

If the free space is zero:
The used flex fraction is zero.

If the free space is a definite length:
The used flex fraction is the result of finding the size of an fr using all of the grid tracks and a space to fill of the available grid space.

If the free space is an indefinite length:
The used flex fraction is the maximum of:

- If the flexible track's flex factor is greater than one, the result of dividing the track's base size by its flex factor; otherwise, the track's base size.
- The result of finding the size of an fr for each grid item that crosses a flexible track, using all the grid tracks that the item crosses and a space to fill of the item's max-content contribution.

If using this flex fraction would cause the grid to be smaller than the grid container's 'min-width/height' (or larger than the grid container's 'max-width/height'), then redo this step, treating the free space as definite and the available grid space as equal to the grid container's content box size when it's sized to its 'min-width/height' ('max-width/height').

For each flexible track, if the product of the used flex fraction and the track's flex factor is greater than the track's base size, set its base size to that product.

§ 11.7.1. Find the Size of an 'fr'

This algorithm finds the largest size that an 'fr' unit can be without exceeding the target size. It must be called with a set of grid tracks and some quantity of ***space to fill***.

1. Let ***leftover space*** be the space to fill minus the base sizes of the non-flexible grid tracks.
2. Let ***flex factor sum*** be the sum of the flex factors of the flexible tracks. If this value is less than 1, set it to 1 instead.
3. Let the ***hypothetical fr size*** be the leftover space divided by the flex factor sum.
4. If the product of the hypothetical fr size and a flexible track's flex factor is less than the track's base size, restart this algorithm treating all such tracks as inflexible.
5. Return the hypothetical fr size.

§ 11.8. Stretch 'auto' Tracks

This step expands tracks that have an 'auto' max track sizing function by dividing any remaining positive, definite free space equally amongst them. If the free space is indefinite, but the grid container has a definite 'min-width/height', use that size to calculate the free space for this step instead.

§ 12. Fragmenting Grid Layout

Grid containers can break across pages between rows or columns and inside items. The 'break-*' properties apply to grid containers as normal for the formatting context in which they participate. This section defines how they apply to grid items and the contents of grid items.

The following breaking rules refer to the fragmentation container as the "page". The same rules apply in any other fragmentation context. (Substitute "page" with the appropriate fragmentation container type as needed.) See the CSS Fragmentation Module [CSS3-BREAK].

The exact layout of a fragmented grid container is not defined in this level of Grid Layout. However, breaks inside a grid container are subject to the following rules:

- The 'break-before' and 'break-after' properties on grid items are propagated to their grid row. The 'break-before' property on the first row and the 'break-after' property on the last row are propagated to the grid container.
- A forced break inside a grid item effectively increases the size of its contents; it does not trigger a forced break inside sibling items.
- Class A break opportunities occur between rows or columns (whichever is in the appropriate axis), and Class C break opportunities occur between the first/last row (column) and the grid container's content edges. [CSS3-BREAK]
- When a grid container is continued after a break, the space available to its grid items (in the block flow direction of the fragmentation context) is reduced by the space consumed by grid container fragments on previous pages. The space consumed by a grid container fragment is the size of its content box on that page. If as a result of this adjustment the available space becomes negative, it is set to zero.
- Aside from the rearrangement of items imposed by the previous point, UAs should attempt to minimize distortation of the grid container with respect to unfragmented flow.

§ 12.1. Sample Fragmentation Algorithm

This section is non-normative.

> This is a rough draft of one possible fragmentation algorithm, and still needs to be severely cross-checked with the [CSS-FLEXBOX-1] algorithm for consistency. Feedback is welcome; please reference the rules above instead as implementation guidance.

1. Layout the grid following the §11 Grid Sizing by using the fragmentation container's inline size and assume unlimited block size. During this step all 'grid-row' 'auto' and 'fr' values must be resolved.
2. Layout the grid container using the values resolved in the previous step.
3. If a grid area's size changes due to fragmentation (do not include items that span rows in this decision), increase the grid row size as necessary for rows that either:
    - have a content min track sizing function.
    - are in a grid that does not have an explicit height and the grid row is flexible.
4. If the grid height is 'auto', the height of the grid should be the sum of the final row sizes.
5. If a grid area overflows the grid container due to margins being collapsed during fragmentation, extend the grid container to contain this grid area (this step is necessary in order to avoid circular layout dependencies due to fragmentation).

> If the grid's height is specified, steps three and four may cause the grid rows to overflow the grid.

§ Acknowledgements

This specification is made possible by input from Erik Anderson, Rachel Andrew, Rossen Atanassov, Manuel Rego Casasnovas, Arron Eicholz, Javier Fernandez, Sylvain Galineau, Markus Mielke, Daniel Holbert, John Jansen, Chris Jones, Kathy Kam, Veljko Miljanic, Mats Palmgren, François Remy, Sergio Villar Senin, Jen Simmons, Christian Stockwell, Eugene Veselov, and the CSS Working Group members, with special thanks to Rossen Atanassov, Alex Mogilevsky, Phil Cupp, and Peter Salas of Microsoft for creating the initial proposal. Thanks also to Eliot Graff for editorial input.

§ Changes

This section documents the changes since previous publications.

§ Changes since the 29 September 2016 CR

A Disposition of Comments is also available.

There remains an open issue on percentages inside shrink-wrapped grid containers at the time of publication.

§ Major Changes

¶ • Deferred ‘subgrid’ feature to Level 2 due to lack of implementation and desire for further discussion. (Issue 958)

¶ • Removed ‘grid-row-gap’ and ‘grid-column-gap’ from the list of properties reset by the ‘grid’ shorthand. (Issue 1036)

¶ • Removed ‘grid-row-gap’, ‘grid-column-gap’, and ‘grid-gap’ properties, replacing with ‘row-gap’, ‘column-gap’, and ‘gap’ which are now defined in CSS Box Alignment. (Issue 1696)

¶ • Changed automatic sizing of grid items (such as images) with an intrinsic size or ratio so that they maintain their intrinsic size/ratio whenever the alignment properties are ‘normal’ (the default case). (Issue #523) See §6.2 Grid Item Sizing (vs. original).

¶ • Changed the behavior of <percentage> tracks inside a grid container whose size depends on the size of those tracks to match implementations by contributing their dimensions sized as ‘auto’ and subsequently resolve the percentage against the resulting grid container size rather than being treated exactly as an ‘auto’ track or having their size and that of the grid container increased from an ‘auto’ size in order to honor the percentage without overflow. This will frequently result in tracks overflowing the grid container and in the contents of tracks overflowing the tracks when <percentage> sizes are used in fit-content-sized grid containers such as ‘auto’-sized inline or floated grid containers. (To avoid this problem, use <flex> units instead, which are intended to maintain their ratios and not overflow when the grid is intrinsically-sized.)

If the size of the grid container depends on the size of its tracks, then the <percentage> must be treated as 'auto' for the purpose of calculating the intrinsic sizes of the grid container and then resolve against that size for the purpose of laying out the grid and its items . ~~The UA *may* adjust the intrinsic size contributions of the track to the size of the grid container and increase the final size of the track by the minimum amount that would result in honoring the percentage.~~

§ Significant Adjustments and Fixes

¶ • Applied flex factor clamping to 1 also to indefinite case (Issue 26, see discussion):

~~Each flexible track's base size divided by its flex factor.~~ If the flexible track's flex factor is greater than one, the result of dividing the track's base size by its flex factor; otherwise, the track's base size.

¶ • Better integrated 'stretch' sizing of grid tracks into the track sizing algorithm. (Issue 1150)

and the tracks are aligned within the grid container according to the 'align-content' and 'justify-content' properties. ~~Note: This can introduce extra space within or between tracks. When introducing space within tracks, only tracks with an 'auto' max track sizing function accept space.~~

> This can introduce extra space between tracks, potentially enlarging the grid area of any grid items spanning the gaps beyond the space allotted to during track sizing.

There are ~~4~~ 5 steps:

1. Initialize Track Sizes
2. Resolve Intrinsic Track Sizes
3. Maximize Tracks
4. Expand Flexible Tracks
5. Expand Stretched 'auto' Tracks

Stretch 'auto' Tracks

This step sizes expands tracks that have an 'auto' max track sizing function by dividing any remaining positive, definite free space equally amongst them.

¶ • Adjusted automatic minimum size of grid items to only trigger when spanning 'auto' tracks (Issue 12) and ensured that this correctly affects the transferred size when the item has an aspect ratio (Issue 11) so that this implied minimum does not end up forcing overflow:

> … the ‘auto’ value of ‘min-width’/‘min-height’ also applies an automatic minimum size in the specified axis to grid items whose ‘overflow’ is ‘visible’ and which span at least one track whose min track sizing function is ‘auto’ . …
>
> However, if the grid item spans only grid tracks that have a fixed max track sizing function, its ~~automatic minimum size~~ specified size and content size in that dimension (and the input to the transferred size in the other dimension) are further clamped to less than or equal to the stretch fit the grid area’s size (so as to prevent the automatic minimum size from forcing overflow of its fixed-size grid area) .

¶ • Adjusted automatic minimum size of grid items to use the transferred size in preference to the content size, rather than taking the smaller of the two. (Issue #1149)

> … ~~The effect is analogous to the automatic minimum size imposed on flex items.)~~ ~~[CSS-FLEXBOX-1]~~
>
> The automatic minimum size for a grid item in a given dimension is its specified size if it exists, otherwise its transferred size if that exists, else its content size, each as defined in [CSS-FLEXBOX-1]. However, if the grid item spans only grid tracks that have a fixed max track sizing function …

¶ • Fixed error in algorithm’s handling of ‘auto’ min track sizes where it didn’t correctly handle max-content constraints; and also made some editorial improvements. (Issue 5)

> 2. **Increase sizes to accommodate spanning items:** Next, consider the items with a span of 2 that do not span a track with a flexible sizing function , treating a min track sizing function of ‘auto’ as ‘min-content’/‘max-content’ when the grid container is being sized under a min/max-content constraint (respectively) :
>
>   1. …
>   2. **For content-based minimums:** Next continue to increase the base size of tracks with a min track sizing function of ‘min-content’ or ‘max-content’ ~~, and tracks with a min track sizing function of ‘auto’ if the grid container is being sized under a min-content constraint,~~ by distributing extra space as needed to account for these items' min-content contributions.
>   3. **For max-content minimums:** Third continue to increase the base size of tracks with a min track sizing function of ‘max-content’ ~~, and tracks with a max track sizing function of ‘auto’ if the grid container is being sized under a max-content constraint,~~ by distributing extra space as needed to account for these items' max-content contributions.

¶ • Fixed error in distribute extra space algorithm, where the accumulation was folded in per item rather than per track set; and where it was not clear that distributed space per item should be max()ed with the planned increase rather than added to it. (Issue #1729)

2. For each considered item,
    1. …
    2. **Distribute space to base sizes up to growth limits:** ~~Distribute the space equally to the planned increase of each spanned track with an affected size~~ Find the *item-incurred increase* for each spanned track with an affected size by distributing the space equally among them , freezing tracks as their size reaches their growth limit (and continuing to grow the unfrozen tracks as needed). …
    3. **Distribute space beyond growth limits:** If space remains after all tracks are frozen, unfreeze and continue to distribute space to the *item-incurred increase* of …
    4. For each affected track, if the track's *item-incurred increase* is larger than the track's *planned increase* set the track's *planned increase* to that value.
3. [numbering change from ~~2.4~~ to 3] **Update the tracks' affected sizes** by adding in the *planned increase*. (If the affected size is an infinite growth limit, set it to the track's base size plus the *planned increase*.)

¶ • Clarified that 'fit-content()' is not affected by 'stretch' 'align-content'/'justify-content'. (Issue 1732

> Represents the formula `min('max-content', max('auto', argument))`, which is calculated like ~~similar to 'auto' (i.e.~~ 'minmax(auto, max-content)' ~~)~~ , except that the track size is clamped at *argument* if it is greater than the 'auto' minimum.

§ Clarifications

- Clarified definition of min-size contribution. (Issue #507)

> Otherwise, set its base size to the maximum of its items' min-size contributions ~~:~~ . The min-size contribution of an item is the ~~value specified by its respective~~ outer size that would result from assuming the item's 'min-width' or 'min-height' value (whichever matches the relevant axis) as its specified size if its specified size ('width' or 'height', whichever matches the relevant axis) is 'auto', or else the item's min-content contribution.

¶ • Clarified that the space allotted through distributed alignment is part of the gutter, and collapses with it. (Issue #1140)

> A collapsed track is treated as having a fixed track sizing function of '0px', and the gutters on either side of it —including any space allotted through distributed alignment— collapse.

See also changes to CSS Box Alignment:

> Alignment Subject(s): ~~The non-collapsed grid tracks in the appropriate axis.~~ The grid tracks in the appropriate axis, with any spacing inserted between tracks added to the relevant gutters, and treating collapsed gutters as a single opportunity for space insertion.

- Miscellaneous trivial fixes: broken references, typos, etc.

§ 13. Privacy and Security Considerations

Grid introduces no new privacy leaks, or security considerations beyond "implement it correctly".

§ 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 41
>
> 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.**

§ 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.

§ Requirements for Responsible Implementation of CSS

The following sections define several conformance requirements for implementing CSS responsibly, in a way that promotes interoperability in the present and future.

§ 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 property 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.

§ Implementations of CR-level Features

Once a specification reaches the Candidate Recommendation stage, implementers should release an unprefixed implementation of any CR-level feature they can demonstrate to be correctly implemented according to spec, and should avoid exposing a prefixed variant of that feature.

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

§ Terms defined by reference

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

gap

generate baselines

grid-column-gap

grid-gap

grid-row-gap

gutter

justify-content

justify-items

justify-self

normal

row-gap

space-around

space-between

space-evenly

stretch (for align-content)

stretch (for align-self)

stretch (for justify-self)

synthesize baselines

[css-cascade-4] defines the following terms:

shorthand

sub-property

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

auto

automatic minimum size

center

content size

flex

flex container

flex item

flex-flow

order

order-modified document order

specified size

transferred size

[css-grid-2] defines the following terms:

subgrid

[css-inline-3] defines the following terms:

vertical-align

[css-multicol-1] defines the following terms:

multi-column element

[css-overflow-3] defines the following terms:

overflow

visible

[css-position-3] defines the following terms:

auto

bottom

left

position

relative

right

static

top

z-index

[css-pseudo-4] defines the following terms:

::first-letter

::first-line

[css-syntax-3] defines the following terms:

name code point

whitespace

[css-text-3] defines the following terms:

white-space

[css-writing-modes-4] defines the following terms:

block axis

block size

block-end

block-start

flow-relative

inline axis

inline size

inline-end

inline-start

orthogonal flow

physical

writing mode

writing-mode

[CSS21] defines the following terms:

margin

max-width

min-height

min-width

[CSS22] defines the following terms:

- clear
- float
- height
- width

[CSS3-BREAK] defines the following terms:

- break-after
- break-before
- fragment
- fragmentation container
- fragmentation context

[CSS3-DISPLAY] defines the following terms:

- anonymous
- block container
- block formatting context
- blockify
- containing block
- display
- inline formatting context
- text node
- text run

[CSS3-SIZING] defines the following terms:

- available space
- definite
- fit-content
- fit-content size
- indefinite
- max-content constraint
- max-content contribution
- max-content size
- min-content constraint
- min-content contribution
- min-content size
- stretch fit

[CSS3-WRITING-MODES] defines the following terms:

- direction
- end
- start

[CSS3VAL] defines the following terms:

- &&
- *
- +
- ,
- <custom-ident>
- <ident>
- <integer>
- <length-percentage>
- <length>
- <percentage>
- <string>
- ?
- calc()
- {a,b}
- |
- ||

[CSSOM] defines the following terms:

- resolved value special case property

[mediaqueries-4] defines the following terms:

- media query

§ References

§ Normative References

[CSS-ALIGN-3]

Elika Etemad; Tab Atkins Jr.. CSS Box Alignment Module Level 3. 6 September 2017. WD. URL: https://www.w3.org/TR/css-align-3/

[CSS-CASCADE-4]
Elika Etemad; Tab Atkins Jr.. CSS Cascading and Inheritance Level 4. 14 January 2016. CR. URL: https://www.w3.org/TR/css-cascade-4/

[CSS-FLEXBOX-1]
Tab Atkins Jr.; Elika Etemad; Rossen Atanassov. CSS Flexible Box Layout Module Level 1. 19 October 2017. CR. URL: https://www.w3.org/TR/css-flexbox-1/

[CSS-GRID-2]
CSS Grid Layout Level 2 URL: https://drafts.csswg.org/css-grid-2/

[CSS-INLINE-3]
Dave Cramer; Elika Etemad; Steve Zilles. CSS Inline Layout Module Level 3. 24 May 2016. WD. URL: https://www.w3.org/TR/css-inline-3/

[CSS-OVERFLOW-3]
David Baron; Florian Rivoal. CSS Overflow Module Level 3. 31 May 2016. WD. URL: https://www.w3.org/TR/css-overflow-3/

[CSS-POSITION-3]
Rossen Atanassov; Arron Eicholz. CSS Positioned Layout Module Level 3. 17 May 2016. WD. URL: https://www.w3.org/TR/css-position-3/

[CSS-PSEUDO-4]
Daniel Glazman; Elika Etemad; Alan Stearns. CSS Pseudo-Elements Module Level 4. 7 June 2016. WD. URL: https://www.w3.org/TR/css-pseudo-4/

[CSS-SYNTAX-3]
Tab Atkins Jr.; Simon Sapin. CSS Syntax Module Level 3. 20 February 2014. CR. URL: https://www.w3.org/TR/css-syntax-3/

[CSS-TEXT-3]
Elika Etemad; Koji Ishii. CSS Text Module Level 3. 22 August 2017. WD. URL: https://www.w3.org/TR/css-text-3/

[CSS-WRITING-MODES-4]
CSS Writing Modes Module Level 4 URL: https://drafts.csswg.org/css-writing-modes-4/

[CSS21]
Bert Bos; et al. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. 7 June 2011. REC. URL: https://www.w3.org/TR/CSS2

[CSS22]
Bert Bos. Cascading Style Sheets Level 2 Revision 2 (CSS 2.2) Specification. 12 April 2016. WD. URL: https://www.w3.org/TR/CSS22/

[CSS3-BREAK]
Rossen Atanassov; Elika Etemad. CSS Fragmentation Module Level 3. 9 February 2017. CR. URL: https://www.w3.org/TR/css-break-3/

[CSS3-DISPLAY]
Elika Etemad. CSS Display Module Level 3. 20 July 2017. WD. URL: https://www.w3.org/TR/css-display-3/

[CSS3-SIZING]
Elika Etemad. CSS Intrinsic & Extrinsic Sizing Module Level 3. 7 February 2017. WD. URL: https://www.w3.org/TR/css-sizing-3/

[CSS3-WRITING-MODES]

Elika Etemad; Koji Ishii. CSS Writing Modes Level 3. 15 December 2015. CR. URL: https://www.w3.org/TR/css-writing-modes-3/

[CSS3VAL]
Tab Atkins Jr.; Elika Etemad. CSS Values and Units Module Level 3. 29 September 2016. CR. URL: https://www.w3.org/TR/css-values-3/

[CSSOM]
Simon Pieters; Glenn Adams. CSS Object Model (CSSOM). 17 March 2016. WD. URL: https://www.w3.org/TR/cssom-1/

[MEDIAQUERIES-4]
Florian Rivoal; Tab Atkins Jr.. Media Queries Level 4. 5 September 2017. CR. URL: https://www.w3.org/TR/mediaqueries-4/

[RFC2119]
S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

§ Informative References

[CSS-MULTICOL-1]
Håkon Wium Lie; Florian Rivoal; Rachel Andrew. CSS Multi-column Layout Module Level 1. 5 October 2017. WD. URL: https://www.w3.org/TR/css-multicol-1/

[HTML5]
Ian Hickson; et al. HTML5. 28 October 2014. REC. URL: https://www.w3.org/TR/html5/

§ Property Index

Name	Value	Initial	Applies to	Inh.	%ages	Media	Animatable	Animation type	Canonical order	Computed value
'grid'	<'grid-template'> | <'grid-template-rows'> / [auto-flow && dense?] <'grid-auto-columns'>? | [auto-flow && dense?] <'grid-auto-rows'>? / <'grid-template-columns'>	see individual properties	grid containers	see individual properties	see individual properties	visual	see individual properties		per grammar	see individual properties

Name	Value	Initial	Applies to	Inh.	%ages	Media	Animat-able	Anim-ation type	Canonical order	Com puted value
‘grid-area’	<grid-line> [/ <grid-line>] {0,3}	see individual properties	grid items and absolutely-positioned boxes whose containing block is a grid container	see individual properties	see individual properties	visual		discrete	per grammar	see individual properties
‘grid-auto-columns’	<track-size>+	auto	grid containers	no	see Track Sizing	visual		discrete	per grammar	see Track Sizing
‘grid-auto-flow’	[row | column] || dense	row	grid containers	no	n/a	visual		discrete	per grammar	specified value
‘grid-auto-rows’	<track-size>+	auto	grid containers	no	see Track Sizing	visual		discrete	per grammar	see Track Sizing
‘grid-column’	<grid-line> [/ <grid-line>]?	see individual properties	grid items and absolutely-positioned boxes whose containing block is a grid container	see individual properties	see individual properties	visual		discrete	per grammar	see individual properties
‘grid-column-end’	<grid-line>	auto	grid items and absolutely-positioned boxes whose containing block is a grid container	no	n/a	visual		discrete	per grammar	specified value
‘grid-column-start’	<grid-line>	auto	grid items and absolutely-positioned boxes whose containing block is a grid container	no	n/a	visual		discrete	per grammar	specified value

Name	Value	Initial	Applies to	Inh.	%ages	Media	Animat-able	Anim-ation type	Canonical order	Com puted value
‘grid-row’	<grid-line> [/ <grid-line>]?	see individual properties	grid items and absolutely-positioned boxes whose containing block is a grid container	see individual properties	see individual properties	visual		discrete	per grammar	see individual properties
‘grid-row-end’	<grid-line>	auto	grid items and absolutely-positioned boxes whose containing block is a grid container	no	n/a	visual		discrete	per grammar	specified value
‘grid-row-start’	<grid-line>	auto	grid items and absolutely-positioned boxes whose containing block is a grid container	no	n/a	visual		discrete	per grammar	specified value
‘grid-template’	none | [<‘grid-template-rows’> / <‘grid-template-columns’>] | [<line-names>? <string> <track-size>? <line-names>?]+ [/ <explicit-track-list>]?	see individual properties	grid containers	see individual properties	see individual properties	visual	see individual properties		per grammar	see individual properties
‘grid-template-areas’	none | <string>+	none	grid containers	no	n/a	visual		discrete	per grammar	specified value

Name	Value	Initial	Applies to	Inh.	%ages	Media	Animat-able	Anim-ation type	Canonical order	Com puted value
'grid-template-columns'	none | <track-list> | <auto-track-list>	none	grid containers	no	refer to corresponding dimension of the content area	visual	as a simple list of length, percentage, or calc, provided the only differences are the values of the length, percentage, or calc components in the list		per grammar	As specified, with lengths made absolute
'grid-template-rows'	none | <track-list> | <auto-track-list>	none	grid containers	no	refer to corresponding dimension of the content area	visual	as a simple list of length, percentage, or calc, provided the only differences are the values of the length, percentage, or calc components in the list		per grammar	As specified, with lengths made absolute

§ Issues Index

ISSUE 1 If you notice any inconsistencies between this Grid Layout Module and the Flexible Box Layout Module, please report them to the CSSWG, as this is likely an error. ↵

ISSUE 2 Would it help to have heuristics that attempt a more accurate initial estimate? E.g. assuming the available space that it would have as the maximum of:

- the sum of all definite track sizes that it spans (using the maximum of a track's min and max sizing functions, if both are definite, the argument to 'fit-content()' if that is definite).
- the item's 'min-content' size, if any track that it spans has a 'min-content' or 'fit-content()' sizing function.
- the item's automatic minimum size, if any track that it spans has an 'auto' min sizing function.
- infinity, if any track that it spans has a 'max-content' min sizing function or a 'max-content', 'auto', or <flex> max sizing function.

This is may reduce the amount of re-layout passes that are necessary, but will it produce a different or better result in any cases? Should we adopt it into the spec?

↲

ISSUE 3 Add example of multiple contributions/shims. ↲